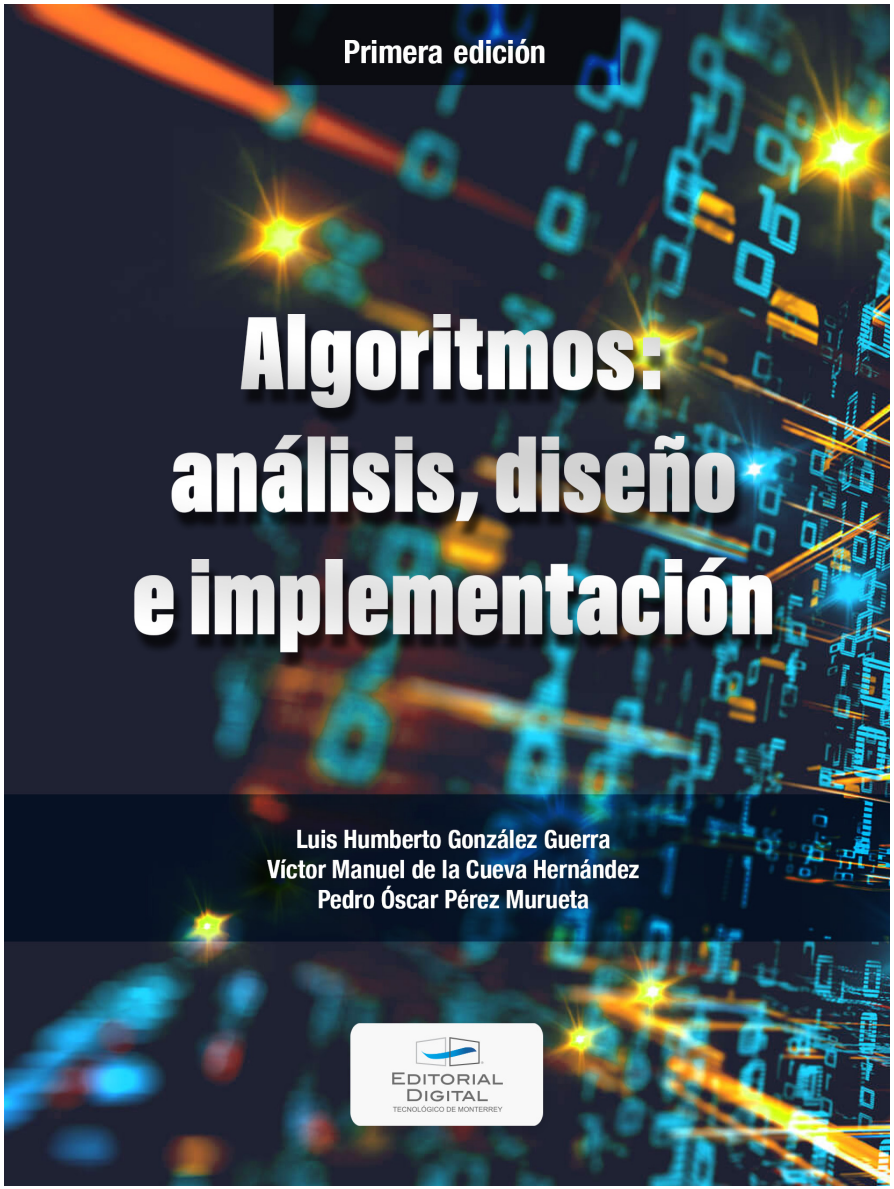
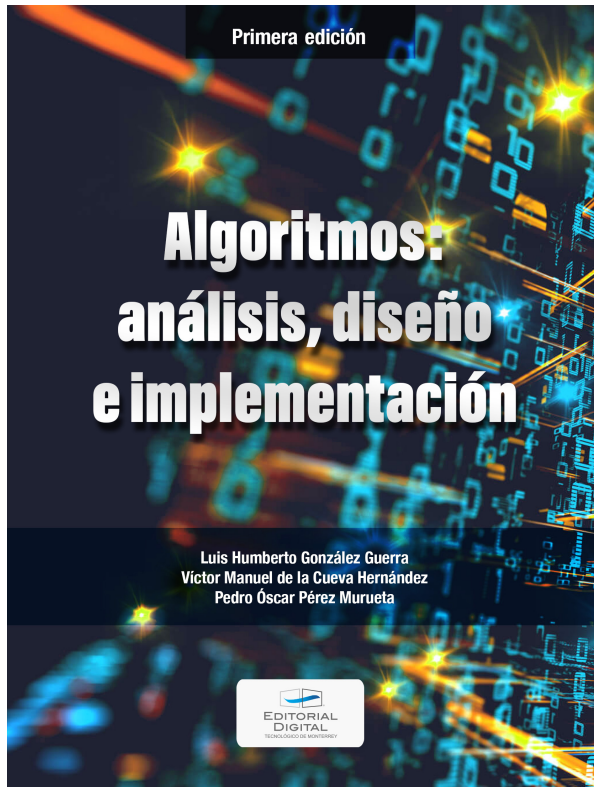




Primera edición
De venta en: Amazon Kindle, Apple Books, Google Books y Amazon.
Fragmento editado, diseñado, publicado y distribuido por el Instituto Tecnológico y de Estudios Superiores de Monterrey.

Ave. Eugenio Garza Sada 2501 Sur Col. Tecnológico C.P. 64849 |
Monterrey, Nuevo León | México.

Página

Índice

Índice 3

Capítulo 1. Herramientas matemáticas y análisis de algoritmos 11

1.1 Herramientas matemáticas básicas 11

1.1.1 Conjuntos 12

1.1.2 Relaciones 16

1.1.3 Funciones 18

1.1.4 Series y sucesiones 22

1.2 Análisis de algoritmos 25

1.2.1 Notación asintónica 25

1.2.1.1 Notación Θ grande (Big-Θ) 26

1.2.1.2 Notación O grande (Big-O) 28

1.2.1.3 Notación Ω grande (Big-Ω) 30

1.2.2 Análisis de algoritmos iterativos 31

1.2.2.1 La secuencia 31

1.2.2.2 La condicional 32

1.2.2.3 Los ciclos 32

1.2.3 Análisis de algoritmos recursivos 35

1.2.3.1 Planteamiento de fórmulas de recurrencia 37

1.2.3.2 Solución de fórmulas de recurrencia 39

1.2.3.3 Teorema maestro 46

1.2.4 Clasificación de problemas 49

1.3 Ejercicios del capítulo 1 50

Capítulo 2. Estructuras de datos 59

2.1 Introducción 59

2.1.1 Vectores 59

2.1.2 Filas (queue) 62

2.1.3 Pilas (stack) 64

2.1.4 Filas priorizadas (priority_queue) 65

2.1.5 Conjuntos (unordered_set) 68

2.1.6 Mapas (unordered_map) 70

2.2 Grafos 72

2.2.1 Terminología de grafos 72

2.2.2 Representación de un grafo 77

2.2.2.1 Matriz de adyacencia 78

2.2.2.2 Lista de adyacencias 79

2.2.2.3 Lista de arcos 80

2.2.3 Recorridos de un grafo 80

2.2.3.1 BFS – Breadth First Search (primero en anchura) 81

2.2.3.2 DFS – Depth First Search (primero en profundidad) 84

2.3 Conjunto disjunto (Disjoint-set) 87

Capítulo 3. Técnicas de diseño de algoritmos 95

3.1 Divide y vencerás 97

3.1.1 Ejemplo de funcionamiento 98

3.1.2 Análisis de su complejidad 105

3.1.3 El método maestro 110

3.1.4 Ejemplo de implementación 113

3.2 Algoritmos avaros 115

3.2.1 Ejemplo de funcionamiento 118

3.2.2 Analizando la complejidad 125

3.2.3 Ejemplo de implementación 126

3.3 Programación dinámica 127

3.3.1 Ejemplo de funcionamiento: la serie de Fibonacci 128

3.3.2 Análisis de complejidad 133

3.3.3 Ejemplo de aplicación de la programación dinámica 135

3.4 Backtracking 139

3.4.1 Ejemplo de funcionamiento 143

3.4.2 Análisis de complejidad 152

3.4.3 Ejemplo de implementación 153

3.5 Ramificación y poda 157

3.5.1 Ejemplo de funcionamiento 162

3.5.2 Análisis de complejidad 169

3.5.3 Ejemplo de implementación 169
3.6 Ejercicios del capítulo 3 173

Capítulo 4. Manejo de strings 177

4.1 Notación especial para strings 178
4.2 Función Z (Z-function) 180
4.2.1 Cálculo de la función Z 181
4.3 Problema de la coincidencia de un patrón 182
4.4 Algoritmo Knuth-Morris-Pratt (KMP) 186
4.4.1 Preprocesamiento del patrón 187
4.4.2 Algoritmo KMP completo 194
4.5 El problema del palíndromo más largo 201
4.5.1 Algoritmo naive para el palíndromo más largo 202
4.5.2 Algoritmo de Manacher 204
4.6 Hash Strings 219
4.7 Arreglo de sufijos (Suffix Array) 223
4.8 Trie 227
4.9 Ejercicios del capítulo 4 232

Capítulo 5. Aplicación de técnicas de diseño de algoritmos 235

5.1 Problema del substring común más largo 235
5.2 Problema de la subsecuencia común más larga 240

5.3 Problema del camino más corto 244
5.3.1 Algoritmo de Dijkstra 245
5.3.2 Algoritmo de Floyd 249
5.4 Problema de la mochila 254
5.4.1 Algoritmo con programación dinámica 255
5.4.2 Algoritmo de divide y vencerás 259
5.4.3 Algoritmo de backtracking 260
5.4.4 Algoritmo de ramificación y poda (Branch & Bound) 271
5.5 Problema del viajero 283
5.5.1 Algoritmo con ramificación y poda (Branch & Bound) 284
5.6 Árbol de mínima expansión 295
5.6.1 Algoritmo de Prim 296
5.6.2 Algoritmo de Kruskal 303
5.7 Problema de la multiplicación encadenada de matrices 310
5.7.1 Algoritmo de Godbole 312
5.8 Problema del BST óptimo 316
5.8.1 Algoritmo de Gilbert and Moore 318
5.9 Problema de coloreo de grafos 322
5.9.1 Algoritmo de Welsh Powell 324
5.10 Problema de flujo máximo 332
5.10.1 Algoritmo de Dinic 334

5.11 Algoritmos aleatorios 340
5.11.1 Algoritmos aleatorizados y "Divide y vencerás" 341
5.11.1.1 Encontrar la mediana 341
5.11.1.2 Quicksort 344
5.12 Ejercicios del capítulo 5 347

Capítulo 6. Geometría computacional 351

6.1 Proximidad e intersección 352
6.1.1 Punto 352
6.1.2 Línea 354
6.1.3 Segmento de línea 358
6.1.4 Polígono 361
6.2 Cascos convexos 366
6.2.1 Algoritmo de exploración de Graham 367
6.3 Diagramas de Voronoi y triangulación de Delaunay 371
6.3.1 Diagramas de Voronoi 371
6.3.2 Triangulación de Delaunay 375
6.4 Búsqueda geométrica 377
6.4.1 Árboles de rango (Range Trees) 378
6.4.2 Árboles Kd (Kd-Trees) 383
6.5 Ejercicios del capítulo 6 390

Capítulo 7. Técnicas de búsqueda avanzada 393

7.1 Backtracking con Bitmask 394

7.2 Encontrarse en el medio 400

7.3 Búsqueda A* 401

7.3.1 Una aplicación de A* 410

7.3.2 Implementación de A* 415

7.3.3 El algoritmo IDA* 419

7.4 Búsqueda de escalada 420

7.4.1 Aplicación de búsqueda de escalada 424

7.4.2 Implementación de búsqueda de escalada 428

7.5 Recocido simulado 430

7.5.1 Aplicación de recocido simulado 433

7.5.2 Implementación de recocido simulado 437

7.6 Ejercicios del capítulo 7 441

Créditos 444

Aviso legal 445

Capítulo 1. Herramientas matemáticas y análisis de algoritmos.

El análisis y diseño de algoritmos es una de las áreas más formales de las ciencias computacionales. Esta formalidad se refiere al uso de las matemáticas para llevar a cabo algún trabajo.

Aunque, con seguridad, ya tienes cierta formación en matemáticas computacionales y un curso previo de estructuras de datos y algoritmos, en donde se estudiaron la mayor cantidad de las herramientas matemáticas que se requieren en el análisis y diseño de algoritmos, es muy importante tenerlas presentes antes de iniciar estos nuevos temas. Por esta razón, se hará un pequeño repaso de las principales herramientas matemáticas que se van a utilizar:

- Conjuntos
- Funciones y relaciones
- Sumatorias y series
- Análisis de la complejidad

Es posible que en algunos casos se requieran algunas herramientas diferentes a las que estudiaremos, estas se dejarán para que el lector las investigue.

1.1 Herramientas matemáticas básicas

Las herramientas que se necesitan pertenecen, en su mayoría, al área de las Matemáticas Discretas, es decir, aquellas que trabajan con algún subconjunto de los números enteros o algún conjunto que tenga una correspondencia uno a uno con él.

1.1.1 Conjuntos

Un conjunto es una colección de elementos bien definida, en la cual no existen elementos repetidos y el orden entre ellos no importa. En la representación de estos se acostumbra llamar a los conjuntos con letras mayúsculas y a los elementos genéricos con letras minúsculas.

Un conjunto se puede definir de dos formas:

- **Enumerativa:** se colocan todos sus elementos entre llaves separados por comas. En otras palabras, se enumeran todos sus elementos y de ahí su nombre. Por ejemplo: la representación **A = {a, e, i, o, u}** es el conjunto **A** formado por 5 elementos que son las vocales.

- **Descriptiva:** se coloca entre llaves una descripción de los elementos que lo forman. Por ejemplo: **A = {x | x es una vocal}** se lee: "el conjunto **A** está formado por todas las **x** tal que **x** es una vocal". El pipe se lee "tal que". Este conjunto es igual al conjunto **A** del punto anterior.

Cuando un elemento **a** pertenece a un conjunto **S**, se escribe **a∈S**, y se lee "el elemento **a** pertenece al conjunto **S**" o simplemente "**a** pertenece a **S**". El símbolo ∈ se lee "pertenece a". Cuando un elemento **b** no pertenece al conjunto **S** se escribe **b∉S**. El símbolo ∉ se lee "no pertenece a".

El número de elementos que contiene un conjunto **A** se denomina **cardinalidad** de **A** y se representa como **|A|**. Si **A = {a,b,c,d}**, entonces la cardinalidad de **A** es 4, es decir, **|A| = 4**. Si la cardinalidad de un conjunto es 0 se dice que es un conjunto vacío y se representa como **{}** o con el símbolo especial ∅.

Si la cardinalidad de un conjunto es un entero **n** se dice que el **conjunto es finito** en caso contrario se dice que el **conjunto es infinito**.

Se dice que dos conjuntos **A** y **B** son iguales si contienen exactamente los mismos elementos (sin importar el orden), por ejemplo: el conjunto **A = {b, a, c}** es igual al conjunto **B = {c, b, a}**. Cuando los conjuntos son finitos y no de muy alta cardinalidad es relativamente fácil establecer si son iguales o no y se hace por comparación directa. Sin embargo, el establecimiento de igualdad no es tan simple si los conjuntos son infinitos y nuestro sentido común no ve cómo se cumple la igualdad , por ejemplo: si comparamos el conjunto de los números enteros representado por $\mathbb{Z}$, y el conjunto de los números naturales representado por $\mathbb{N}$ podríamos pensar que $|\mathbb{Z}| < |\mathbb{N}|$, sin embargo, se puede demostrar que son del mismo tamaño.

Si todos los elementos de un conjunto **A** están también en el conjunto **B**, se dice que **A** es un **subconjunto** de **B** y que **B** es un **superconjunto** de **A** y se escribe $A \subseteq B$. Si **A** es un subconjunto de **B** y no son iguales, se puede escribir $A \subset B$ y se dice que **A** es un subconjunto **propio** de **B**. Por definición el conjunto vacío es subconjunto de cualquier conjunto. Además, todo conjunto es subconjunto de sí mismo. El conjunto formado por todos los posibles subconjuntos de un conjunto **A** se llama conjunto potencia de **A**. Si $|A|=n$, la cardinalidad de su conjunto potencia es 2^n, por lo que es común representar el conjunto potencia de **A** como 2^A.

Si se quiere saber cuántos subconjuntos de cardinalidad **k** se pueden tener de un conjunto de cardinalidad **n**, la respuesta se obtiene por medio de las combinaciones de **n** en **k**, recordando que esta cantidad se obtiene de la siguiente forma:

$$C(n,k) = nCk = \binom{n}{k} = \frac{n!}{k!(n-k)!} \qquad (1.1)$$

Existen varias operaciones con conjuntos, algunas de las más comunes son:

- **Intersección:** la intersección de dos conjuntos **A** y **B** está formada por todos los elementos que están en **A** y en **B**, y se presenta como **A∩B**.

$$A \cap B = \{x \mid x \in A \text{ y } x \in B\} \qquad (1.2)$$

- **Unión:** la unión de dos conjuntos **A** y **B** está formada por todos los elementos que están en **A** o en **B** (o en ambos) y se representa como **A∪B**.

$$A \cup B = \{x \mid x \in A \text{ o } x \in B\} \qquad (1.3)$$

- **Diferencia:** la diferencia de dos conjuntos **A** y **B** está formada por todos los elementos que están en **A**, pero no están en **B** y se representa como **A − B**.

$$A - B = \{x \mid x \in A \text{ y } x \notin B\} \qquad (1.4)$$

- **Complemento:** el complemento de un conjunto **A** está formado por todos los elementos que no están en el conjunto **A** y se representa como **A'**. En realidad, esta definición informal daría un conjunto infinito como respuesta, puesto que cualquier cosa que no esté en **A** sería parte del complemento de **A**. Por esa razón, el complemento se limita a un conjunto universal **U** que contiene todos los posibles elementos que pueden formar conjuntos, así que lo que no esté en **A**, pero que esté en **U**, formará el complemento de **A**, lo cual se puede representar mejor por medio de la diferencia **U − A**.

$$\mathbf{A' = U - A = \{x \mid x \in U \text{ y } x \notin A\}} \qquad (1.5)$$

Por ejemplo: si **A = {a, b, c, d, e}** y **B = {d, e, f, g, h}: A∩B = {d, e} y A∪B = {a, b, c, d, e, f, g}** y también se pueden obtener las diferencias **A − B = {a, b, c} y B − A = {f, g, h}**. Para obtener el complemento de A o de B se tendría que definir el conjunto universal. Si **U = {a, b, c, d, e, f, g, h, i, j}** el complemento de **A** sería **A' = {f, g, h, i, j}**.

Una **tupla** es una sucesión finita de elementos donde el orden sí importa. Se representa por los elementos colocados entre paréntesis y separados por comas, por ejemplo: la tupla **(a,b,c)** es diferente de la tupla **(b, c, a)**. Si la tupla tiene dos elementos se le conoce como "par"; si tiene tres, se le conoce como "tercia"; pero si tiene más se complican los nombres, por lo que, en general, a una tupla de **k** elementos se le conoce como una **k-tupla**.

Es muy importante recordar que el orden en los elementos de una tupla sí importa. Por ejemplo: los puntos en un plano cartesiano se pueden representar por un par ordenado o 2-tupla, llamada coordenadas del punto, donde el primer elemento representa el valor de la coordenada en **x** (abscisa) y el segundo el de su coordenada en **y** (ordenada). Eso significa que no es lo mismo el par (2,3) que el par (3,2), ya que representan diferentes puntos en el plano.

Los elementos de una tupla pueden ser de diferentes tipos. En algunas aplicaciones de Ciencias Computacionales, como las Bases de Datos o la Ciencia de Datos, los datos se pueden representar por medio de una tupla, en donde el primer elemento corresponde al valor de una variable o característica (**feature**) en ciencia de datos, el segundo al valor de otra variable y así sucesivamente. Cada una de las variables puede ser un tipo muy diferente. Por ejemplo: si tenemos datos formados por dos varia-

bles, donde la primera representa el peso en kilogramos de una persona y la segunda su estatura en metros, el par (54.8, 1.63) representa a una persona de 54.8 kg de peso y 1.63 m de estatura.

Las tuplas nos permiten definir una operación más entre dos conjuntos, llamada productor cruz y representada con el símbolo **X**. El producto cruz de dos conjuntos **A** y **B**, representado por **A X B**, está formado por todas las parejas ordenadas en las que el primer elemento pertenece a **A** y el segundo pertenece a **B**. Por ejemplo: si **A = {a, b, c}** y **B = {1, 2}** el producto cruz **A X B** es:

$$\mathbf{A \times B = \{(a, 1), (a, 2), (b, 1), (b, 2), (c, 1), (c, 2)\}}$$

La cardinalidad del productor cruz de dos conjuntos, **A** y **B**, se obtiene mediante la multiplicación de las cardinalidades de cada conjunto, esto es, $\mathbf{|A \times B| = |A||B|}$.

Se puede tener un producto cruz entre más de dos conjuntos, lo que nos daría como resultado tuplas de más elementos. Por ejemplo: el producto cruz entre tres conjuntos **A**, **B** y **C** indicado por **A X B X C** sería el conjunto de todas las tuplas de tres elementos, en donde el primero es elemento de **A**, el segundo de **B** y el tercer de **C**. El producto cruz se puede extender a cualquier cantidad de conjuntos.

1.1.2 Relaciones

Una **relación** es cualquier subconjunto de un producto cruz, es decir, es un conjunto donde sus elementos son tuplas. Una relación muy común en las matemáticas es la de "menor que" entre los elementos del conjunto de los números enteros denotado por Z. Por ejemplo: son elementos de esta relación las parejas (2, 11) y (100, 1989) y no son miembros de esta relación las parejas (5, 3)

y (-1, -5). En este caso, el conjunto de esta relación es infinito y se define en forma descriptiva como:

$$\{(x, y) | x \in \mathbb{Z}, y \in \mathbb{Z} \text{ y } x < y\} \qquad (1.6)$$

En computación se utilizan mucho las relaciones. Por ejemplo: cada uno de los datos que se encuentra en una base de datos es en realidad una relación, lo mismo sucede con las tablas de ejemplos en ciencia de datos. En los lenguajes de programación lógicos, como Prolog, también se usan relaciones, por ejemplo: el predicado **Hermano(x, y)**, podría tener el significado de que **x** es hermano de **y**, esto es una relación. Muchos de los problemas que tratan de relaciones binarias se pueden representar por medio de un grafo, que es otra de las grandes áreas de las ciencias computacionales.

Las relaciones tienen ciertas propiedades que definen el tipo de relación de que se trata, algunas de las cuales se definen informalmente a continuación. Si la relación **R** es un subconjunto del producto cruz de **A X A**, es decir, **R ⊆ A X A**, se dice que la relación es:

- **Reflexiva:** si para toda **x ∈ A, (x, x) ∈ R**, es decir, si cada elemento de **A** se relaciona con él mismo.

- **Simétrica:** si cuando **(x, y) ∈R**, entonces también **(y, x) ∈ R**, es decir, si **x** se relaciona con **y**, entonces, también **y** se relaciona con **x**.

- **Transitiva:** si siempre que **(x, y) ∈ R** y **(y, z) ∈ R**, entonces también **(x, z) ∈ R**.

Por ejemplo: la relación "menor que" en los enteros de la que se habló anteriormente no es Reflexiva, porque ningún número

entero es menor que él mismo. Tampoco es simétrica porque si a es menor que **b**, **b** no es menor que **a**. Pero sí es transitiva, porque si **a** es menor que **b** y **b** es menor que **c**, entonces **a** es menor que **c**.

Hay un tipo especial de relación que cumple con las tres propiedades anteriores, es decir, es reflexiva, simétrica y transitiva. Cuando esto sucede se dice que se trata de una relación de equivalencia, normalmente representada por el símbolo ≡. Por ejemplo: la relación de igualdad en los números enteros es una relación de equivalencia.

Las relaciones de equivalencia son muy importantes en varias aplicaciones porque logran dividir al conjunto con el que se relacionan, digamos A en una **partición**. Una partición es una serie de conjuntos $\mathbf{A_1}$, $\mathbf{A_2}$, **...** $\mathbf{A_n}$, tales que nos hay intersección entre ninguno de ellos, es decir, $\mathbf{A_i} \cap \mathbf{A_j} = \varnothing$, para toda $\mathbf{i \neq j}$, y la unión de todos es igual a **A** y los elementos en cada uno de los conjuntos $\mathbf{A_i}$, son equivalentes entre sí.

1.1.3 Funciones

Las funciones son un tipo especial de relación. Dada una relación del conjunto **A** al conjunto **B**, dicha relación sería una función si cumple con la siguiente condición especial:

- Cada elemento de **A** debe estar relacionado con uno y solo un elemento de **B**.

Al conjunto **A** en una función se le conoce como **dominio** de la función, al conjunto **B** se le conoce como el **codominio** de la función y al subconjunto de elementos de **B** que se relacionan con alguno de los elementos de **A** se les llama el **rango** o la **imagen** de la función. Por la característica antes descrita que debe

cumplir una función se puede observar que la imagen puede no ser igual al codominio **B** porque es un subconjunto de **B**. Esto implica que todos los elementos de **A** deben estar en la relación, pero no todos los elementos de **B** deben estar en la misma.

Como los científicos que iniciaron la computación, al menos en la parte teórica, eran en su mayoría matemáticos, no pasó mucho tiempo para que las funciones quedaran incorporadas como una parte sumamente importante de los lenguajes de programación. De hecho, el objetivo fundamental del primer lenguaje de alto nivel que se creó, **Fortran** (cuyo nombre se forma de ***For****mula* ***Tran****slation*), era el de poder implementar fácilmente funciones matemáticas para su rápida evaluación en la computadora. Inclusive, poco tiempo después del desarrollo de **Fortan** salió **LISP** (de ***Lis****t* ***P****rocessing*), un lenguaje completamente basado en funciones matemáticas que dio lugar a un nuevo enfoque para crear lenguajes de programación llamado **Programación Funcional**. Posteriormente, la idea de incorporar las funciones a los lenguajes de programación ha sido inseparable hasta nuestros días. Hoy en día, todos los lenguajes de programación permiten crear funciones como parte de su sintaxis, aunque en los lenguajes orientados a objetos estas funciones están dentro del objeto mismo y se llaman **métodos**.

En computación, una función es un procedimiento que recibe ciertos valores y regresa un valor único. Los valores que recibe y el valor que regresa son los que forman la relación. Una función matemática hace exactamente lo mismo. Por ejemplo, si la función matemática es:

$$f(x) = x^2 + 2x - 1 \qquad (1.7)$$

Y está definida para los números reales $\mathbb{R}$, es decir, su dominio es $\mathbb{R}$ y si su codominio también es $\mathbb{R}$, esto es, la función es un subconjunto de $\mathbb{R} \times \mathbb{R}$, significa que la función recibe el valor de $\mathbf{x} \in \mathbb{R}$ y regresa el valor de la función (al cual normalmente le llamamos **y**, y $\mathbf{y} \in \mathbb{R}$). Si la $\mathbf{x = 2}$ la función $f(x)=f(2)$ regresaría el valor de $4 + 4 - 1 = 7$, lo que quiere decir que 7 es el elemento con el que se relaciona dentro del codominio, en otras palabras, indica que la relación (2,7) pertenece a la función. Usando un lenguaje de programación, por ejemplo: C++, una posible implementación de esta función sería la siguiente:

```
1   float f(float x){
2     return pow(x,2) + 2*x - 1;
3   }
```

En la línea 1, la palabra **float** inicial indica que el resultado que va a regresar esta función es **float**, el cual es equivalente a **real,** en el caso de las matemáticas, aunque en computación solo se trata de un subconjunto, ya que el conjunto completo es infinito. El valor que está entre paréntesis es su parámetro e indica el valor al que le queremos encontrar el elemento con el que se relaciona en el codominio. En la misma línea 1, la palabra **float** antes de la x indica que el tipo de su parámetro, esto es, también es un número real. En la línea 2, la palabra *return* dice que lo que sigue a continuación es el valor que va a regresar como el segundo de la relación. Este valor es encontrado al evaluar el valor del parámetro x dentro de la expresión indicada. La primera parte de la expresión **pow(x,2)** es a su vez una función llamada **pow**, que recibe dos parámetros y regresa el resultado de elevar el primero a la potencia que indica el segundo. Si corren esta función dentro de un programa de C++ y llaman a la función con un 2 el resultado obtenido será 7, como se explicó anteriormente, indicando que la pareja (2, 7) pertenece a la función.

Hay varias funciones especiales que son de gran utilidad en el análisis de algoritmos. A continuación, se definen algunas de las más importantes:

- **Piso:** la función piso de un número real **x** se representa como $\lfloor x \rfloor$ y se define como el entero más grande que es menor o igual a que **x**. Por ejemplo: $\lfloor 1.76 \rfloor = 1$. Algunas veces, a esta función también se le llama **trucamiento** y en matemáticas se conoce como **entero mayor** o **mayor entero**. Esta función va de los números reales (que recibe como parámetros) a los números enteros (que da como resultado), lo cual se expresa como $\mathbb{R} \rightarrow \mathbb{Z}$.

- **Techo:** la función techo de un número real **x** se representa como $\lceil x \rceil$ y se define como el entero menor que es mayor o igual a **x**. Es similar a la función piso solo que hacia el otro lado. Por ejemplo: $\lceil 1.15 \rceil = 2$.

- **Logaritmo:** la función logaritmo base **a** de un número **x** se representa como $log_a x$ y se define como el número al que tengo que elevar la base **a** para que dé como resultado el número **x**. Por ejemplo: $log_2 16 = 4$ porque $2^4 = 16$. Los logaritmos tienen muchas propiedades que nos ayudan a resolver un gran número de problemas de una forma más simple y son muy utilizados en el análisis asintótico de la complejidad de un algoritmo, sobre todo el logaritmo base 2. Por ejemplo, la siguiente propiedad nos ayuda a cambiar la base de un logaritmo:

$$log_c x = \frac{log_b x}{log_b c} \qquad (1.8)$$

1.1.4 Series y sucesiones

Un grupo de elementos que aparecen en un orden específico se llama **sucesión**. Por ejemplo, la sucesión de los números naturales:

$$0, 1, 2, 3, \ldots \qquad (1.9)$$

O la sucesión de los números primos:

$$1, 2, 3, 5, 7, 11, \ldots \qquad (1.10)$$

Las sucesiones son en realidad un tipo especial de función que tiene por dominio un conjunto de enteros consecutivos, los cuales indican las posiciones de los elementos dentro de la sucesión, por lo que se les conoce como **índices**. El **n-ésimo** término de una sucesión **S** se denomina $\mathbf{S_n}$ o con notación de función **S(n)**. Los elementos de una sucesión pueden ser identificados por la posición que guardan dentro de la sucesión, donde el primer índice puede ser cualquier entero y los que le siguen serán consecutivos.

Si una sucesión tiene un dominio finito se dice que se trata de una sucesión finita y si es infinito se trata de una sucesión infinita. El codominio de una sucesión puede contener cualquier tipo de elementos (por ejemplo: letras, números enteros, números reales, palabras, entre otros). Las sucesiones, como son funciones, pueden tener elementos repetidos. Como se puede ver en la sucesión de los números de Fibonacci, donde el 1 se repite 2 veces:

$$0, 1, 1, 2, 3, 5, 8, \ldots \qquad (1.11)$$

Otra característica es que pueden tener un nombre cualquiera. Por ejemplo: la sucesión de Fibonacci, que es infinita, se define por medio de la siguiente función recursiva:

$$fib(n) = \begin{cases} n & si\ n = \{0,1\} \\ fib(n-1) + fib(n-2) & si\ n > 1 \end{cases}, \text{ donde } n \geq 0 \qquad (1.12)$$

En la ecuación 1.12, la **n** indica la posición que guarda en la sucesión el número de Fibonacci que se desea encontrar, donde el primer elemento tiene el índice 0. La función **fib(n)** indica el número de Fibonacci que se encuentra en la posición con índice **n** dentro de la sucesión, por ejemplo, **fib(0) = 0, fib(1) = 1** y **fib(6) = 8**. La sucesión de los números naturales sería definida por la función **f(n) = n**, donde **n ≥ 0**. Desde luego, hay algunas sucesiones, para las que hasta el momento, no se tiene ninguna representación compacta en forma de función, tal es el caso de la sucesión de los números primos.

Hay varias operaciones que se pueden hacer sobre los elementos de las sucesiones. Por ejemplo: una de las más comunes en sucesiones numéricas es la suma de sus elementos, a la cual se le denomina **serie**. A esta operación se le llama sumatoria y se representa con la letra sigma mayúscula (∑) colocando en su parte baja el **nombre** usado para el índice y su primer valor (**valor inicial**) y en la parte superior el último valor (**valor final**) que toma el índice, pudiendo ser ∞. A continuación del símbolo de sumatoria se coloca la regla (función) que obtiene el valor del elemento buscado. Por ejemplo:

$$\sum_{i=1}^{\infty} 2i \qquad (1.13)$$

Al índice se le llama i, el primer valor que toma es 1 (**i = 1**) y el último valor que toma es ∞. La regla para obtener cada valor está definida por la función **2i** donde la **i** toma los valores indicados (de 1 a ∞), es decir, define la suma 2+4+6+…

Algunas veces, el resultado de una serie puede ser representado por una expresión que está en función de su límite superior.

Para algunas sumatorias infinitas también se puede saber cuál será su resultado.

Hay series muy usadas en el análisis de algoritmos, algunas de las cuales se muestran a continuación.

Polinómica

$$\sum_{i=1}^{n} i^d \approx \frac{n^{d+1}}{d+1} \qquad (1.14)$$

Geométrica

$$\sum_{i=0}^{d} ar^i = a\left(\frac{r^{d+1}-1}{r-1}\right) \qquad (1.15)$$

Aritmética: es aquella donde la resta de dos elementos consecutivos de la suma es una constante. Si se están sumando n elementos, donde el primero es $\mathbf{a_1}$ y el último es $\mathbf{a_n}$ la suma aritmética se obtiene con la expresión:

$$\frac{n(a_1+a_n)}{2} \qquad (1.16)$$

Logarítmica

$$\sum_{i=1}^{n} \log(i) \qquad (1.17)$$

Polinómica-Logarítmica

$$\sum_{i=1}^{n} i^d \log(i) \qquad (1.18)$$

1.2 Análisis de algoritmos

Un algoritmo es un procedimiento paso a paso que ayuda a solucionar un problema, tradicionalmente hablando desde el punto de programación, un algoritmo se puede expresar en pseudocódigo y este se convierte en la solución a un problema. Para un problema dado, pueden existir varias soluciones, es por eso que estas soluciones son analizadas para poder elegir la más eficiente. La eficiencia de los algoritmos se puede medir desde la prospectiva de tiempo de ejecución y de uso de memoria, donde estaremos hablando del tiempo de ejecución en términos del tamaño de la entrada (llamémosle n) y, naturalmente, estaremos analizando algoritmos para muy grandes; por otro lado, el uso de memoria se refiere a qué tanta memoria adicional requiere el algoritmo para ejecutarse. Desde el punto de vista de memoria, se mide la cantidad de memoria extra que se requiere para poder realizar el algoritmo. Este libro se estará enfocando más en la eficiencia del tiempo de ejecución y no tanto la eficiencia con el uso de memoria adicional, aunque sí lo estaremos mencionando.

1.2.1 Notación asintótica

Dado un algoritmo que tiene como entrada n datos y cuyo tiempo de ejecución en instrucciones máquina se puede expresar con el polinomio $3n^2 + 87n + 230$. El término $87n + 230$ crece más lentamente que el término $3n^2$ conforme va aumentando el tamaño de los datos de entrada y aunque para entradas chicas el término $3n^2$ tiene un mejor comportamiento, siempre se tomarán en cuenta los escenarios de entradas grandes. En la Figura 1.1 se puede apreciar el comportamiento de ambos términos.

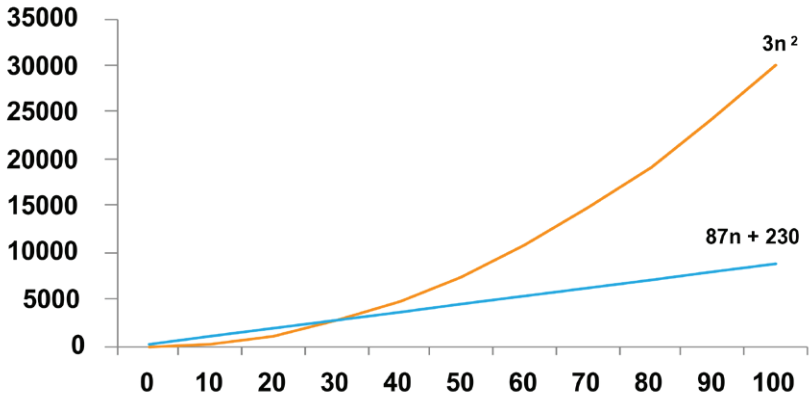


Figura 1.1 Crecimiento de los términos

En la Figura 1.1 se puede apreciar que el término que crece más rápidamente es el más significativo y al descartar los términos menos significativos y las constantes que contenga el término más significativo estaremos utilizando la **notación asintótica**. Existen notaciones: Notación Θ grande (Big-Θ), Notación O grande (Big-O) y Notación Omega Ω grande (Big-Ω).

1.2.1.1 Notación Θ grande (Big-Θ)

Dado un algoritmo básico, la búsqueda secuencial de un dato sobre un arreglo de enteros de tamaño n y que regrese el índice donde se localiza el dato -1 en caso de que no lo encuentre, el algoritmo sería como:

```
1 int busqSecuencial(int arreglo[], int n, int dato){
2   for (int i=0; i<n; i++) {
3     if (arreglo[i] == dato){
4       return i;
5     }
6   }
7   return -1;
8 }
```

El ciclo *for* entra en su peor caso n veces y esto sería cuando no localiza el dato o cuando este se encuentra en la última posición, por cada entrada al ciclo se realiza la comparación, la cual toma un tiempo constante, así como al incrementar el valor y volver a comparar y si lo encuentra, se regresa el valor del índice, la suma de todas estas operaciones sería una constante llamémosla **a**. Por lo tanto, en el peor caso tendríamos que el ciclo toma ***a * n*** y como en caso de que no se encuentre tenemos que regresará **-1**, y si asumimos que esto toma un valor constante que llamaremos **b** nos quedaría que la función de búsqueda secuencial toma ***a * n + b***. Para **n** muy grande, las constantes **a** y **b** se desprecian, así que, se dice que la búsqueda secuencial en su peor caso crece como el tamaño del arreglo, por lo que el tiempo de ejecución es Θ(**n**), y se dice que es *"Theta grande de n"* o simplemente *"Theta de n"*.

En general, cuando tenemos una función de tiempo de ejecución f(n), donde n es suficientemente grande, el tiempo de ejecución estará entre ***c1 * f(n) y c2 * f(n)***. Mientras existan contantes **c1** y **c2** que delimiten **f(n)** para n muy grandes decimos que el tiempo de ejecución del algoritmo es **(n)**. La Figura 1.2 muestra la notación Θ**(n).**

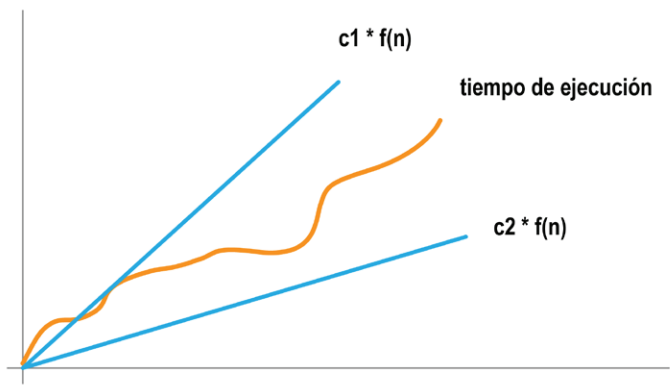


Figura 1.2 Notación Θ(n).

Se eliminan los factores constantes y términos de orden inferior, por lo cual podemos decir que el grado del polinomio es el que delimita la notación Θ **grande** con lo que estamos diciendo que tenemos una cota asintóticamente ajustada con respecto al tiempo de ejecución.

1.2.1.2 Notación O grande (Big-O)

La notación Θ grande sirve para acotar de manera asintótica el crecimiento de un tiempo de ejecución dentro de un rango, pero normalmente solo se desea saber el acotamiento superior. Por ejemplo, la búsqueda secuencial es Θ(n), pero no es correcto decir que ese tiempo es para todos los casos, ya que se puede encontrar el valor buscado en la primera posición, por lo tanto, se ejecuta en Θ(1), el tiempo de ejecución para la búsqueda secuencial nunca es peor que Θ(n), pero pudiera ser mejor que ese tiempo. Usaremos la notación O grande para las cotas superiores asintóticas para entradas muy grandes y se dice que el tiempo de ejecución es "O grande de *f(n)*", "O de *f(n)*" o simplemente "Orden de *f(n)*". Entonces, podemos decir que la búsqueda secuencial **O(n)**. La Figura 1.3 muestra la notación **O(n)**.

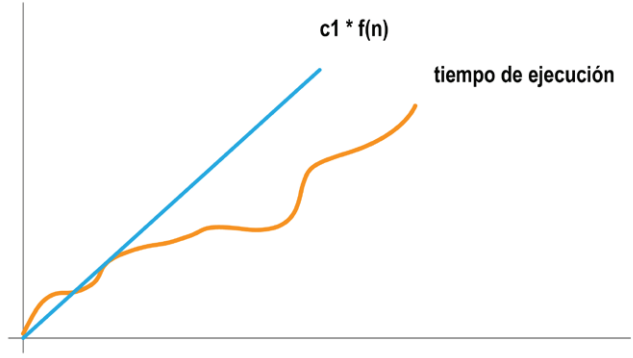


Figura 1.3 Notación **O(n)**

Analizando la definición de la notación Θ **grande** nos podemos dar cuenta que se parece a la notación O grande, excepto que la notación Θ**grande** acota el tiempo de ejecución de los algoritmos por arriba y por abajo, en cambio la notación **O grande** solo acota por arriba.

Las principales notaciones O grande que se tienen en los algoritmos son:

- O(1) – Constante (que no depende de la entrada)
- O(log n) – Logarítmica
- O(n) – Lineal
- O(n log n) – Lineal * Logarítmica
- O(n2) – Cuadrática
- O(n3) – Cúbica
- O(nc) – Polinomial (donde c es una constante)
- O(cn) – Exponencial (donde c es una constante)
- O(n!) - Factorial

Es esta notación la que se estará utilizando durante todo el libro. En la Figura 1.4 se puede apreciar el crecimiento de las notaciones **O grande** (**Big-O**) más básicas.

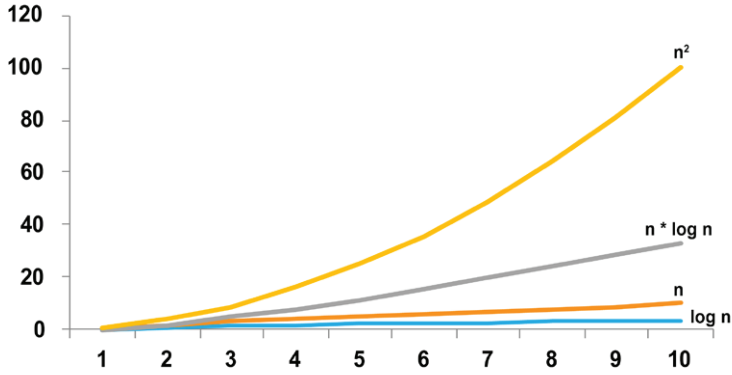


Figura 1.4 Crecimiento de las notaciones Big-O más básicas

1.2.1.3 Notación Ω grande (Big-Ω)

La Notación Ω grande es cuando se desea decir que un algoritmo toma por lo menos cierta cantidad de tiempo, sin querer ofrecer la cota superior. De esta manera, la notación Ω grande es para mostrar lo límites asintóticos inferiores. La Figura 1.5 muestra el comportamiento de la notación **Ω(n)**.

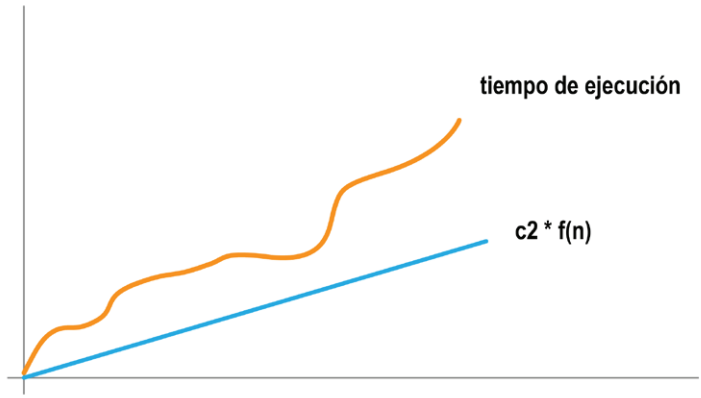


Figura 1.5 Notación **Ω(n)**

1.2.2 Análisis de algoritmos iterativos

La notación de Big-O de los algoritmos iterativos normalmente son polinomiales, para los cuales se tiene que hacer un análisis del algoritmo desde lo más interno hasta lo más externo, siguiendo tres reglas fundamentales: secuencia, condicional y ciclos.

1.2.2.1 La secuencia

Cuando se tiene una secuencia de estatutos de códigos, se selecciona la que impacta más fuertemente a la secuencia de instrucciones, es decir a la que tenga un orden mayor.

Ejemplo: si tuviéramos 3 ciclos en forma secuencial, Ciclo1 con ***O(n)***, Ciclo2 con ***O(n • log n)*** y Ciclo3 con ***O(log n)***, el Orden de la secuencia es ***O(n • log n)***.

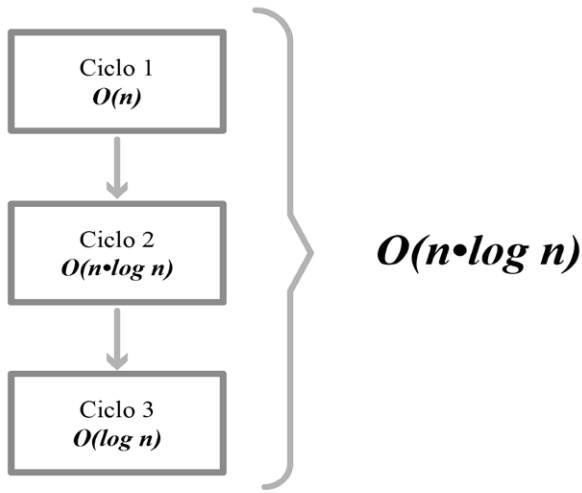


Figura 1.6

1.2.2.2 La condicional

Naturalmente, en la condicional (if) se tiene que realizar algo si la condición es verdadera y algunas veces se tiene que hacer algo si la condición es falsa. Siempre se tiene que escoger como el Orden de la condicional el mayor de estas partes (si solo existe la parte verdadera se toma directamente como el Orden de la condicional).

Ejemplo: si se tuviera una condicional que tiene en la parte verdadera un Orden ***O(log n)*** y en la parte falsa un Orden ***O(n^2)*** el Orden de la condicional será ***O(n^2)***.

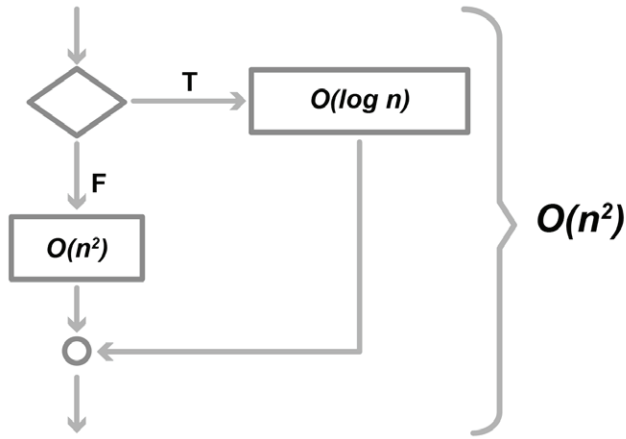


Figura 1.7

1.2.2.3 Los ciclos

Generalmente, en los ciclos se tiene una condición y algo que hacer en repetidas ocasiones mientras la condición sea verdadera. El orden de los ciclos será la cantidad de veces que realiza el ciclo por el orden de las operaciones internas que se encuentran dentro mismo del ciclo.

Ejemplo: si se tuviera un ciclo que se realiza ***n/2 veces*** y las operaciones internas tienen un Orden ***O(log n)*** el ciclo tendría un orden total de ***O(n * log n)***.

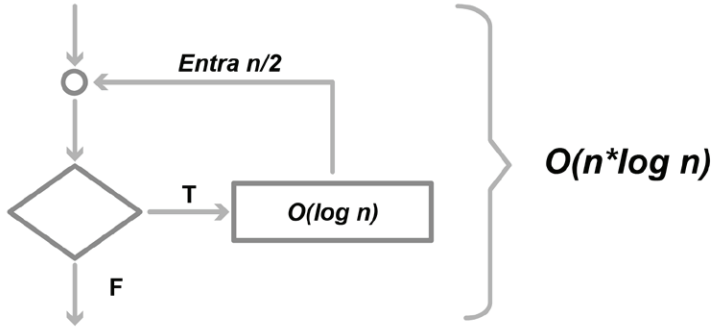


Figura 1.8

Cuando los ciclos tienen su rango de repetición en términos de n, entonces hay que revisar el comportamiento de la variable de control y el orden normalmente será:

- ***Lineal:*** cuando a la variable de control se le suma o se le resta una constante.

- ***Logarítmico:*** cuando a la variable de control se le divide o se le multiplica una constante.

Y al tener ciclos anidados se van multiplicando los órdenes. Es muy recomendable analizar los órdenes de las instrucciones desde las más internas hasta las más externas.

Ejemplos:

```
1 for (int i=0; i<10; i++) {
2   instruccion;
3 }
```

O(1)

```
1 for (int i=1234; i>0; i--) {
2   instruccion;
3 }
```

O(1)

```
1 for (int i=0; i<n; i++) {
2   instruccion;
3 }
```

O(n)

```
1 for (int i=n; i>0; i--) {
2   instruccion;
3 }
```

O(n)

```
1 for (int i=0; i<n/2; i++) {
2   instruccion;
3 }
```

O(n)

```
1 for (int i=n/5; i>0; i--) {
2   instruccion;
3 }
```

O(n)

```
1 for (int i=1; i<=n; i+=4) {
2   instruccion;
3 }
```

O(n)

```
1 for (int i=n; i>0; i-=5) {
2   instruccion;
3 }
```

O(n)

```
1 for (int i=1; i<=n/2; i+=5) {
2   instruccion;
3 }
```

O(n)

```
1 for (int i=n/7; i>0; i-=3) {
2   instruccion;
3 }
```

O(n)

```
1 for (int i=1; i<=n; i*2) {
2   instruccion;
3 }
```

O(log n)

```
1 for (int i=n; i>0; i/=3) {
2   instruccion;
3 }
```

O(log n)

```
1  if (n%=2){
2    for (int i=1; i<=n; i*=2){
3          instruccion;
4    }
5 }
6 else{
7    for (int i=4; i<100; i++){
8      instruccion;
9    }
10 }
```

O(log n)

```
1 if (n%=2){
2   for (int i=1; i<=n; i*=2){
3         instruccion;
4   }
5 }
6 else{
7   for (int i=4; i<n; i++){
8     instruccion;
9   }
10 }
```

O(n)

```
1 for (int i=1; i<=n; i*=2) {
2   for (int j=4; j<n; j++){
3     instruccion;
4   }
5 }
```

*O(n * log n)*

```
1 for (int i=1; i<=n; i+=2) {
2   for (int j=1; j<n; j++){
3     instruccion;
4   }
5 }
```

O(n^2)

1.2.3 Análisis de algoritmos recursivos

Para poder analizar la eficiencia de los algoritmos recursivos se tiene que ver la cantidad de llamadas recursivas en ejecución que se realizan, así como el comportamiento del parámetro de control de la función recursiva. Normalmente se comportan de una de las siguientes formas:

- ***O(n)*** – Cuando se tiene una sola llamada recursiva en ejecución y su parámetro de control se disminuye o incrementa en un valor constante.

- ***O(log_b n)*** – Cuando se tiene una sola llamada recursiva en ejecución y su parámetro de control se divide o se multiplica por un valor **b** constante.

- ***O(c^n)*** – Cuando se tienen **c** llamadas recursivas en ejecución y su parámetro de control se incrementa o decrementa en una constante.

- ***O($n^{log_b c}$)*** – Cuando se tienen **c** llamadas recursivas en ejecución y su parámetro de control se divide o se multiplica por un valor b constante.

Ejemplos:

```java
int sumaPares(int n){
  if (n <= 0)
    return 0;
  else if (n%2 == 0)
    return n+sumaPares(n-2);
  else
    return sumaPares(n-1);
}
```

O(n)

```java
int fact(int n){
  if (n <= 0)
    return 1;
  else
    return n*fact(n-1);
```

O(n)

```java
int contPot2(int n){
  if (n <= 0)
    return 0;
  else
    return 1+contPot2(n/2);
}
```

$O(\log_2 n)$

```java
int contPot5(int n){
  if (n <= 0)
    return 0;
  else
    return 1+contPot3(n/3);
}
```

$O(\log_3 n)$

```java
int algo(int n){
    if (n <= 0)
        return 400;
    else
        return algo(n-2)+algo(n-2)+algo(n-2)+algo(n-2);
}
```

$O(4^n)$

```java
int algo(int n){
    if (n <= 0)
        return 123;
    else
        return algo(n-4)+algo(n-4)+algo(n-4);
}
```

$O(3^n)$

```java
int algo(int n){
    if (n <= 0)
        return 400;
    else
        return 1+algo(n/2)+algo(n/2)+algo(n/2)+algo(n/2);
}
```

$O(n^{\log_2 4}) = O(n^2)$

```java
int algo(int n){
    if (n <= 0)
        return 123;
    else
        return 1+algo(n/4)+algo(n/4)+algo(n/4);
}
```

$O(n^{\log_4 3}) = O(n^{0.7924})$

1.2.3.1 Planteamiento de fórmulas de recurrencia

Para poder comprobar le eficiencia de los algoritmos recursivos es necesario platear el tiempo de ejecución en una fórmula de recurrencia en términos de T(n) e igualándola, por un lado, a la cantidad de llamadas recursivas en términos de la modificación de su parámetro de control y, por otro lado, al tiempo requerido del caso base en función de la cantidad de operaciones básicas que llevaría su proceso.

Ejemplo

En la solución del factorial recursivo se tiene que, matemáticamente el mínimo factorial que se puede tener es el factorial de 0 que da como resultado 1 y además se sabe que el cálculo del factorial para cualquier entero positivo es la multiplicación de ese entero por la llamada recursiva de la función factorial en términos del entero original menos 1. Su planteamiento en fórmula recursiva sería, por un lado, su caso base en donde solo toma 1 operación básica por el *return 1*; y, por otro lado, tiene en su parte recursiva 1 operación básica por el *return* y hay que agregarle el tiempo que toma la llamada a factorial para n-1. El algoritmo y la fórmula recursiva quedaría de la siguiente forma:

```
1 int fact(int n){
2   if (n <= 0)
3     return 1;
4   else
5     return n*fact(n-1);
```

$$T(n) = \begin{cases} 1, & n = 0 \\ 1 + T(n-1), & n > 0 \end{cases}$$

Ejemplo

Dado un algoritmo recursivo en donde en su caso base tenga una operación básica que regrese un número cualesquiera y en su parte recursiva tenga 2 llamadas recursivas con un valor menos que su valor original de n, el algoritmo y la fórmula recursiva sería como sigue:

```
1 int a(int n){
2   if (n == 0)
3     return 123;
4   else
5     return a(n-1)+a(n-1);
6 }
```

$$T(n) = \begin{cases} 1, & n = 0 \\ 1 + 2T(n-1), & n > 0 \end{cases}$$

Ejemplo

Otro ejemplo pudiera ser un algoritmo recursivo que en su caso base tenga una operación aritmética cualesquiera, la cual regresaría y en su parte recursiva tendría 3 llamadas recursivas de la mitad de n. El algoritmo y la fórmula recursiva quedaría como sigue a continuación:

```
1 int a(int n){
2   if (n == 0)
3     return 123*45;
4   else
5     return a(n/3)+a(n/3);
6 }
```

$$T(n) = \begin{cases} 1, & n = 0 \\ 1 + 2T(n/3), & n > 0 \end{cases}$$

1.2.3.2 Solución de fórmulas de recurrencia

Cuando se tiene una fórmula recursiva o de recurrencia, lo que se desea es tenerla en como una fórmula cerrada, esto es, sin recursividad. Para lograr esto se tiene que realizar el proceso de estar iterando en su recursividad hasta lograr checar su patrón de comportamiento para así poder llevarlo rápidamente al caso base y ahí ya obtener la fórmula cerrada.

Ejemplo

Retomando el ejemplo 1.1 del punto anterior, el algoritmo de la factorial tiene una sola llamada recursiva con término del parámetro de control de n-1, por lo que debe de dar un **O(n)**, la demostración se basa en la solución de su fórmula recursiva.

$$T(n) = \begin{cases} 1, & n = 0 \\ 1 + T(n-1), & n > 0 \end{cases}$$

La base de fórmula recursiva será:

$$T(n) = T(n-1) + 1 \qquad (1.19)$$

Entonces, se tiene que buscar quién es T(n-1) y se logra sustituyendo con **n-1** en la fórmula general (1.19) donde diga **n**, obteniendo:

$$T(n-1) = T(n-2) + 1 \qquad (1.20)$$

Ahora se sustituye la fórmula (1.20) en la fórmula base (1.19) quedando:

$$T(n) = (T(n-2) + 1) + 1 \qquad (1.21)$$

$$T(n) = T(n-2) + 2 \qquad (1.22)$$

Ahora se buscará quién es T(n-2) y se logra sustituyendo con **n-2** en la fórmula general (1.19) donde diga **n** y se obtiene:

$$T(n-2) = T(n-3) + 1 \qquad (1.23)$$

Ahora se sustituye la fórmula (1.23) en la fórmula (1.22), quedando

$$T(n) = (T(n-3) + 1) + 2 \qquad (1.24)$$

$$T(n) = T(n-3) + 3 \qquad (1.25)$$

Se puede ver que si le rebaja 1 al término interno de T se le suma 1, si se le rebaja 2 se le suma 2 y si se le rebaja 3 se le suma 3, así el patrón es que por cada contante que se le rebaje al valor interno de T se le suma esa misma contante, ahora bien se tiene que llevar al caso base para que pare la recursividad, el caso base es T(0), ahí ya no hay recursividad y sabiendo que T(0) es 1, por lo tanto queda:

$$T(n) = T(n-n) + n \qquad (1.26)$$

$$T(n) = T(0) + n \qquad (1.27)$$

Ejemplo

En el caso del ejemplo 1.2 del punto anterior, el algoritmo tiene 3 llamadas recursivas rebajándole la constante de 1 en cada una de sus llamadas, por lo que debe de dar un **O(3ⁿ)**; la demostración se basa en la solución de su fórmula recursiva.

$$T(n) = \begin{cases} 1, & n = 0 \\ 1 + 3T(n-1), & n > 0 \end{cases}$$

La base de fórmula recursiva será:

$$T(n) = 3T(n-1) + 1 \qquad (1.28)$$

Entonces, se tiene que buscar quién es T(n-1) y se logra sustituyendo **n-1** en la fórmula general (1.28) donde diga **n** y se obtiene:

$$T(n-1) = 3T(n-2) + 1 \qquad (1.29)$$

Ahora se sustituye la fórmula (1.29) en la fórmula general (1.28) quedando:

$$T(n) = 3(3T(n-2) + 1) + 1 \qquad (1.30)$$

$$T(n) = 3^2T(n-2) + 3 + 1 \qquad (1.31)$$

Ahora se buscará quién es T(n-2) y se logra sustituyendo **n-2** en la fórmula general (1.28) donde diga **n** y se obtiene:

$$T(n-2) = 3T(n-3) + 1 \qquad (1.32)$$

Ahora se sustituye la fórmula (1.32) en la fórmula general (1.27), quedando:

$$T(n) = 3^2(3T(n-3) + 1) + 3 + 1 \qquad (1.33)$$

$$T(n) = 3^3T(n-3) + 3^2 + 3 + 1 \qquad (1.34)$$

$$T(n) = 3^3T(n-3) + 3^2 + 3^1 + 3^0 \qquad (1.35)$$

Se puede ver que por cada constante que se le rebaja al término interno de T se le suma 1 a la potencia de 3 que lo está multiplicando y se agrega el término 3 elevado a la constante -1 a la sumatoria que se está generando, ahora bien se tiene que llevar al caso base para que pare la recursividad, el caso base es T(0), el cual tiene un valor de 1 y ahí ya no hay recursividad, por lo tanto queda:

$$T(n) = 3^nT(n-n) + 3^{n-1} + 3^{n-2} + \ldots + 3^2 + 3^1 + 3^0 \qquad (1.36)$$

quedando:

$$T(n) = \sum_{i=0}^{n} 3^i \qquad (1.37)$$

y sabiendo que

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1} \qquad (1.38)$$

por lo tanto, queda:

$$T(n) = \frac{3^{n+1} - 1}{3 - 1} \rightarrow O(3^n)$$

Ejemplo

Para el ejemplo 1.3 del punto anterior, el algoritmo tiene 3 llamadas recursivas de la mitad de la n cada una de sus llamadas, por lo que debe de dar un $\mathbf{O(n^{\log_2 3})}$, la demostración se basa en la solución de su fórmula recursiva.

$$T(n) = \begin{cases} 1, & n = 1 \\ 1 + 3T(n/2), & n > 0 \end{cases}$$

La base de fórmula recursiva será:

$$T(n) = 3T(n/2) + 1 \qquad (1.39)$$

Se recomienda que cuando se tenga para la llamada recursiva el valor de la **n** dividido entre una constante se sustituya la **n** por la constante elevada a la **k**, esto debido a que debe llegar al caso base en una determinada cantidad (**k**) de llamadas recursivas, por lo tanto:

$$\frac{n}{2^k} = 1 \text{ y despejando } n = 2^k \qquad (1.40)$$

Por lo tanto, sustituyendo **n** por $\mathbf{2^k}$ en la fórmula (1.39) y sabiendo que $2^k/2 = 2^{k-1}$, queda:

$$T(2^k) = 3T(2^{k-1}) + 1 \qquad (1.41)$$

Entonces se tiene que buscar quién es $T(2^{k-1})$, y se logra sustituyendo por $\mathbf{2^{k-1}}$ en la fórmula general (1.41) donde diga $\mathbf{2^k}$ y se obtiene:

$$T(2^{k-1}) = 3T(2^{k-2}) + 1 \qquad (1.42)$$

Ahora se sustituye en la fórmula general (1.41) quedando:

$$T(2^k) = 3(3T(2^{k-2}) + 1) + 1 \qquad (1.43)$$

$$T(n) = 3^2T(2^{k-2}) + 3 + 1 \qquad (1.44)$$

Ahora se buscará quién es $T(3^{k-2})$, y se logra sustituyendo $\mathbf{2^{k-2}}$ en la fórmula general (1.41) donde diga $\mathbf{2^k}$ y se obtiene:

$$T(2^{k-2}) = 3T(2^{k-3}) + 1 \qquad (1.45)$$

Ahora se sustituye en la fórmula general (1.43) quedando:

$$T(n) = 3^2(3T(2^{k-3}) + 1) + 3 + 1 \qquad (1.46)$$

$$T(n) = 3^3T(2^{k-3}) + 3^2 + 3 + 1 \qquad (1.47)$$

$$T(n) = 3^3T(2^{k-3}) + 3^2 + 3^1 + 3^0 \qquad (1.48)$$

Se puede ver que por cada constante que se le rebaja al término interno de T se le suma 1 a la potencia de 3 que lo está multiplicando y se agrega el término 3 elevado a la constante -1 a la sumatoria que se está generando, ahora bien se tiene que llevar al caso base para que pare la recursividad, el caso base es T(1) y ahí ya no hay recursividad, además sabiendo que $2^0 = 1$ queda:

$$T(2^k) = 3^kT(2^{k-k}) + 3^{k-1} + 3^{k-2} + \ldots + 3^2 + 3^1 + 3^0 \qquad (1.49)$$

quedando:

$$T(2^k) = \sum_{i=0}^{k} 3^i \qquad (1.50)$$

y sabiendo que:

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1} \qquad (1.51)$$

queda:

$$T(2^k) = \frac{3^{k+1}-1}{3-1} = \frac{3*3^k-1}{2} \qquad (1.52)$$

despejando la fórmula de $\mathbf{n = 2^k}$ (1.38) y despejando la **k** queda que:

$$k = \log_2 n \qquad (1.53)$$

sustituyendo la **k** queda:

$$T(n) = \frac{3 * 3^{log_2 n} - 1}{2} \qquad (1.54)$$

por la propiedad de los logaritmos se sabe que:

$$x^{log_b y} = y^{log_b x} \qquad (1.55)$$

por lo tanto, queda:

$$T(n) = \frac{3n^{\log_2 3} - 1}{2} \longrightarrow O(n^{\log_2 3}) = O(n^{1.5849})$$

1.2.3.3 Teorema maestro

Con el teorema maestro se puede deducir la complejidad en el tiempo de ejecución de muchos de los algoritmos recursivos rápidamente.

Teorema maestro

Sea el tiempo de ejecución del algoritmo recursivo de la siguiente forma:

$$T(n) = a\,T\left(\frac{n}{b}\right) + O(n^c) \qquad (1.56)$$

Donde a, b y c son constantes.

La complejidad en el tiempo de ejecución del algoritmo será:

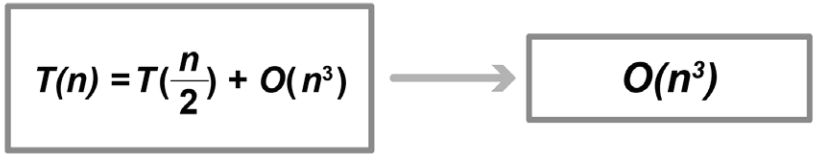
$$T(n) = \begin{cases} O(n^c), & \log_b a < c \\ O(n^c \log_b n), & \log_b a = c \\ O(n^{\log_b a}), & \log_b a > c \end{cases} \qquad (1.57)$$

Ejemplo

Teniendo un algoritmo cuya fórmula recursiva sea:

$$T(n) = T\left(\frac{n}{2}\right) + O(n^3) \qquad (1.58)$$

Por lo tanto, **a = 1, b = 2** y **c = 3** dando el $\log_2 1 = 0$ así, $\log_b a < c$ y la complejidad en el tiempo de ejecución es:

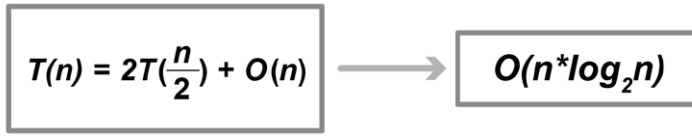


Ejemplo

El algoritmo de *Merge Sort* tiene una fórmula recursiva de:

$$T(n) = 2\,T\left(\frac{n}{2}\right) + O(n) \qquad (1.59)$$

Por lo tanto, **a = 2, b = 2** y **c = 1** dando el $log_2 2 = 1$ así $log_b a = c$ y la complejidad en el tiempo de ejecución es:

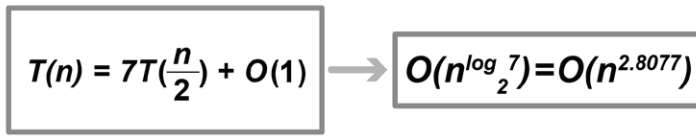


Ejemplo

El algoritmo de Stassen para multiplicar matrices tiene una fórmula recursiva de:

$$T(n) = 7\,T\left(\frac{n}{2}\right) + O(1) \qquad (1.60)$$

Por lo tanto, **a = 7, b = 2** y **c = 0** dando el $log_2 7 = 2.80$ así $log_b a > c$ y la complejidad en el tiempo de ejecución es:



Existen muchos problemas que no pueden ser trabajados con el teorema maestro, como los exponenciales, pero en muchas ocasiones es muy benéfico poder obtener rápidamente la solución a una función recursiva.

1.2.4 Clasificación de problemas

La teoría de la clasificación de problemas está basada en qué tan difícil se puede resolver el problema y básicamente puede ser:

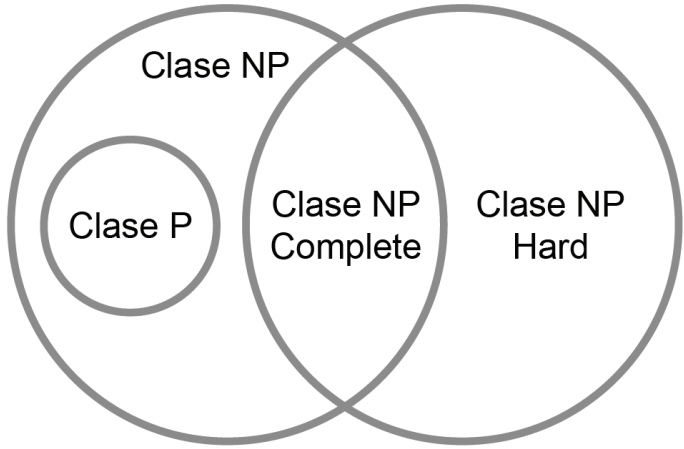


Figura 1.9

- ***Problemas de Clase NP:*** Un problema que puede ser resuelto por un tiempo No-determinístico.
  - Un algoritmo No-Determinístico es un algoritmo que consta de dos fases: suponer y comprobar.
  - Si la complejidad temporal en la etapa de comprobación de un algoritmo no-determinístico es polinomial, entonces este algoritmo se denomina algoritmo polinomial no-determinístico.
- ***Problemas de Clase P:*** Un problema es asignado a la clase P (Tiempo Polinomial) cuando el tiempo de esta solución está en una potencia constante del tamaño de la entrada $(\mathbf{n^k})$

- ***Problemas de Clase NP-Complete:*** Un problema **B** es NP-Complete si cumple con las siguientes dos características:
  - Ser un problema de **Clase NP.**
  - Para otro problema **A** en **NP**, A se reduce polinomialmente en **B**.
- ***Problemas de Clase NP-Hard:*** Un problema se dice que es NP-Hard si solo cumple con la característica de:
  - Para otro problema **A** en **NP**, A se reduce polinomialmente en **B**.

Reducción polinomial es una manera de relacionar dos problemas, cuando se tiene un algoritmo **A** que se resuelve en tiempo polinomial hay que ver la forma de que la salida de este pueda ser la entrada del algoritmo **B**, la idea es que la solución del algoritmo **A** apoye a la solución del algoritmo **B**.

1.3 Ejercicios del capítulo 1

1. Dados los conjuntos A = {1, 3, 5, 7} y B = {2, 3, 4, 7, 9}, encuentra los conjuntos:

a. $A \cup B$

b. $A \cap B$

c. $A - B$

d. $B - A$

e. A X B

f. B X A

g. 2A

2. Sea el conjunto C = {x | x es un par positivo y x ≤ 9}, escribe en forma enumerativa.

3. Sea el conjunto $D=\{x/x\in\mathbb{R}\ y\ 0\leq x\leq 1\}$. ¿Es posible escribirlo en forma enumerativa? Explica por qué.

4. Sea el conjunto A = {a, b, c} y la relación R: A →A, definida como:

R = {(a, a), (a, b), (a, c), (b, a), (b, b), (b, c), (c, a), (c, b), (c, c)}

Menciona si R tiene cada una de las siguientes propiedades, explicando la razón:

a. Reflexiva

b. Simétrica

c. Transitiva

5. Menciona si la relación "menor o igual que", definida para el conjunto de los números enteros tiene cada una de las siguientes propiedades:

a. Reflexiva

b. Simétrica

c. Transitiva

6. Sea el conjunto E = {a, b, c} y las relaciones definidas sobre E → E, menciona cuáles de las siguientes relaciones son funciones y explica la razón:

a. R1 = {(a, a), (b, b), (c, c)}

b. R2 = {(a, b), (b, b), (c, c)}

c. R3 = {(a, a), (b, b)}

d. R4 = {(a, b), (b, a), (a, c), (c, a)}

7. Sea la sucesión definida como C(n) = 2n+1, $1 \leq n \leq 6$:

a. Obtén el elemento cuyo índice es 3

b. Obtén el elemento cuyo índice es 0

c. Obtén el elemento cuyo índice es 5

d. Obtén el elemento cuyo índice es 9

8. La famosa serie de Gauss se define como $\sum_{i=1}^{n} i$. Obtén la expresión para el resultado de la suma.

9. Dadas las series, indica qué tipo de series son, explicando tu razonamiento.

a. $\sum_{i=0}^{7} \frac{1}{2^i}$

b. $\sum_{i=0}^{10} 2^i$

c. $\sum_{i=0}^{14} i^2$

10. Encuentra la notación asintótica (**Big O**) para cada uno de los siguiente algoritmos, asumiendo un O(1) para la instrucción:

```
a. for (int i=2; i<n; i++){
     instruccion;
   }

b. int x=n;
   while (x > 0){
     instruccion;
     x--;
   }

c. for (int i=1; i<=n; i+=2){
     instrucción;
   }

d. int x=n;
   while (x > 0){
     instrucción;
     x-=3;
   }
```

```
e. for (int i=2; i<n; i*=2){
     instrucción;
   }

f. int x=n;
   while (x > 0){
     instrucción;
     x/=3;
   }

g. for (int i=1; i<=n; i+=2){
     for (int j=n; j>0; j-=2){
       instrucción;
     }
   }

h. int x=n;
   while (x > 0){
     for (int y=n; y>=0; y/=2){
       instrucción;
     }
     x-=3;
   }

i. for (int i=1; i<=n; i*=2){
     for (int j=n; j>0; j-=2){
       instrucción;
     }
   }

j. for (int i=1; i<=n; i*=2){
     for (int j=n; j>0; j/=2){
       instrucción;
     }
   }

k. int algo(int n){
     if (n<=0)
       return 123;
     else
       return 1+algo(n-2);
   }

l. int algo(int n){
     if (n<=0)
       return 123;
     else
       return 1+algo(n/2);
   }
```

```
m. int algo(int n){
     if (n<=0)
       return 123;
     else
       return 1+algo(n-2)+algo(n-2)+algo(n-2)+algo(n-2);
   }

n. int algo(int n){
     if (n<=0)
       return 123;
     else
       return 1+algo(n/2)+algo(n/2)+algo(n/2)+algo(n/2);
   }
```

11. Encuentra la fórmula recursiva que representa el tiempo de ejecución de los siguientes algoritmos recursivos:

```
a. int algo(int n){
     if (n<=0)
       return 123;
     else
       return 1+algo(n-2)+algo(n-2)+algo(n-2)+algo(n-2);
   }

b. int algo(int n){
     if (n<=0)
       return 123;
     else
       return 1+algo(n-5)+algo(n-5);
   }

c. int algo(int n){
     if (n<=0)
       return 123;
     else
       return 1+algo(n/2)+algo(n/2)+algo(n/2)+algo(n/2);
   }

d. int algo(int n){
     if (n<=0)
       return 123;
     else
       return 1+algo(n/5)+algo(n/5)+algo(n/5)
   }
```

12. Encuentra la fórmula cerrada de las siguientes fórmulas recursivas:

a. $T(n) = \begin{cases} 1, & n = 0 \\ 1 + 5T(n-1), & n > 0 \end{cases}$

b. $T(n) = \begin{cases} 1, & n = 0 \\ 1 + 8T(n-1), & n > 0 \end{cases}$

c. $T(n) = \begin{cases} 1, & n = 0 \\ 1 + 4T(n/2), & n > 0 \end{cases}$

d. $T(n) = \begin{cases} 1, & n = 0 \\ 1 + 5T(n/3), & n > 0 \end{cases}$

e. $T(n) = \begin{cases} 1, & n = 0 \\ n + T(n/2), & n > 0 \end{cases}$

Capítulo 2. Estructuras de datos.

2.1 Introducción

Existen muchas estructuras de datos que son especiales para cierto tipo de problemas, algunas de las más importantes que utilizaremos con los algoritmos que trabajaremos en el presente libro serán repasadas en el presente capítulo.

2.1.1 Vectores

Una estructura muy utilizada en c++ son los vectores, que son como los arreglos, estos pueden ir creciendo en forma dinámica. Para el uso de los vectores se requiere incluir la librería de vector:

```
#include <vector>
```

Una vez incluida podremos utilizar diferentes variables invocando a sus constructores, algunos ejemplos son:

- vector<int> datos1; // Inicializa el vector datos1 de int sin nada almacenado.
- vector<int> datos2(100); // Inicializa el vector datos2 de int con 100 casillas.
- vector<int> datos3(100, -5): // Inicializa el vector datos 3 de int con 100 casillas de -5.

Algunos de los principales métodos de los vectores cuya complejidad es constante **O(1)** son:

- push_back(T); // Mete un dato al final del vector.
- size(); // Regresa las cantidad de casillas del vector.
- begin(); // Regresa un iterador del inicio del vector.
- end(); // Regresa un iterador al final del vector.
- operator[i] // Regresa el contenido de la casilla i del vector.

Ejemplo de uso de vectores en c++

```
1 vector<int> vec;
2 for (int i=1; i<=10; i++){
3    vec.push_back(i*10);
4 }
5 for (auto it=vec.begin(); it!=vec.end(); it++){
6     cout << *it << endl;
7 }
8 cout << endl;

  Salida: 10 20 30 40 50 60 70 80 90 100
```

A un vector se le puede aplicar directamente el *sort* de c++, ya que muchos compiladores requieren que se incluya la librería de *algorithm* para poder procesar el *sort* pasando como parámetros el inicio y fin de la parte del vector que se desea ordenar, por default ordenaría en forma ascendente.

Ejemplo de uso de *sort* con ordenamiento ascendente en c++

```
1 vector<int> vec{12, -5, 1, 21, 15, -3, 33, 4};
2 sort(vec.begin(), vec.end());
3 for (auto it=vec.begin(); it!=vec.end(); it++){
4     cout << *it << " ";
5 }
6 cout << endl;

  Salida: -5 -3 1 4 12 15 21 33
```

En caso de que se desee ordenar en forma descendente, la función greater() se pasa como tercer parámetro

Ejemplo de uso de *sort* con ordenamiento descendente en c++

```
1 vector<int> vec{12, -5, 1, 21, 15, -3, 33, 4};
2 sort(vec.begin(), vec.end(),greater<int>());
3 for (auto it=vec.begin(); it!=vec.end(); it++){
4     cout << *it << " ";
5 }
6 cout << endl;

  Salida: 33 21 15 12 4 1 -3 -5
```

En caso de que se desee acomodar con orden en particular dado un struct, se tiene que tener una función comparadora para que le diga cómo ordenar y ubicarla como tercer parámetro.

Ejemplo de uso de *sort* con ordenamiento particular

```
struct empleado {
    int nomina, edad;
    string nombre;
};

bool compare(const empleado &e1, const empleado &e2){
    if (e1.edad == e2.edad){
        return (e1.nomina < e2.nomina);
    }
    return e1.edad > e2.edad;
}

int main(){
    vector<empleado> vec { { 123, 23, "Juan"}, { 829, 30, "Pedro" }, { 226, 30, "Luis" }, { 550, 25, "Angel" } };
    sort(vec.begin(), vec.end(), compare);
    for (auto it=vec.begin(); it!=vec.end(); it++){
        cout << "[" << (*it).nomina << ", " << (*it).nombre << ", " << (*it).edad << "] " << endl;
    }
    return 0;
}
```

```
Salida: [226, Luis, 30]
        [829, Pedro, 30]
        [550, Angel, 25]
        [123, Juan, 23]
```

2.1.2 Filas (queue)

La fila o *queue*, por su nombre en inglés, es una estructura que controla los datos con la filosofía de que el primero que entra es el primero que sale. Se tiene un frente y un final, los datos entran por el final y salen por el frente. La Figura 2.1 muestra un ejemplo de cómo trabaja la fila (*queue*).

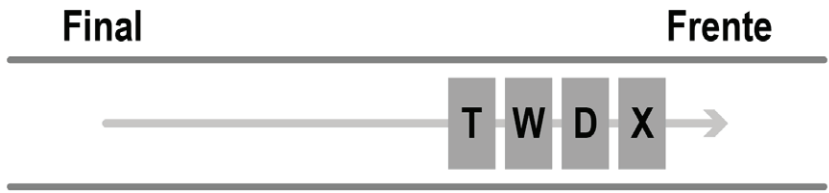


Figura 2.1 Ejemplo gráfico de una fila (*queue*)

Para poder usar las filas en c++ se requiere incluir la librería *queue*.

#include <queue>

Una vez incluida podremos utilizar en las instancias necesarias y utilizar los principales métodos de las filas (*queue*):

- push(T) // Mete el dato al final de la fila, **O(1)**.
- pop() // Saca el dato del frente de la fila, **O(1)**.
- front() // Indica que valor está al frente de la fila, **O(1)**.
- empty() // Valor booleano que representa si la fila está vacía o no, **O(1)**.
- size() // Regresa la cantidad de elementos de la fila, **O(1)**.

Ejemplo de uso de *queue* en c++

```
queue<char> q;
q.push('X');
q.push('D');
q.push('W');
q.push('T');
while (!q.empty()){
    cout << q.front() << " ";
    q.pop();
}
cout << endl;

  Salida: X D W T
```

2.1.3 Pilas (stack)

La pila o *stack*, por su nombre en inglés, es una estructura que controla los datos con la filosofía de que el último que entra es el primero que sale. Se tiene un tope por donde los datos entran y salen. La Figura 2.2 muestra un ejemplo de cómo trabaja la pila (*stack*).

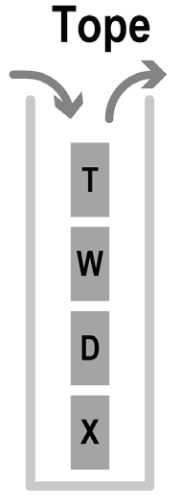


Figura 2.2 Ejemplo gráfico de una pila *(stack)*

Para poder usar las pilas en c++ se requiere incluir la librería *stack*.

```
#include <stack>
```

Una vez incluida podremos utilizar en las instancias necesarias y utilizar los principales métodos de las filas (*stack*):

- push(T) // Mete el dato al tope de la pila, **O(1)**.

- pop() // Saca el dato del tope de la pila, **O(1)**.
- top() // Indica que valor está en el tope de la pila, **O(1)**.
- empty() // Valor booleano que representa si la pila está vacía o no, **O(1)**.
- size() // Regresa la cantidad de elementos de la pila, **O(1)**.

Ejemplo de uso de *stack* en c++

```
 1 stack<char> p;
 2 p.push('X');
 3 p.push('D');
 4 p.push('W');
 5 p.push('T');
 6 while (!p.empty()){
 7     cout << p.top() << " ";
 8     p.pop();
 9 }
10 cout << endl;

   Salida: T W D X
```

2.1.4 Filas priorizadas (priority_queue)

Una fila priorizada, es una fila que está ordenada por su prioridad, donde el elemento con mayor prioridad se encontrará al frente y el de menor prioridad se encontrará al final de la fila. Cada vez que ingrese un elemento se insertará en donde el orden le corresponda. La figura 2.3 muestra un ejemplo de cómo trabaja la fila priorizada con prioridad valor mayor (*priority_queue*).

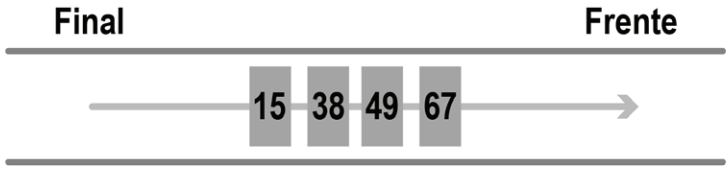


Figura 2.3 Ejemplo gráfico de una fila priorizada con prioridad valor mayor (priority_queue)

Para poder usar las filas priorizada en c++ se requiere incluir la librería *queue*.

#include <queue>

Una vez incluida podremos utilizar en las instancias necesarias y utilizar los principales métodos de las filas priorizadas (priority_queue):

- push(T) // Mete el dato a la fila priorizada, **O(log n)**.
- pop() // Saca el dato del frente (de mayor prioridad) de la fila, **O(log n)**.
- top() // Indica que valor está al frente (de mayor prioridad) de la fila, **O(1)**.
- empty() // Valor booleano que representa si la fila está vacía o no, **O(1)**.
- size() // Regresa la cantidad de elementos de la fila, **O(1)**.

Ejemplo de uso de *priority_queue* con prioridad valor mayor en c++

```
priority_queue<int> pq;
pq.push(38);
pq.push(15);
pq.push(67);
pq.push(49);
pq.push(20);
pq.push(52);
while (!pq.empty()){
    cout << pq.top() << " ";
        pq.pop();
}
cout << endl;

  Salida: 67 52 49 38 20 15
```

Si se desea tener una prioridad de valor menor se tiene que modificar el constructor para agregar la función greater y quedaría como sigue:

priority_queue<int, vector<int>, greater<int>> pq;

Ejemplo de uso de *priority_queue* con prioridad valor menor en c++

```
priority_queue<int, vector<int>, greater<int>> pq;
pq.push(38);
pq.push(15);
pq.push(67);
pq.push(49);
pq.push(20);
pq.push(52);
while (!pq.empty()){
    cout << pq.top() << " ";
        pq.pop();
}
cout << endl;

  Salida: 15 20 38 49 52 67
```

Si se desea tener una prioridad en particular se tiene que sobrecargar el operador de < dentro del *struct* o lo que se desee almacenar en la fila priorizada.

Ejemplo de uso de *priority_queue* con prioridad en particular en c++

```
struct empleado {
    int nomina, edad;
    string nombre;
    bool operator<(const empleado& e) const{
        if (edad == e.edad){
            return (nomina > e.nomina);
        }
        return edad < e.edad;
    }
};
int main(){
    priority_queue<empleado> pq;
    pq.push({ 123, 23, "Juan"});
    pq.push({ 829, 30, "Pedro" });
    pq.push({ 226, 30, "Luis" });
    pq.push({ 550, 25, "Angel" });
    while (!pq.empty()){
        cout << "[" << pq.top().nomina << ", " << pq.top().nombre <<
  ", " << pq.top().edad << "] " << endl;
        pq.pop();
    }
}

  Salida:  [226, Luis, 30]
           [829, Pedro, 30]
           [550, Angel, 25]
           [123, Juan, 23]
```

En este ejemplo tiene mayor prioridad mayor edad y en caso de ser la misma edad tomaría como prioridad el menor número de nómina.

2.1.5 Conjuntos (unordered_set)

Los conjuntos son estructuras de datos contenedores en donde se almacenan llaves únicas y se aprovechan para obtener

un acceso rápido para la búsqueda de datos. En c++ existen el *set* y los unordered_set, la diferencia básicamente es que, aunque no está estipulada su estructura interna, el *set* dado su tiempo de búsqueda al ser O(log n) nos dice que está implementado en árbol balanceado; y cuando el unordered_set es implementado en las versiones más recientes de c++ el tiempo promedio de búsqueda es O(1), por lo que nos dice que muy probablemente está implementado en una hash_table. Por estos motivos recomendamos usar unordered_set cuando se desee utilizar una estructura de datos de conjunto.

Para poder usar el unordered_set en c++ se requiere incluir la librería queque.

#include < unordered_set >

Una vez incluida podremos utilizar en las instancias necesarias y utilizar los principales métodos delunordered_set:

- insert(T) // Inserta un elemento al conjunto, si es que no existe, **O(1)**.
- find(T) // Busca un elemento en el conjunto, **O(1)**.
- empty() // Valor booleano que representa si el conjunto está vacío o no, **O(1)**.
- size() // Regresa la cantidad de elementos del conjunto, **O(1)**.

Ejemplo de uso de unordered_set en c++

```
unordered_set<string> miSet;
miSet.insert("Naranjas");
miSet.insert("Peras");
miSet.insert("Manzanas");
if (miSet.find("Peras") != miSet.end()){
    cout << "Si hay peras."<<endl;
}
else{
    cout << "No hay peras."<<endl;
}
if (miSet.find("Uvas") != miSet.end()){
    cout << "Si hay uvas."<<endl;
}
else{
    cout << "No hay uvas."<<endl;
}
for (auto it=miSet.begin(); it!=miSet.end(); it++){
    cout << "[" << *it << "]" << endl;
}
```

```
Salida: Si hay peras.
        No hay uvas.
        [Manzanas]
        [Peras]
        [Naranjas]
```

2.1.6 Mapas (unordered_map)

El unordered_map es un contenedor que almacena elementos formados por una combinación de una llave y un valor mapeado a ella. La llave no se puede repetir y se puede acceder directamente mediante el operador [].

Para poder usar el unordered_map en c++ se requiere incluir la librería unordered_map.

#include <unordered_map>

Una vez incluida podremos utilizar en las instancias necesarias y utilizar los principales métodos son:

- operator[] // Si la llave existe regresa el valor, sino inserta esa <llave, valor> **O(1)**.
- empty() // Valor booleano que representa si la fila está vacía o no, **O(1)**.
- size() // Regresa la cantidad de elementos de la fila, **O(1)**.

Ejemplo de uso de unordered_map en c++

```
unordered_map<string, int> miMap;
miMap["Luis"] = 30;
miMap["Angel"] = 23;
miMap["Pedro"] = 25;
miMap["Juan"] = 31;
miMap["Pablo"] = 28;
for (auto it=miMap.begin(); it!=miMap.end(); it++){
    cout << it->first << " tiene " << it->second << " años." <<endl;
}
cout << "----------" << endl;
unordered_map<string, int>::iterator it;
it = miMap.find("Paco");
if (it != miMap.end()){
    cout << "La edad de Paco es: "<< it->second << endl;
}
else{
    cout << "Paco no esta en el mapa." << endl;
}
it = miMap.find("Pedro");
if (it != miMap.end()){
    cout << "La edad de Pedro es: "<< it->second << endl;
}
else{
    cout << "Pedro no esta en el mapa." << endl;
}

  Salida: Pablo tiene 28 años.
          Juan tiene 31 años.
          Pedro tiene 25 años.
          Angel tiene 23 años.
          Luis tiene 30 años.
          ----------
          Paco no esta en el mapa.
          La edad de Pedro es: 25

          [Peras]
          [Naranjas]
```

2.2 Grafos

En las jerarquías de la organización de estructuras de datos, los grafos son el caso más general que existe. Son una estructura de tipo RED donde se mantiene una relación de muchos a muchos (N:M) entre sus elementos. Una red de carreteras entre las diferentes ciudades del país es un ejemplo de un grafo. La Figura 2.4 nos muestra un ejemplo gráfico de un grafo.

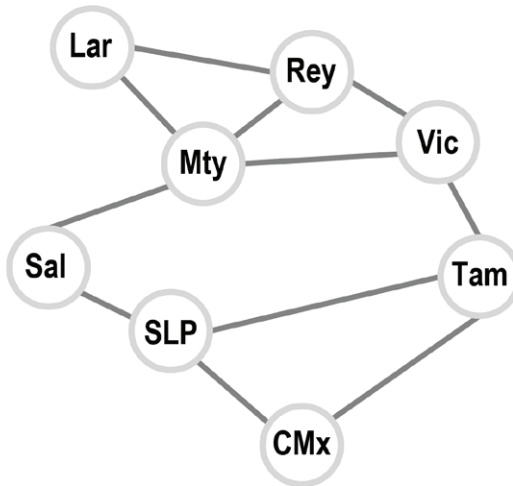


Figura 2.4 Ejemplo gráfico de un grafo

2.2.1 Terminología de grafos

Un grafo se puede describir además como un conjunto de nodos y arcos.

Nodo

Los nodos, también llamados vértices (*vertex* en inglés), son los elementos básicos de información de un grafo.

Arco

Los arcos, también llamados aristas (*edges* en inglés), son las conexiones entre dos nodos, estableciendo la relación entre dos elementos del grafo.

Dentro de la Figura 2.4, el conjunto de nodos es:

$$V=\{Lar,Rey,Mty,Sal,Vic,SLP,Tam,CMx\}$$

Y el conjunto de arcos es:

$$E = \begin{Bmatrix} (Lar,Rey),(Lar,Mty),(Rey,Mty),(Mty,Sal),(Mty,Vic),(Vic,Tam), \\ (Sal,SLP),(Tam,SLP),(Tam,CMx),(SLP,CMx) \end{Bmatrix}$$

Subgrafo

Un subgrafo de un grafo es un grafo, cuyo conjunto de nodos es un subconjunto de los nodos del grafo y el conjunto de arcos es un subconjunto de los arcos del grafo .

Dentro de la Figura 2.4, un subgrafo pudiera ser los conjuntos V' y E':

$$V' = \{Lar, Rey, Mty\}$$
$$E' = \{(Lar, Rey), (Lar, Mty), (Rey, Mty)\}$$

La Figura 2.5 muestra gráficamente los conceptos de nodo, arco y subgrafo.

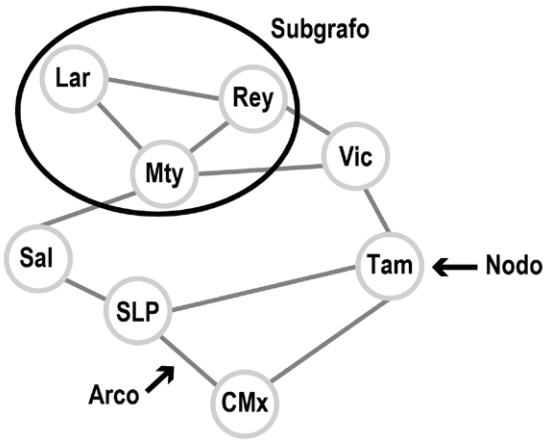


Figura 2.5 Ejemplo gráfico de nodo, arco y subgrafo

Nodos adyacentes

Cuando se tiene una conexión entre dos nodos **A** y **B** se dice que **A** y **B** son adyacentes.

Vecinos de un nodo

Al conjunto de nodos que son adyacentes a un nodo **A** se dice que estos son vecinos del nodo **A**. En la Figura 2.5 se puede apreciar, por ejemplo, que **SLP** y **CMx** son nodos adyacentes, además que los vecinos de **Vic** son: *{Rey,Mty,Tam}*.

Camino

Un camino o trayectoria es una secuencia de nodos para ir de un nodo **A** un nodo **B**, donde para cada par de nodos en secuencia deben de ser adyacentes.

Trayectoria simple

Una trayectoria simple es aquel camino para ir del nodo **A** al nodo **B**, en donde todos los nodos contenidos en el camino son distintos.

En la figura 2.5, un camino simple para ir de **Rey** a **SLP** es: *Rey → Vic → Tam → CMx → SLP.*

Grafo no dirigido

Un grafo no dirigido es aquel donde los arcos no tienen una dirección, esto es que si se tiene un arco *(A, B)* este puede ir tanto de **A** a **B** como de **B** a **A**. La Figura 2.6 muestra un ejemplo de grafo no dirigido.

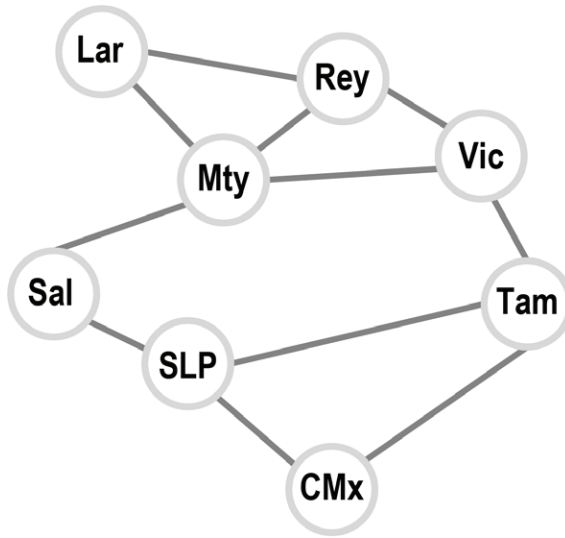


Figura 2.6 Grafo no dirigido

Grafo dirigido

Un grafo dirigido es aquel donde los arcos tienen una dirección, esto es que si se tiene un arco *(A, B)* este solo puede ir a **A** a **B** y no puede ir de **B** a **A**. La Figura 2.7 muestra un ejemplo de grafo dirigido.

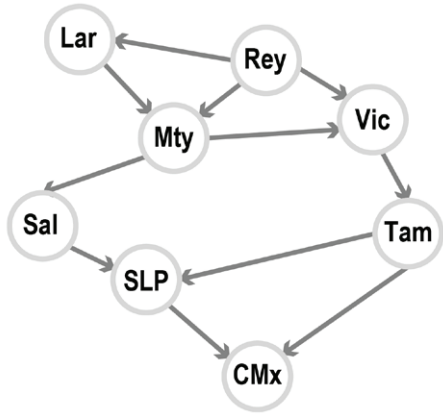


Figura 2.7 Grafo dirigido

Grafo ponderado

Cuando los arcos tienen un valor asociado se dice que el grafo es un grafo ponderado, típicamente este valor representa el valor del problema que se esté solucionando como: costo, distancia, kilómetros, gasolina, entre otros. La Figura 2.8 muestra un grafo no dirigido y ponderado.

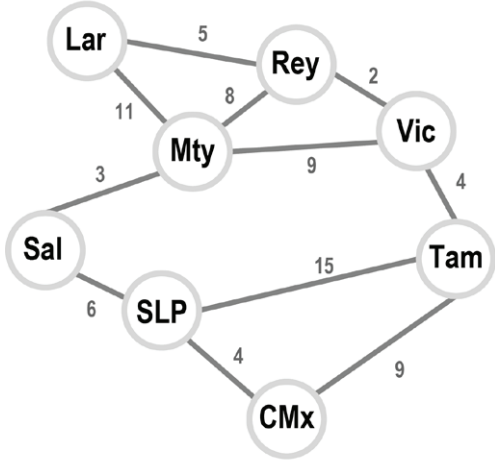


Figura 2.8 Grafo no dirigido y ponderado

Ciclo

Un ciclo es una trayectoria donde el nodo de inicio y el de terminación son el mismo nodo. En la Figura 2.8 un ciclo puede ser: *Rey* → *Vic* → *Mty* → *Rey*.

2.2.2 Representación de un grafo

Un ADT Grafo debe contener al menos las siguientes operaciones:

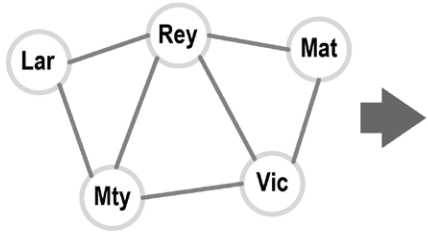
- Insertar un nodo
- Insertar un arco
- Borrar un nodo
- Borrar un arco
- Buscar un nodo
- Recorrer el grafo

Y para poder soportar estas operaciones se tiene que pensar en cómo poder representar un grafo en el lenguaje de programación; existen muchas formas de representar un grafo, pero las más comunes son

- Matriz de adyacencia
- Lista de adyacencia
- Lista de arcos

2.2.2.1 Matriz de adyacencia

La representación de un grafo en una matriz de adyacencia tendría una matriz de **N x N** donde **N** es la cantidad de Nodos y cada celda de la matriz representaría si hay o no adyacencia entre el nodo del renglón con el nodo de la columna. En la Figura 2.9 se puede observar un grafo no dirigido y no ponderado de 5 nodos y su representación en una matriz de adyacencia.

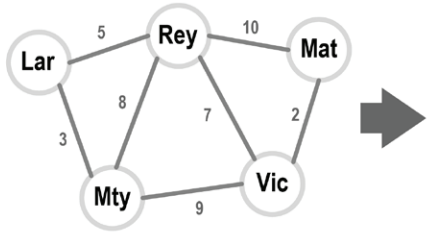


	Lar	Mty	Rey	Mat	Vic
Lar	F	T	T	F	F
Mty	T	F	T	F	T
Rey	T	T	F	T	T
Mat	F	F	T	F	T
Vic	F	T	T	T	F

Figura 2.9 Grafo no dirigido y no ponderado en matriz de adyacencias

En caso de que el grafo fuera ponderado, en lugar de una matriz de booleanos tendríamos una matriz de enteros donde se pondría la ponderación de ir saliendo del nodo del renglón al nodo de la columna, cuando no exista conexión tradicionalmente se pone infinito y en la diagonal principal se pone 0. Cuando el grafo es no dirigido, el triángulo superior de la matriz es un espejo del triángulo. En la Figura 2.10 se puede observar un grafo no dirigido y ponderado de 5 nodos y su representación en una matriz de adyacencia.

La ventaja de representar un grafo en una matriz de adyacencia es que las operaciones de manipulación de arcos son muy fáciles, pero se requiere saber el número exacto de nodos en un inicio y este no podrá ser modificado tan fácilmente.



	Lar	Mty	Rey	Mat	Vic
Lar	0	3	5	∞	∞
Mty	3	0	8	∞	9
Rey	5	8	0	10	7
Mat	∞	∞	10	0	2
Vic	∞	9	7	2	0

Figura 2.10 Grafo no dirigido y ponderado en matriz de adyacencias

2.2.2.2 Lista de adyacencias

La representación de un grafo en una lista de adyacencia sería tener, por lado, una lista de **N** nodos y, por cada nodo, una lista de los nodos con los que tiene adyacencia y su ponderación en caso de que sea un grafo ponderado. En la Figura 2.11 se puede observar un grafo no dirigido y ponderado de 5 nodos y su representación en una lista de adyacencia.

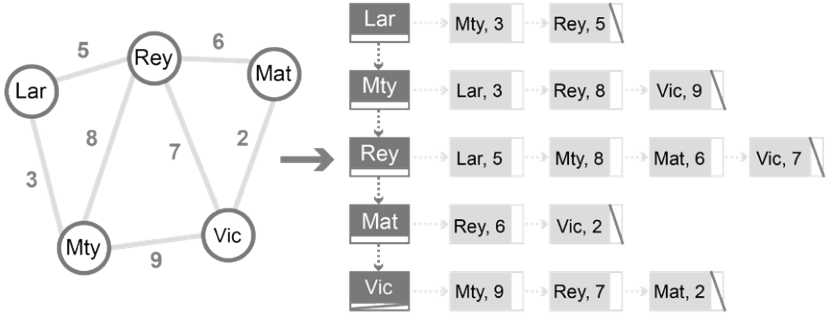


Figura 2.11 Grafo no dirigido y ponderado en la lista de adyacencias

La ventaja de representar un grafo en una lista de adyacencia es no se requiere conocer con anterioridad la cantidad de nodos y arcos que conforman el grafo, pero requiere más espacio de memoria debido al manejo de apuntadores.

2.2.2.3 Lista de arcos

La representación de un grafo en una lista de arcos es la más compleja de implementar, se tiene una lista de los nodos y cada nodo tiene dos apuntadores: uno a una lista de arcos donde ese nodo es el origen del arco y otro apuntador a una lista arcos donde ese nodo es el destino del arco. La Figura 2.12 se muestra un grafo no dirigido y no ponderado de 5 nodos y 7 arcos en una representación de lista de arcos.

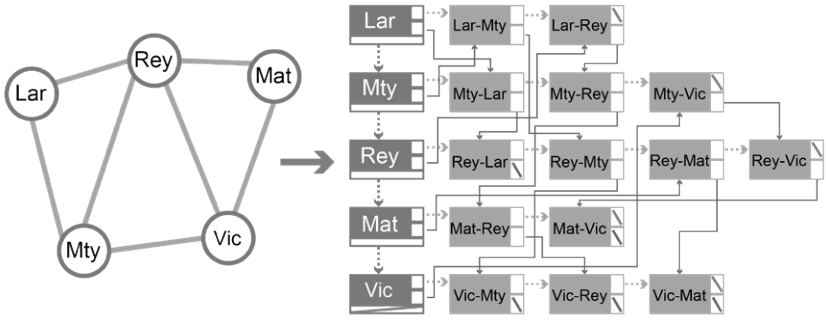


Figura 2.12 Grafo no dirigido y ponderado en la lista de arcos

La ventaja de representar un grafo en una lista de arcos es que se obtiene una representación bastante eficiente, pero requiere más espacio de memoria debido al manejo de apuntadores.

2.2.3 Recorridos de un grafo

Un recorrido sobre una estructura de datos consiste en visitar a cada uno de los nodos una sola vez; en el caso de los grafos no existe un nodo inicial, por lo que se tiene que indicar el nodo sobre el cual iniciará el recorrido. Los recorridos más utilizados en los Grafos son:

- BFS – *Breadth First Search* (primero en anchura).

- DFS – *Depth First Search* (primero en profundidad).

Los dos recorridos son métodos sistemáticos que sirven para visitar todos los nodos y arcos de un grafo exactamente una vez, por lo cual podríamos decir que estos algoritmos nos permiten realizar recorridos controlados sobre el grafo.

Una de la operaciones más sencillas y elementales de cualquier estructura de datos es la búsqueda, por lo que se ha estandarizado el uso de estos dos recorridos para ello y, por eso, se les conoce como algoritmos de búsqueda, pero son utilizados para muchas otras operaciones sobre los grafos que no necesariamente realicen la búsqueda de un elemento dentro del grafo.

2.2.3.1 BFS – Breadth First Search (primero en anchura)

La idea fundamental de este recorrido es que, a partir de visitar el nodo de inicio se visitan a todos sus vecinos, después a los vecinos de estos vecinos simulando un recorrido nivel por nivel si habláramos de un árbol binario de búsqueda. El orden en que se vistan a los vecinos es normalmente el orden en que fueron almacenados los datos en la representación que se esté utilizando.

Este algoritmo utiliza una fila (*queue*) auxiliar que para evitar ciclos infinitos requiere un almacenamiento de estatus para cada nodo, los cuales arrancan en "Espera" y una vez que son procesados se cambia a "Procesado".

El algoritmo general para **BFS** es:

Algoritmo 2.1 BFS

Entrada: recibe el grafo en alguna de sus representaciones.

Salida: regresa un vector C donde coloca el recorrido BFS.

1. Inicializar el estatus de todos los nodos a "*En Espera*".

2. Para cada nodo del grafo:

    2.1 ¿El estatus del nodo es "En Espera?

        2.1.1 Sí,

            2.1.1.1 Insertar el Nodo en la FILA cambiándole su estado a "Procesado"

            2.1.1.2 Meter en el vector C el nodo.

        2.1.2 Mientras la FILA no esté vacía

            2.1.2.1 Sacar Nodo del frente la FILA y procesarlo.

            2.1.2.2 Meter a la FILA todos los vecinos del nodo que tengan estatus "En Espera", cambiándolo a "Procesado" y meter el Nodo al vector C.

La implementación del BFS utilizando una representación del grafo en matriz de adyacencias sería:

```
/* matAdj =  una matriz de bool que
             representa la matriz de adjacencia
   N      =  cantidad de Nodos.
*/
void BFS(bool matAdj[N][N]){
/* status es un vector de bool
    false = "En Espera"
    true = "Procesado"
*/
    vector<bool> status(N,false)
    queue<int> proceso;
    int data;
    for (int i=0; i<N; i++){
        if (!status[i]){
           fila.push(i);
           status[i] = true;
           while (!fila.empty()){
               data = fila.front();
               fila.pop();
               cout << (data) << " ";
               for (int j=0; j<v; j++){
                   if (matAdj[data][j] && !status[j]){
                      fila.push(j);
                      status[j] = true;
                   }
               }
           }
        }
    }
}
```

La figura 2.13 muestra el recorrido **BFS** que tendría un grafo no dirigido y no ponderado de 8 nodos.

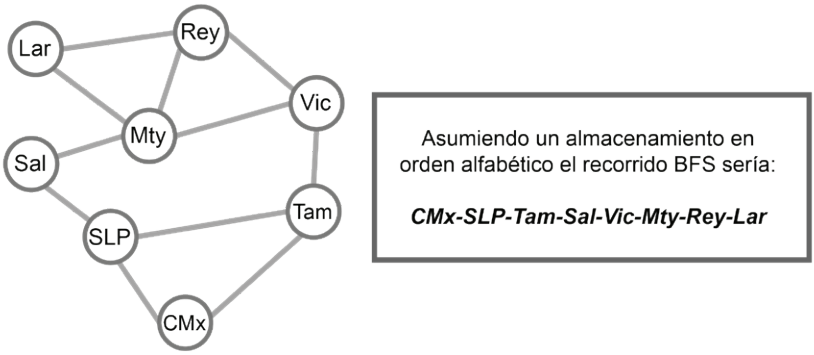


Figura 2.13 Recorrido BFS en un grafo no dirigido y no ponderado

2.2.3.2 DFS – Depth First Search (primero en profundidad)

Este recorrido se basa primordialmente en que, a partir de visitar el nodo de inicio se visita al primer vecino de él, luego al primer vecino del primer vecino y así recursivamente a todos sus vecinos no procesados. El orden en que se visita a los vecinos es normalmente el orden en que fueron almacenados los datos en la representación que se esté utilizando.

Este algoritmo utiliza una pila (*stack*) auxiliar para simular la recursividad, que para evitar ciclos infinitos requiere un almacenamiento de estatus para cada nodo, los cuales arrancan en "Espera" y una vez que son procesados se cambia a "Procesado".

El algoritmo general para **DFS** es:

Algoritmo 2.2 DFS

Entrada: recibe el grafo en alguna de sus representaciones.

Salida: regresa un vector C donde coloca el recorrido DFS.

1. Inicializar el estatus de todos los nodos a "*En Espera*".

2. Para cada nodo del grafo:

2.1 ¿El estatus del nodo es "*En Espera*?

2.1.1 Sí,

2.1.1.1 Insertar el Nodo en la PILA

2.1.1.2 Mientras la PILA no esté vacía.

2.1.1.2.1 Sacar nodo del tope la PILA, en caso de que su estado sea "*En Espera*" entonces procesarlo y cambiarle su estado a "*Procesado*".

2.1.1.2.2 Meter a la PILA todos los vecinos del nodo que tengan estatus "*En Espera*", cambiándolo a "*Procesado*" y meter el nodo al vector C.

La implementación del **DFS** y utilizando una representación del grafo en lista de adyacencias sería:

```
/* listAdj = es un vector de vector de enteros que
             representa la lista de adjacencia
*/
void DFS(vector<vector<int> > &listAdj){
    int v = listAdj.size();
    stack<int> pila;
    int data;
/* status es un vector de bool
       false = "En Espera"
       true = "Procesado"
*/
    vector<bool> status(v, false);
    for (int i=0; i<N; i++){
        if (!status[i]){
            pila.push(i);
            while (!pila.empty()){
                data = pila.top();
                pila.pop();
                if (!status[data]){
                    cout << (data)<< " ";
                    status[data] = true;
                    for (int i=listAdj[data].size()-1; i>=0; i--){
                        if (!status[listAdj[data][i]]){
                           pila.push(listAdj[data][i]);
                        }
                    }
                }
            }
        }
    }
    cout << endl;
}
```

La Figura 2.14 muestra el recorrido **DFS** que tendría un grafo no dirigido y no ponderado de 8 nodos

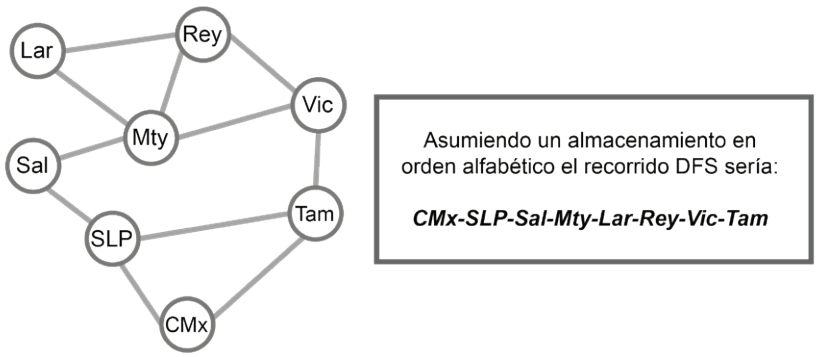


Figura 2.14 Recorrido DFS en un grafo no dirigido y no ponderado

2.3 Conjunto disjunto (Disjoint-set)

La estructura de datos de conjunto disjunto o por su nombre en inglés Disjoint-set, también conocida como estructura de datos unión-*find*, es una estructura de datos que almacena una colección de conjuntos disjuntos, esto es un conjunto de pequeños conjuntos independientes. La Figura 2.15 muestra una representación de un conjunto disjunto con 7 conjuntos disjuntos.

Figura 2.15 Ejemplo de un conjunto disjunto

Las operaciones más importantes de la estructura de datos conjunto disjunto son:

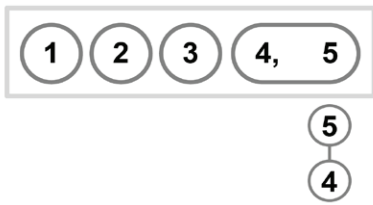
- *Add:* Agrega un nuevo conjunto a la estructura de datos.
- *Find:* Busca el conjunto al que pertenece un dato dentro de la estructura de datos.
- *Merge:* combina dos conjuntos de la estructura de datos para formar uno solo.

Primeramente, lo que se requiere es la construcción de la estructura de datos, para esto se tendrá un *struct* dentro de c++ que contenga la cantidad de conjuntos, así como dos vectores, los cuales irán simulando un árbol, uno de los vectores contendrá el padre del conjunto del índice y el otro contendrá la cantidad de niveles (*rank*) que tiene como descendientes. Las siguientes figuras van mostrando paso a paso desde la construcción de un disjoint-set de 5 conjuntos hasta formar un solo conjunto.

Dato	1	2	3	4	5
parent	1	2	3	4	5
rank	0	0	0	0	0

Figura 2.16

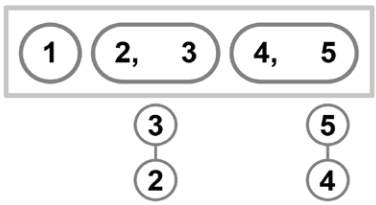
El constructor genera conjuntos independientes, en el vector parent coloca el mismo número que el conjunto y en el vector rank coloca 0 en cada casilla, ya que serían conjuntos independientes. Si se desea integrar al 4 y al 5 en un mismo conjunto quedaría de la siguiente forma:



Dato	1	2	3	4	5
parent	1	2	3	5	5
rank	0	0	0	0	1

Figura 2.17

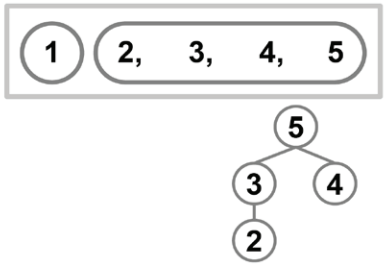
Indicando que el padre de 4 ahora es 5 y que el 5 tiene debajo de él 1 nivel. Si ahora se desea integrar al 2 y al 3 en un mismo conjunto quedaría de la siguiente forma:



Dato	1	2	3	4	5
parent	1	3	3	5	5
rank	0	0	1	0	1

Figura 2.18

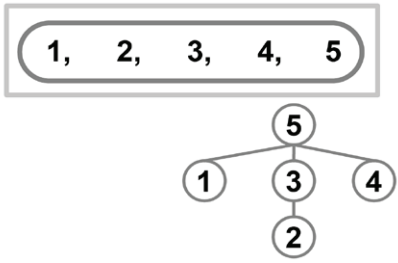
Ahora el padre de 2 es 3 y l 3 tiene 1 nivel debajo de él. Si se desea integrar al 2 con el 4 quedaría de la siguiente forma.



Dato	1	2	3	4	5
parent	1	3	5	5	5
rank	0	0	1	0	2

Figura 2.19

Esto nos indica que se unieron el 2, 3, 4 y 5 en un mismo conjunto, el padre de 2 es 3 y el padre de 4 es 5, al tener el mismo *rank* los dos padres se decide que el nuevo padre de los 4 será 5, poniendo como padre de 3 a 5 y como *rank* de 5 uno más. Si ahora se quiere integrar al 1 con el 5 quedaría de la siguiente forma:



Dato	1	2	3	4	5
parent	5	3	5	5	5
rank	0	0	1	0	2

Figura 2.20 Ejemplo paso a paso de la unión en un conjunto disjunto

Como se puede apreciar ya queda todo unido poniendo al 5 como padre de 1 y el *rank* queda como estaba anteriormente, ya que no aumentó de nivel.

La implementación para la creación de la estructura de datos de disjoint-set es:

```
 1 struct disjointSet{
 2     vector<int> parent;
 3     vector<int> rank;
 4     int n;
 5
 6     disjointSet(int n){
 7         this->n = n;
 8         for (int i = 0; i <= n; i++){
 9             parent.push_back(i);
10             rank.push_back(0);
11         }
12     }
13     void add();
14     int find(int u);
15     void merge(int x, int y);
16 };
```

En donde básicamente genera los dos vectores (*parent* y *rank*) de tamaño n+1, inicializando en *parent* el mismo índice y *rank* inicializa con ceros.

La implementación del método *add* para agregar un nuevo conjunto queda como a continuación se detalla:

```
1  void disjointSet::add(){
2      n++;
3      parent.push_back(n);
4      rank.push_back(0);
5  }
```

En donde se incrementa en uno la cantidad de elementos dentro del disjoint-set y se coloca el mismo como padre y con *rank* de 0 diciendo con esto que es un conjunto inicial de un solo elemento.

El método *find* recibe como parámetro el elemento, del cual se desea encontrar la raíz del árbol *parent*, esto significa en conjunto al que pertenece; la implementación quedaría como a continuación se describe:

```
1  int disjointSet::find(int u){
2      if (u != parent[u]){
3         parent[u] = find(parent[u]);
4      }
5      return parent[u];
6  }
```

Como se puede notar, *find* es una función recursiva que para cuando un elemento es padre de sí mismo, siendo este la raíz del árbol del conjunto de ese elemento.

El método *merge* recibe como parámetro dos elementos, de los cuales se desean unir los dos conjuntos a los que ellos pertenecen, la precondición es que ambos pertenezcan a diferentes conjuntos. La implementación quedaría como a continuación se describe:

```
 1 void disjointSet::merge(int x, int y) {
 2     x = find(x);
 3     y = find(y);
 4     if (rank[x] > rank[y]){
 5             parent[y] = x;
 6     } else{
 7         parent[x] = y;
 8     }
 9     if (rank[x] == rank[y]){
10        rank[y]++;
11     }
12 }
```

Primeramente, se invoca a la función *find* para encontrar la raíz del árbol del conjunto de cada uno de los elementos. Posteriormente se revisa el *rank* de cada uno de esos árboles y el más chico se agrega al más grande. En caso de que sean los *rank* iguales el primer conjunto pasa a agregarse al segundo conjunto.

La estructura de conjunto disjunto es muy utilizada en muchos algoritmos para solucionar problemas tanto de grafos como de otras problemáticas, en capítulos posteriores se utilizarán en algunos de los algoritmos que trabajaremos.

Capítulo 3. Técnicas de diseño de algoritmos.

La historia de la computación está compuesta por la historia del *hardware* y del *software*. A su vez, la historia del software es, en gran medida, la historia de los algoritmos.

Los algoritmos son una de las partes medulares de los sistemas computacionales. Los científicos de la computación se han enfrentado a problemas cuya solución se da por medio de un algoritmo y la aparición de estos ha sido fundamental para el desarrollo de la computación, sobre todo cuando logran solucionar problemas que se presentan a menudo en los sistemas.

Las características que se pueden estudiar en los algoritmos son muy variadas, pero es muy importante que un buen algoritmo posea dos en específico:

- Que logre solucionar correctamente el problema en todas sus instancias.
- Que la solución sea eficiente en tiempo y memoria.

¿Cómo se diseña un algoritmo? Hasta el momento no existe una sola técnica de diseño de un algoritmo. A lo largo de la corta historia de los algoritmos computacionales, los científicos han desarrollado y utilizando diferentes enfoques para su diseño, sin embargo algunos de estos han sido usados para generar una gran cantidad de algoritmos y es por esto que su estudio es la parte fundamental de este capítulo.

El tiempo nos ha mostrado que una buena forma de aprender a diseñar nuevos algoritmos o inclusive, nuevos enfoques para el diseño de algoritmos, es estudiar y entender los enfoques que se han desarrollado y que han sido muy exitosos. Debemos conocer el algoritmo mismo, sus características, las matemáticas que

están detrás de él y, desde luego, su implementación. Las cinco técnicas que abordaremos en este capítulo son: **divide y vencerás**, **algoritmos avaros** (*greedy*), **programación dinámica**, y **ramificación y poda** (*branch and bound*).

Debemos recordar que la implementación de un algoritmo siempre va acompañada de una estructura de datos y que la selección de dicha estructura afecta directamente la eficiencia del algoritmo, por lo que también será necesario recordar algunas de las estructuras de datos clásicas o aprender algunas nuevas, para seleccionar la mejor.

El estudio de los algoritmos tiene una parte fuertemente formal. Esto nos lleva a la posibilidad de poder demostrar matemáticamente si un algoritmo es el mejor de su tipo, es decir, que tiene la mayor eficiencia que se pueda lograr. Sin embargo, como todo en las matemáticas, estas demostraciones no son simples y también tienen que crearse. Se han logrado hacerse para algunos algoritmos, pero para muchos no. Desde luego, los científicos siguen trabajando en eso. Esto no es del todo malo porque siempre permitirá que los desarrolladores puedan pensar en crear mejores algoritmos a los ya existentes, desde luego de aquellos en los que no se ha demostrado que sean los mejores.

Para crear un nuevo algoritmo existen dos posibilidades:

- Crear un algoritmo que resuelva un problema que hasta el momento no tiene solución.

- Crear un algoritmo cuya eficiencia sea mejor que la de los algoritmos previos que resuelven el mismo problema.

El punto clave para los desarrolladores de algoritmos es que al terminar de diseñar su algoritmo siempre y de forma inmediata se hagan la pregunta mágica que ha permitido y que se-

guirá permitiendo el gran avance que tenemos hasta ahora: ¿se puede hacer mejor?

3.1 Divide y vencerás

La frase **divide y vencerás** es muy antigua. No se sabe quién exactamente la inventó, pero se sabe que algunos dirigentes romanos ya la usaban y la ponían en práctica. La idea es muy simple, si logras dividir a tus enemigos será más fácil vencerlos porque tienes que pelear contra grupos más pequeños.

En computación, **divide y vencerás** es una de las técnicas más utilizadas en el diseño de algoritmos. Lo sorprendente es que la idea principal es la misma. Se trata de dividir un problema en varios subproblemas más pequeños del mismo tipo, los cuales serán más fáciles de resolver por ser más simples. Una vez resueltos estos problemas, se puede encontrar la solución del problema mayor combinando de alguna manera las soluciones de los subproblemas. Además, como los subproblemas son del mismo tipo, estos a su vez se puede resolver de forma similar (recursión), hasta llegar a un tipo especial de subproblema que se resuelve en forma directa (caso base). Generalmente, este tipo de algoritmos se implementa por medio de la recursión. Los pasos de este tipo de algoritmos se pueden ver en el Algoritmo 3.1

Algoritmo 3.1 Divide y vencerás

Entrada: el problema a resolver.

Salida: el problema resuelto.

1. Pensar en una forma de representar el problema de tal forma que pueda ser dividido en k problemas más simples, pero del mismo tipo.

2. Resolver los k subproblemas en forma recursiva hasta llegar a un subproblema que se pueda resolver en forma directa.

3. Combinar las soluciones de los k subproblemas para encontrar la solución del problema mayor.

Como ya se comentó, hay muchos algoritmos que siguen exactamente este diseño. Tal es el caso de los algoritmos de ordenamiento *Merge Sort* y *Quick Sort*.

Algunos algoritmos caen en esta categoría, aunque no resuelven todos los k problemas en los que se dividen sino simplemente seleccionan uno de esos k subproblemas donde saben que se encuentra la solución del problema mayor. Como el caso del algoritmo de **búsqueda binaria** (*Binary Search*).

3.1.1 Ejemplo de funcionamiento

Como ya se comentó, *Merge Sort* es un clásico ejemplo de este tipo de algoritmos. La idea de John Von Newmann, creador de *Merge Sort* en 1945, fue que para ordenar un arreglo de números era más fácil partir ese arreglo en dos partes, cada una con la mitad de sus elementos (o lo más aproximado) y ordenar dos arreglos más pequeños, luego, combinar los dos arreglos ya ordenados para obtener el arreglo mayor ordenado. Es claro que para su diseño usó la técnica de **divide y vencerás**, por lo que los dos arreglos más pequeños se ordenan a su vez usando *Merge Sort* hasta que la división provoca arreglos que solo tienen un ele-

mento, el cual queda automáticamente ordenado. Después, toma las soluciones obtenidas y las combina para obtener la solución del problema mayor, es decir, Von Newmann ideó un método que recibe dos arreglos ya ordenados y regresa un arreglo mayor ordenado formado por los dos arreglos. El algoritmo de *Merge Sort* se puede ver en el Algoritmo 3.2.

Algoritmo 3.2 *Merge Sort*

Entrada: recibe un arreglo desordenado.

Salida: regresa el arreglo ordenado.

1. Si el arreglo recibido es de más de un elemento:

    1.1 Dividir el arreglo en dos subarreglos: arreglo izquierdo y arreglo derecho

    1.2 Ordenar el arreglo izquierdo usando *Merge Sort*.

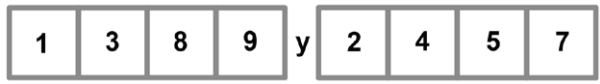
    1.3 Ordenar el arreglo derecho usando *Merge Sort*.

    1.4 Combinar los dos subarreglos ordenados para obtener el arreglo original ordenado

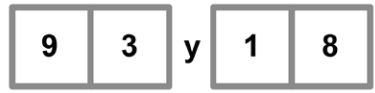
2. Regresar el arreglo.

Veamos un ejemplo de su aplicación si queremos ordenar el siguiente arreglo pequeño:

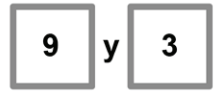
Merge Sort inicia dividiendo el arreglo en dos partes iguales:



Ahora nuestra tarea es ordenar dos arreglos de 4 elementos en lugar de uno de 8. Esta tarea debería ser más fácil. Procedemos con el arreglo de la izquierda usando el mismo *Merge Sort*, por lo que lo partimos en dos partes iguales:



Tomamos el arreglo de 2 elementos de más a la izquierda y volvemos a ordenarlo usando Merge Sort, por lo que lo dividimos en dos partes iguales:



Los dos nuevos arreglos ya no se pueden partir porque están formados sólo por un elemento, por lo que cada uno de esos arreglos está automáticamente ordenado.

Ahora, combinamos los dos últimos arreglos usando un procedimiento muy eficiente llamado *Merge* (de ahí su nombre), el cual se describe en el Algoritmo 3.3.

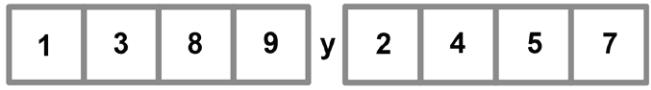
Algoritmo 3.3 *Merge* (Combina)

Entrada: recibe dos arreglos ordenados A y B.

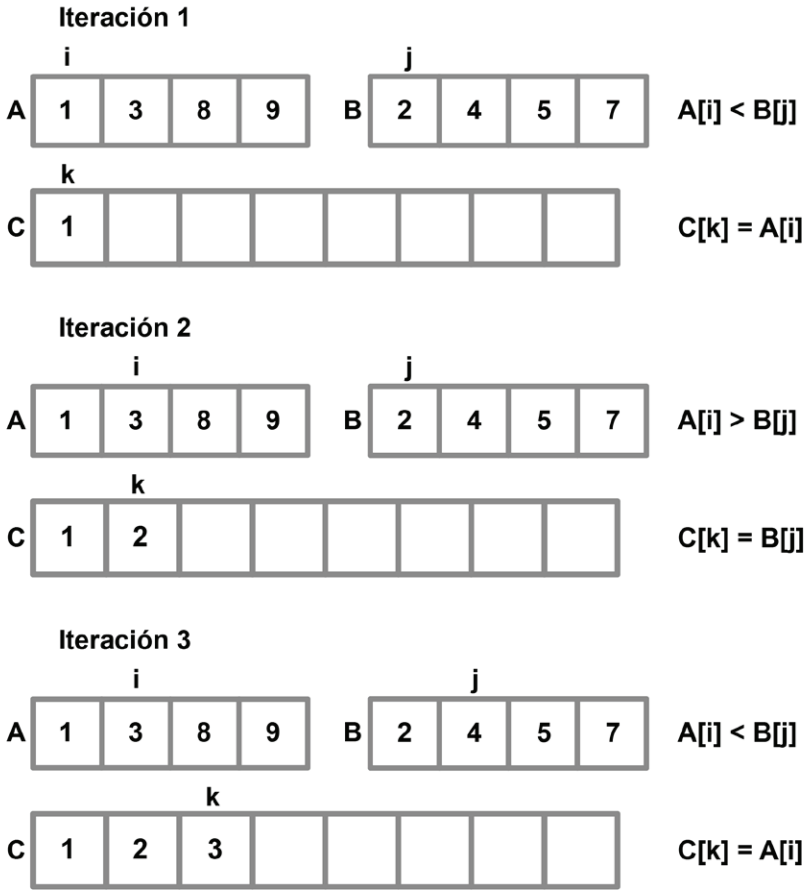
Salida: regresa un arreglo C ordenado formado por los elementos de los dos arreglos de entrada.

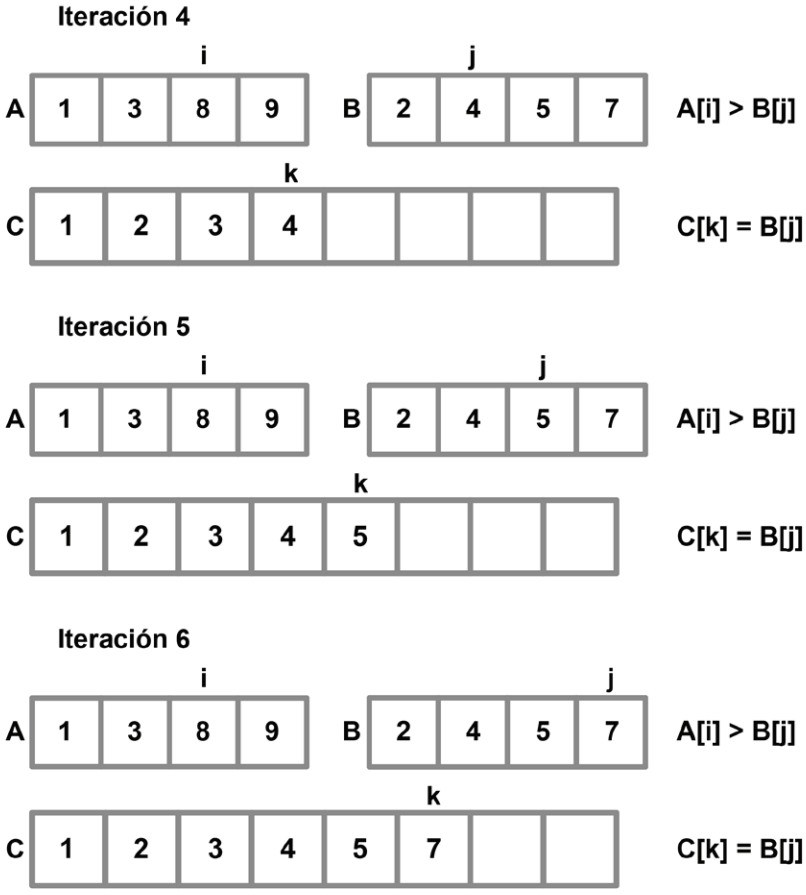
1. Hacer i=0, j=0 y k=0
2. ¿i llegó a la longitud de A?
    - 2.1 Sí,
        - 2.1.1 Copiar los elementos restantes de B en C
        - 2.1.2 Ir a 5
    - 2.2 No,
        - 2.2.1 Copiar los elementos restantes de A en C
        - 2.2.2 Ir a 5
3. ¿A[i] < B[i]?
    - 3.1 Sí,
        - 3.1.1 C[k] = A[i]
        - 3.1.2 i = i + 1
    - 3.2 No,
        - 3.2.1 C[k] = B[j]
        - 3.2.2 j = j + 1
4. k = k + 1, ir a 2
5. Regresar C

Como ejemplo del algoritmo *Merge* combinemos los dos últimos arreglos ordenados para obtener el arreglo final. El proceso recibe los arreglos:



El proceso completo se presenta en la Figura 3.1.





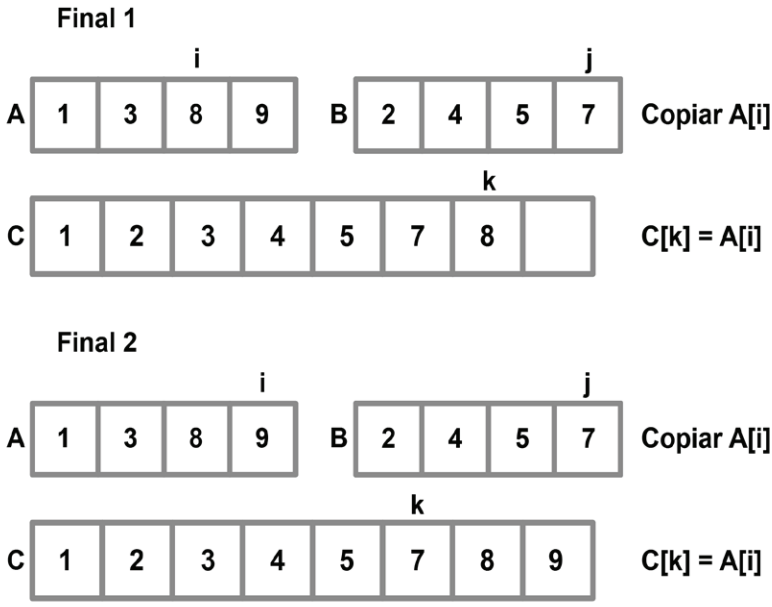


Figura 3.1 Ejemplo de proceso de combinación de arreglos (Merge) en Merge Sort

En la Iteración 1, se copia A[i] a C[k] porque A[i] es menor que B[j]. En la Iteración 2, se copia B[j] porque es el menor. Así sucesivamente hasta que en la Iteración 6 se alcanza el final del arreglo B y en los dos últimos pasos Final 1 y Final 2 simplemente se copian los elementos que faltan del arreglo A a C con lo que se completa el proceso.

Repetimos el proceso de *Merge Sort* con el resto de los arreglos, como se puede ver completo en la Figura 3.2.

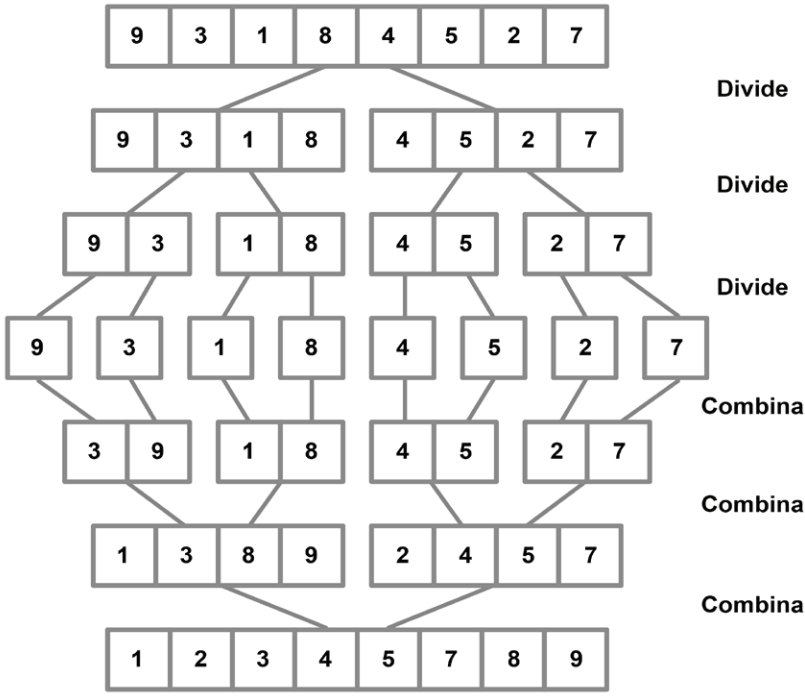


Figura 3.2 Ejemplo de Merge Sort

3.1.2 Análisis de su complejidad

Los algoritmos **divide y vencerás** son recursivos y su complejidad se analiza usando una ecuación de recurrencia de la forma:

$$T(n) = bT\left(\frac{n}{c}\right) + f(n) \qquad (3.1)$$

Donde:

- b es el número de subproblemas que se resuelven en forma recursiva

- c es el factor en el que disminuye el tamaño del problema
- T(n) es el número de operaciones realizadas en el paso recursivo
- f(n) es el número de operaciones expresadas como una función del tamaño de la entrada, que se realizan en forma no recursiva

En el caso del *Merge Sort*, como el arreglo se divide en dos arreglos que contienen la mitad de los elementos, entonces c = 2. Cada uno de estos arreglos se ordena usando *Merge Sort*, es decir, b = 2. Finalmente, como los arreglos ordenados obtenidos se combinan en un solo arreglo para que quede ordenado, su número de operaciones estará dado por el método *Merge*, el cual no es recursivo, pero sí depende del tamaño de la entrada, por lo que este número de pasos queda identificado por la función f(n).

Es fácil observar que, en el peor de los casos, el número máximo de comparaciones que hace el proceso *Merge* es $n/2+n/2=n$, esto es, tendría que recorrer los n/2 elementos de un arreglo y los n/2 elementos del otro. Esto nos lleva a que la ecuación de recursión para el Merge Sort es:

$$T(n) = 2T\left(\frac{n}{2}\right) + n \qquad (3.2)$$

Expandamos la expresión unos cuantos pasos más. Si calculamos la ecuación de recurrencia para $T\left(\frac{n}{2}\right)$ obtenemos:

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n/2}{2}\right) + \frac{n}{2} \qquad (3.3)$$

Lo cual nos da:

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2} \qquad (3.4)$$

Si aplicamos la misma ecuación para $T\left(\frac{n}{2}\right)$ obtenemos:

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{n}{4} \qquad (3.5)$$

Sustituyendo la ecuación (3.5) en la (3.4) y la (3.4) en la (3.2), tenemos:

$$T(n) = 2\left(2\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + \frac{n}{2}\right) + n \qquad (3.6)$$

Podemos observar que en la ecuación (3.6) los denominadores de la n se eliminan con las multiplicaciones de los números 2 que son coeficientes de la T. Usando un poco de álgebra en la ecuación 3.6 obtenemos:

$$T(n) = n + n + \cdots + n \qquad (3.7)$$

Donde tenemos tantas n como subdivisiones se pueden hacer de los arreglos para llegar a tener arreglos de un solo elemento, como se muestra en la Figura 3.3.

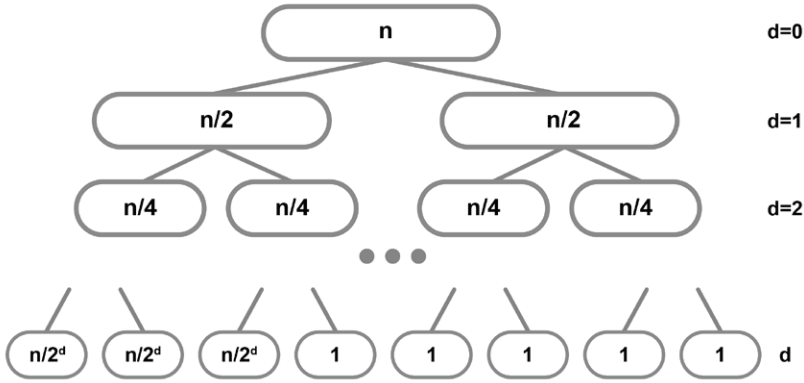


Figura 3.3 Árbol binario que resulta del proceso de división del Merge Sort

En la Figura 3.3 se puede observar que, en cada nivel del árbol el número de elementos que hay en cada subarreglo está dado por $\frac{n}{2^d}$ donde d es la profundidad del nivel (recuerda que la profundidad de la raíz es d=0). Además, sabemos que en el último nivel el número de elementos de cada subarreglo es 1, esto implica que:

$$\frac{n}{2^d} = 1 \qquad (3.8)$$

Despejando d, encontramos que:

$$d = logn \qquad (3.9)$$

Lo cual significa que en la ecuación (3.7) tenemos $logn$ letras n, por lo que el número total de operaciones en la ecuación recursiva es:

$$T(n) = nlogn \qquad (3.10)$$

Que nos da la complejidad del *Merge Sort:* $O(nlogn)$.

Para obtener la complejidad de la **búsqueda binaria**, el procedimiento es prácticamente el mismo, solosolamente que su ecuación de recurrencia es:

$$T(n) = T\left(\frac{n}{2}\right) + 1 \qquad (3.11)$$

Debido a que solo se sigue el proceso recursivo en uno de los dos arreglos como se muestra en la Figura 3.4 y el número de operaciones no recursivas es solo una, que es ver si el número buscado es el que se encuentra en el centro del arreglo (en realidad son 2 operaciones: ver si el número buscado en menor que el del centro y si no, ver si es mayor, pero en complejidad es una constante y las representamos con un 1).

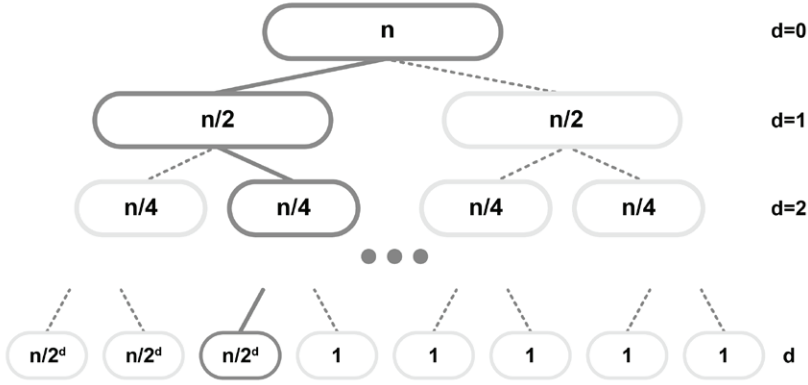


Figura 3.4 Árbol binario para la búsqueda binaria. A diferencia del Merge Sort, aquí solo se sigue una rama de las dos formadas

Si realizamos la expansión de las llamadas recursivas tenemos que:

$$T(n) = \left(\left(T\left(\frac{n}{8}\right) + 1\right) + 1\right) + 1 \qquad (3.12)$$

Lo que finalmente nos llevaría a:

$$T(n) = 1 + 1 + \cdots + 1 \qquad (3.13)$$

Y el número de unos es igual a la profundidad del árbol binario mostrado en la Figura 3.4, el cual es similar al de *Merge Sort* mostrado en la Figura 3.3, con la diferencia de que *Merge Sort* recorre las dos ramas en cada paso y **búsqueda binaria** solamente recorre una, por lo que su profundidad es $d=logn$, lo que nos lleva a:

$$T(n) = logn \qquad (3.14)$$

Que nos daría la complejidad de la **búsqueda binaria:** *O(logn).*

3.1.3 El método maestro

Calcular la complejidad de los algoritmos recursivos no es trivial. En muchas ocasiones se requiere un conocimiento más amplio sobre funciones especiales relacionadas con los logaritmos y sus transformaciones. Afortunadamente, el Teorema 3.1 nos lleva a la concepción de un procedimiento que simplifica mucho los cálculos para muchos (no todos) de los procesos recursivos de **divide y vencerás**. Dicho método se ha dado en llamar el **método maestro** *(Master Method).*

Teorema 3.1. Teorema Maestro

Una ecuación de recurrencia de la forma:

$$T(n) = bT\left(\frac{n}{c}\right) + f(n) \qquad (3.15)$$

Tiene las siguientes formas de solución, donde E=logb/logc, llamado el exponente crítico:

1. Si $f(n) \in O(n^{E-\epsilon})$, para una $\epsilon > 0$, entonces $T(n) = \Theta(n^E)$.
2. Si $f(n) \in \Theta(n^E)$, para una $\epsilon > 0$, entonces $T(n) = \Theta(f(n)logn)$.
3. Si $f(n) \in \Omega(n^{E+\epsilon})$, para una $\epsilon > 0$, y $f(n) \in O(n^{E+\delta})$, con $\delta \geq \varepsilon$, entonces $T(n) = \Theta(f(n))$.

Del Teorema 3.1 se deriva el **método maestro**, el cual es una versión más práctica que se define como se indica a continuación.

Método Maestro

Si la ecuación recursiva es de la forma:

$$T(n) = bT\left(\frac{n}{c}\right) + O(n^d) \qquad (3.16)$$

Que es de la misma forma que la ecuación (3.15), pero con d el exponente de las operaciones no recursivas y d≥0.

Su solución es:

$$T(n) = \{O(n^d logn) \quad si\ b = c^d \quad Caso\ 1\ O(n^d) \quad si\ b < c^d \quad Caso\ 2\ O(n^{log_c b}) \quad si\ b > c^d \quad Caso\ 3$$

Es importante señalar que en el caso 1, la base del logaritmo no importa, pero en el caso 3 la base del logaritmo sí importa y debe ser b.

Apliquemos el **método maestro** a los dos ejemplos cuya complejidad fue calculada en el punto anterior: *Merge Sort* y **búsqueda binaria**.

La ecuación de recurrencia para *Merge Sort* es la mostrada en (3.2) y aquí la colocamos de nuevo:

$$T(n) = 2T\left(\frac{n}{2}\right) + n \qquad (3.17)$$

Podemos observar que b = 2, c = 2 y d = 1, por lo que c^d=2 y b=2, es decir $b=c^d$, lo que nos lleva al caso 1 donde la complejidad es $O(n^d\ logn)$ y sustituyendo el valor de d tenemos que $O(nlogn)$, la cual es la misma que nos dio en la sección anterior para *Merge Sort*.

En el caso de **búsqueda binaria**, la ecuación de recurrencia es:

$$T(n) = T\left(\frac{n}{2}\right) + 1 \qquad (3.18)$$

Donde b = 1, c = 2 y d = 0 (ya que n^0=1) entonces c^d=1 y b=1, es decir $b=c^d$, lo que nos lleva al caso 1 donde la complejidad es $O(n^d\ logn)$ y sustituyendo el valor de d tenemos que $O(logn)$, la cual es la misma que nos dio en la sección anterior para **búsqueda binaria**.

3.1.4 Ejemplo de implementación

Veamos cómo se implementa un algoritmo **divide y vencerás** en algún lenguaje de programación. Vamos a usar C++, aunque, como todo algoritmo computacional, su implementación se puede hacer en cualquier lenguaje de programación.

La implementación de un algoritmo de este tipo es similar en lo general, esto es, todos siguen el algoritmo 3.1, pero cada uno de ellos tiene cosas diferentes en lo particular, por ejemplo: para el caso del *Merge Sort* se hacen dos llamadas recursivas mientras que para la **búsqueda binaria**, se hace una solamente. Además, el algoritmo *Merge Sort* manda a llamar el método Merge, cosa que no hace la **búsqueda binaria**.

Teniendo todo esto en cuenta, vayamos a la implementación del *Merge Sort* para tenerlo como ejemplo de este tipo de algoritmos.

La implementación que se presenta a continuación se hace únicamente con funciones que pueden ser llamadas en forma independiente desde otras funciones o desde el main. En algunas implementaciones colocan el *Merge Sort* como un método de una clase para evitarse el enviar los arreglos o vectores como parámetros. La implementación que se presenta guarda los datos en un **Vector**, el cual, por simplicidad será entero, pero se puede extender fácilmente a cualquier tipo, inclusive **objetos** que tienen una llave (*key*), mediante la cual se pueden ordenar. Además se agrega una función **leerDeArchivo** para leer los datos a ser ordenados directamente de un archivo texto, cuyo nombre recibe como parámetro en un *string* y los coloca en un vector que también recibe como parámetro; y el método **imprimeDatos**, que se utiliza para ver los datos guardados en el **Vector** que recibe como parámetro. A continuación, se presenta el código.

```
1 ////////////////////////////////////////////////////////////
2 // Implementación del Merge Sort                          //
3 ////////////////////////////////////////////////////////////
4
5 #include <iostream>
6 #include <fstream>
7 #include <vector>
8
9 using namespace std;
10
11 // imprime el contenido del vector de datos
12 void imprimeDatos(vector<int> datos){
13   for (int i = 0; i < datos.size(); i++){
14     cout << datos[i] << " ";
15   }
16   cout << endl;
17 }
18
19 // Lee un archivo de números enteros al vector de datos
20 void leerDeArchivo(vector<int> &datos, string archivo){
21   int n, numero;
22   ifstream miArchivo(archivo);
23   if (miArchivo.is_open()){
24     miArchivo >> n;
25     for (int i = 0; i < n; i++){
26       miArchivo >> numero;
27       datos.push_back(numero);
28     }
29     miArchivo.close();
30   }
31   else cout << "No se puede abrir el archivo";
32 }
33
34 // Ordena el arreglo usando Merge Sort
35 void combina(vector<int> &datos, int inicio, int medio, int fin){
36   vector<int> temp(datos.size(),0);
37   int i = inicio;
38   int j = medio+1;
39   int k = inicio;
40   while (i < medio+1 && j < fin+1){
41     if (datos[i] < datos[j])
42       temp[k++] = datos[i++];
43     else
44       temp[k++] = datos[j++];
45   }
46   while (i < medio+1)
47     temp[k++]=datos[i++];
48   while (j < fin+1)
49     temp[k++]=datos[j++];
50   for (int i=inicio; i <= fin; i++)
51     datos[i] = temp[i];
52 }
53
54 // Ordena el arreglo usando Merge Sort
55 void mergeSort(vector<int> &datos, int inicio, int fin){
56   if (inicio < fin){
57     int medio = (inicio+fin)/2;
```

```
58      mergeSort(datos, inicio, medio);
59      mergeSort(datos, medio+1, fin);
60      combina(datos, inicio, medio, fin);
61    }
62 }
63
64 int main(){
65      vector<int> datos;
66      string archivo;
67      cout << "Archivo de datos (con extensión): ";
68      cin >> archivo;
69      leerDeArchivo(datos, archivo);
70      cout << "Los números leídos son: ";
71      imprimeDatos(datos);
72      mergeSort(datos,0,datos.size()-1);
73      cout << "Los números ordenados por burbuja son: ";
74      imprimeDatos(datos);
75 }
```

En la línea 55 se encuentra la función principal para el *Merge Sort* que se llama *mergeSort*, la cual recibe como parámetros los datos a ser ordenados (un vector), así como la parte donde inicia y donde termina el subarreglo que se va a ordenar. Se puede observar que la implementación de esta función sigue exactamente el Algoritmo 3.2, el cual tiene la forma exacta de **divide y vencerás**.

En la línea 35 está la implementación de la función combina (*merge*), la cual es llamada por *mergeSort* una vez que se han terminado las llamadas recursivas. Se puede observar que esta función sigue exactamente el Algoritmo 3.3.

3.2 Algoritmos avaros

Los algoritmos avaros, también llamados codiciosos, *greedy* (en inglés), glotones, etc., son un grupo de algoritmos que hacen honor a su nombre. Estos algoritmos tratan de encontrar la solución a un problema avanzando paso a paso y en cada uno de ellos tratan de mejorar la solución obtenida lo "más" que se pueda (de ahí viene su nombre) es decir, hasta que la solución ya no puede ser mejorada más. En otras palabras, en cada paso un

algoritmo avaro tiene varias opciones como posibles soluciones parciales del problema. La selección de una de ellas se hace basada en medida de la mejora de la solución que se tenga hasta ahora. Todas las opciones posibles son evaluadas y el algoritmo se queda con la que brinda "más" mejora sobre la solución con la que se cuenta hasta ahora. Continua así hasta que la solución actual ya no puede ser mejorada por las opciones disponibles. En ese momento, el algoritmo supone que se cuenta con la mejor solución posible, sin embargo, esto no siempre sucede así.

El funcionamiento de los algoritmos avaros se puede ver como una especie de búsqueda local en la que estando en un punto del espacio de búsqueda (el espacio de todas las posibles soluciones) solo se pueden ver las opciones que están más cercanas al punto actual (otras posibles soluciones). El algoritmo procede a evaluar cada una de las opciones de acuerdo con una métrica establecida (la cual varía de un problema a otro) y se mueve a la opción que "más" mejora la solución actual. Si ninguna de las opciones mejora a la actual, el algoritmo se detiene dando como resultado la solución actual. El Algoritmo 3.4 contiene el algoritmo general.

Algoritmo 3.4. Algoritmo avaro

Entrada: el problema a resolver y una métrica para evaluar soluciones parciales.

Salida: la mejor solución encontrada.

1. Iniciar con una solución parcial del problema y hacerla la solución actual

2. Obtener todas las posibles soluciones parciales que se pueden generar a partir de la solución actual

3. Evaluar cada una de las opciones disponibles en base a una métrica predefinida

4. Si ninguna de las opciones mejora la solución actual, el algoritmo termina. Ir a 6.

5. Seleccionar la opción que más mejora la solución actual y hacerla la nueva solución actual. Ir a 2.

6. Regresar la solución actual como la solución del problema.

Debido a su mecanismo de funcionamiento, lo más que pueden asegurar, los algoritmos avaros, es llegar a un óptimo local, por lo que dependiendo de la forma que tenga el espacio de búsqueda, por ejemplo, en cuanto al número de óptimos locales y dependiendo de la solución parcial en la que inicia el algoritmo, el resultado puede o no encontrar la solución óptima del problema.

Debemos considerar que los algoritmos avaros son realmente muy simples de implementar, pero hay que recordar que lo más común es que los algoritmos avaros no encuentren la solución correcta para un problema dado. Normalmente, la prueba de que un algoritmo de este tipo es correcto es realmente complicada. No obstante, este paradigma de diseño de algoritmos se sigue utilizando porque hay algunos problemas muy importantes cuya solución siempre puede ser obtenida usando un algoritmo avaro, lo cual es una gran ventaja, debido , como ya se comentó, a su sencillez de implementación.

3.2.1 Ejemplo de funcionamiento

El funcionamiento de un algoritmo avaro se puede visualizar mejor con un problema muy simple. Se tiene un conjunto A de dígitos y se quiere formar con ellos, sin repetirlos, el mayor número entero que sea posible. La solución óptima a este problema se puede obtener fácilmente aplicando el algoritmo 3.5, el cual es una forma particular del Algoritmo 3.4.

Algoritmo 3.5. Formar el mayor número entero
Entrada: un conjunto A de dígitos. **Salida:** el mayor entero formado con los dígitos del conjunto A, sin repetición.
1. Iniciar con un entero E formado por 0 dígitos 2. Si el conjunto A es vacío ir a 7 3. Seleccionar el mayor dígito d del conjunto A 4. Concatenar d al entero E 5. Borrar d de A 6. Ir a 2 7. Regresar E como la solución del problema

Se puede observar que el algoritmo 3.5 es avaro debido a que en el paso 3 se selecciona, de todas las opciones posible, es decir, de todos los dígitos en A, el "más" grande de todos. Es claro que la métrica para seleccionar una de las opciones es "el que tenga mayor valor", porque será el que más grande haga el entero for-

mado al ser concatenado al mismo. La solución ya no se puede mejorar cuando el conjunto A está vacío.

Apliquemos el Algoritmo 3.5 al conjunto A = {3, 8, 1, 7, 9} por algunos pasos:

Iteración 1

1. Iniciamos con E = nulo

2. A no es vacío por lo que seguimos al paso 3

3. Como 9 es el mayor dígito en A hacemos d = 9

4. Concatenamos el 9 a E, E = 9

5. Borramos el 9 de A, A = {3, 8, 1, 7}

Iteración 2

2. A no es vacío y continuamos al paso 3

3. Ahora, 8 es el mayor dígito en A hacemos d = 8

4. Concatenamos el 8 a E, E = 98

5. Borramos el 8 de A, A = {3, 1, 7}

Iteración 3

2. A no es vacío y continuamos al paso 3

3. Ahora, 7 es el dígito mayor en A hacemos d = 7

4. Concatenamos el 7 a E, E = 987

5. Borramos el 7 de A, A = {3, 1}

Iteración 4

2. A no es vacío y continuamos al paso 3

3. Ahora, 3 es el dígito mayor en A hacemos d = 3

4. Concatenamos el 3 a E, E = 9873

5. Borramos el 3 de A, A = {1}

Iteración 5

2. A no es vacío y continuamos al paso 3

3. Ahora, 1 es el dígito mayor en A hacemos d = 1

4. Concatenamos el 1 a E, E = 98731

5. Borramos el 1 de A, A = {}

Iteración 6

2. A es vacío y continuamos al paso 7

7. Regresamos E = 98731

El método avaro funcionó perfectamente bien para este problema y nos entregó la solución óptima. Sin embargo, no siempre sucede así en otros problemas, como se muestra en el siguiente ejemplo.

En un país A existen monedas de 50 centavos, 25 centavos, 10 centavos y 5 centavos. Dada una cantidad de dinero D, ¿cuál es la cantidad mínima de monedas que requerimos para formarla? Podemos aplicar un algoritmo avaro como el mostrado en el Algoritmo 3.6.

Algoritmo 3.6. El problema de las monedas

Entrada: un conjunto de tipos de monedas disponibles y la cantidad D de dinero a ser formada.

Salida: el menor número de monedas requerido, así como los tipos para formar la cantidad D.

1. Hacer la cantidad d = 0 (cantidad formada) y n = 0 (número de monedas usadas)

2. Si la cantidad d = D ir a 4

3. Seleccionar la moneda más grande M que al ser sumada a d dé una cantidad menor a D

    a. Hacer d = d + D y n = n + 1

    b. Ir a 2

4. Regresar n como la respuesta

5. FIN

Por ejemplo, si tenemos una cantidad D = 175 centavos y queremos saber cuál es la menor cantidad de monedas que la puede formar aplicamos el Algoritmo 3.6 de la siguiente forma:

Iteración 1

1. d = 0, n = 0

2. d < D y continuamos a 3

3. M = 50 por lo que d = 0 + 50 = 50 y n = 0 + 1 = 1

Iteración 2

2. d < D y continuamos a 3

3. M = 50 por lo que d = 50 + 50 = 100 y n = 1 + 1 = 2

Iteración 3

2. d < D y continuamos a 3

3. M = 50 por lo que d = 100 + 50 = 150 y n = 2 + 1 = 3

Iteración 4

2. d < D y continuamos a 3

3. M = 25 por lo que d = 150 + 20 = 175 y n = 3 + 1 = 4

Iteración 5

2. d = D por lo que terminamos y vamos a 4

4. Regresar n = 4 (3 monedas de 50 y una de 25 forman la cantidad)

Todo parece indicar que el algoritmo avaro funciona muy bien para este problema, ya que si lo hacemos a mano encontramos exactamente la misma solución, 4 es el menor número de monedas con el que podemos formar la cantidad 1.75. Veamos otro caso del mismo problema. Tenemos un país B con monedas de 25 centavos, 20 centavos, 10 centavos y 5 centavos. Apliquemos el mismo algoritmo a la cantidad de dinero D = 40 centavos.

Iteración 1

1. d = 0, n = 0

2. d < D y continuamos a 3

3. M = 25 por lo que d = 0 + 25 = 25 y n = 0 + 1 = 1

Iteración 2

2. d < D y continuamos a 3

3. M = 10 por lo que d = 25 + 10 = 35 y n = 1 + 1 = 2

Iteración 3

2. d < D y continuamos a 3

3. M = 5 por lo que d = 35 + 5 = 40 y n = 1 + 1 = 3

Iteración 4

2. d = D por lo que terminamos y vamos a 4

4. Regresar n = 3 (1 moneda de 25, 1 de 10 y 1 de 5 forman la cantidad)

Es claro que el algoritmo avaro no funcionó en este caso porque el menor número de monedas para formar 40 centavos es 2 (2 de 20) mientras que el algoritmo regresa 3.

Esto comprueba que el algoritmo avaro 3.6, no es un algoritmo correcto, es decir, no resuelve **todas** las instancias del problema de las monedas. En este caso, el algoritmo avaro solo resuelve correctamente algunas instancias del problema (como la del país A), pero no todas (no para el país B).

Es un buen momento para recordar la definición de un algoritmo correcto.

Algoritmo correcto

Se dice que un algoritmo es correcto cuando es capaz de resolver correctamente todas las instancias de un problema dado.

Para asegurarse que un algoritmo es correcto debe haber una demostración formal (matemática), la cual muchas veces no se puede obtener de una forma sencilla. Esta demostración es independiente de la complejidad y de la sencillez de la implementación de un algoritmo.

Desde luego que hay algoritmos avaros que resuelven problemas más complicados, los cuales son muy utilizados en diversas áreas, tal es el caso del problema del árbol de expansión mínimo, que es resuelto con un par de algoritmos avaros: el algoritmo de **Prim** y el de **Kruskal**. De la misma forma podemos hablar del muy famoso algoritmo de Dijkstra para encontrar la distancia más corta entre dos nodos de un grafo.

La complejidad computacional de un algoritmo avaro no es tan genérica como en el caso de los algoritmos de **divide y vencerás**. Prácticamente, se debe encontrar para cada algoritmo que se diseñe. Los algoritmos avaros de **Prim**, **Kruskal** y Dijkstra y su complejidad serán tratados en el capítulo 4.

Cuando tenemos que pensar en diseñar un algoritmo que resuelva un problema es muy común que una de las primeras opciones que se nos viene a la cabeza es la de un algoritmo avaro. Solo debemos recordar que, en muchos casos, el diseño de un algoritmo avaro no da un algoritmo correcto. La única forma de

estar seguros sería haciendo la demostración matemática, la cual como hemos mencionado con anterioridad, no siempre se puede realizar fácilmente. Las demostraciones de que los algoritmos son correcto para los algoritmos avaros más utilizados, como el caso de **Prim** y **Kruskal**, resultan ser bastante complicadas y quedan fuera del alcance de este libro.

3.2.2 Analizando la complejidad

La complejidad en los algoritmos avaros depende mucho del método que se utilice para seleccionar el elemento que "más" o que "menos" afecte a un determinado criterio previamente establecido.

En el caso del problema de los números (ver primer ejemplo de la sección 3.2.1), el conjunto A contiene números al inicio, el algoritmo tiene que ir seleccionando el elemento mayor (el "más" grande) en cada paso y eso implica que su complejidad dependerá en gran medida del método que siga para seleccionar dicho elemento mayor. Por ejemplo, si se colocan los dígitos en un arreglo y se ordenan usando *Bubble Sort* para obtener el mayor simplemente los tendría que ir tomando del arreglo uno a uno, incrementando su índice. El ordenamiento tendría una complejidad de $O(n^2)$ y la selección del mayor en cada paso es constante, por lo que seleccionar los n mayores tiene una complejidad $O(n)$. La complejidad total de este algoritmo sería $O(n^2+n)=O(n^2)$. Sin embargo, si para el ordenamiento se utiliza un mejor algoritmo, digamos *Merge Sort*, la complejidad de este algoritmo se reduce a $O(nlogn+n)=O(nlogn)$. De la misma forma, si se utiliza una cola con prioridad implementada con un *Max-Heap*, los dígitos se insertarían en $O(nlogn)$ y se sacarían uno a uno con la misma complejidad, por lo que la complejidad del algoritmo avaro total usando una cola con prioridad sería $O(nlogn)$.

3.2.3 Ejemplo de implementación

Las implementaciones de este tipo de algoritmos son muy variadas, ya que, como se comentó anteriormente, su funcionamiento depende mucho de la forma en la que se selecciona el siguiente elemento que "más" o que "menos" afecte a un criterio establecido previamente. Aún así, las implementaciones que usan las estructuras de datos y los algoritmos adecuados no son muy complicadas para la mayoría de los algoritmos de este tipo.

Veamos el problema de los números mostrado en la sección 3.2.1, implementado con un *Max-Heap* en C++, utilizaremos la librería *queue* que tiene implementada la clase **priority_queue**, donde la prioridad de un dígito sería simplemente su valor. Esto significa que el dígito con mayor valor tendría una mayor prioridad y se colocaría al frente de la cola. A continuación se muestra el código.

```
#include <iostream>
#include <queue>

using namespace std;

int main(){
  priority_queue<char> myHeap; // cola con prioridad
  int n; // número de dígitos en el conjunto A
  char d; // un dígito como char

  // Pedir los dígitos disponibles al usuario
  // Y colocar uno a uno en la cola con prioridad
  cout << "Cuántos dígitos contiene el conjunto: ";
  cin >> n;
  for (int i=0; i<n; i++){
    cout << "Dígito " << i << ": ";
    cin >> d;
    myHeap.push(d); // se coloca en la cola con prioridad
  }

  // Ir sacando dígito por dígito hasta que la cola esté vacía
  // Los dígitos se colocan en un string para su presentación final
  string numero = ""; // inicia con el string nulo
  while (!myHeap.empty()){
    numero += myHeap.top();
    myHeap.pop();
  }
  // Se imprime la respuesta
  cout << "El máximo entero formado es: " << numero << endl;
}
```

Para este problema, si los dígitos se manejan como char (línea 9) es más fácil ir formado la solución final con la simple concatenación de caracteres (línea 25) para formar un *string*. Por esa razón, los dígitos tomados del usuario se leen como char (línea 17). Esto obliga a que la cola con prioridad se declare para contener char (línea 7). Aprovechando que el ASCII de un dígito está en el mismo orden que su valor (el 9 tiene mayor ASCII que el 8 y también mayor valor numérico) la cola con prioridad funciona para dígitos que son tratados como char de la misma forma que si son tratados como int.

3.3 Programación dinámica

Algunos algoritmos recursivos, aunque su implementación es realmente simple, elegante y directa, tardan una gran cantidad de tiempo en ejecutarse debido, principalmente a que realizan cálculos repetidos una y otra y otra vez, por lo que quedan escondidos en la recursión, lo que los hace difíciles de detectar. El tiempo de ejecución para estos algoritmos puede reducirse si dedica una memoria a guardar los resultados de cálculos que ya se hayan hecho y, cando se requieran nuevamente en lugar de volver a hacer el cálculo completo, simplemente se toma el resultado de la memoria si es que ya lo tenemos almacenado y si no, se realiza el cálculo y se guarda en la memoria para futuros usos. Al proceso de ir almacenando soluciones candidatas se le conoce como **memoization**. En 1957, Richard Bellman propuso un método especial para resolver problemas de programación dinámica con esta y algunas otras características que comentaremos más adelante. Dicho método se ha tomado como base para resolver otros problemas con características similares y el nombre de los problemas originales se mantuvo dando lugar al enfoque de programación dinámica para el diseño de algoritmos. Una de las características fundamentales de todos los algoritmos diseñados

con este enfoque es que logran disminuir la complejidad de algunos cálculos de exponenciales a polinomiales.

Otra de las características de un algoritmo de programación dinámica es que resuelve un problema grande y complejo descomponiéndose en subproblemas más pequeños, cuya solución es más simple, de la misma forma que lo hace el enfoque **divide y vencerás**. La diferencia con la técnica anterior es que en la programación dinámica la solución del problema más simple podría o no ser parte de la solución del problema más complejo, es decir, es solo una solución candidata, mientras que en **divide y vencerás**, la solución de los problemas más pequeños siempre son parte de la solución del problema mayor.

La tercera característica de la programación dinámica es la utilización de una ecuación de recurrencia que relaciona la solución de un problema con las soluciones obtenidas de los subproblemas en los que se divide.

El resultado es un enfoque muy poderoso para diseñar algoritmos llamado programación dinámica (por el tipo de problemas originales en los que Bellman lo aplicó), cuyo funcionamiento se verá a continuación con algunos ejemplos.

3.3.1 Ejemplo de funcionamiento: la serie de Fibonacci

Uno de los ejemplos más simples para entender el funcionamiento de los algoritmos de programación dinámica es la serie de Fibonacci. En 1202, en su famosa obra *Liber Abaci*, Leonardo de Pisa, alias Fibonacci, planteó, un problema relacionado con conejos: un hombre tenía una pareja de conejos y deseaba saber cuántos se podrían reproducir en un año a partir de la pareja inicial si se considera que una pareja tiene una nueva pareja (sus hijos) en un mes y que una pareja nueva llega a la madurez a

partir del segundo mes y se empiezan a reproducir. La solución a dicho problema se realiza por medio de una serie que puede ser definida en forma recursiva:

Serie de Fibonacci

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases} \quad (3.19)$$

El código creado en forma directa de la definición (3.19) se observa a continuación:

```
1 int fibonacciDirecto(int n){
2 // implementación directa de la definición de factorial
3   if (n <= 1)
4     return n;
5   else
6     return fibonacciDirecto(n-1) + fibonacciDirecto(n-2);
7 }
```

Si corremos el código, el tiempo de respuesta es bastante bueno hasta alrededor del número 41. De ahí en adelante, el cálculo tarda mucho (por ejemplo, se nota mucho con 45). La razón es muy simple, el cálculo de algunos números se hace repetidas veces, por ejemplo, en la figura 3.5 se observa el árbol para hacer el cálculo del número de Fibonacci de 5.

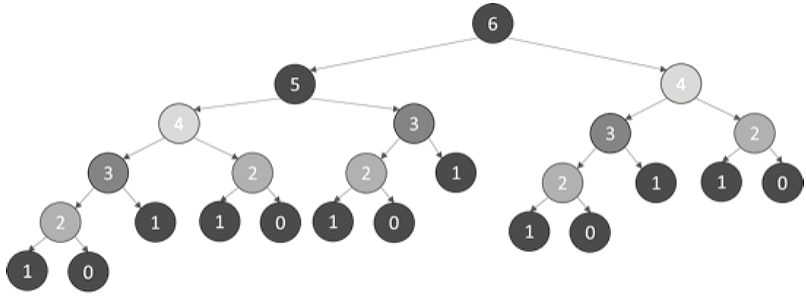


Figura 3.5 Árbol para el cálculo del número de Fibonacci de 6. Cada círculo con un número n representa una llamada a la función fib(n)

En la Figura 3.5 se puede observar que el cálculo de fib(4) (círculos amarillos) se hace dos veces, el cálculo de fib(3) (círculos verdes) se hace 3 veces y el cálculo de fib(2) (círculos naranjas) se hace 5 veces. Eso implica repetir una gran cantidad de trabajo. Si el número al que se le quiere calcular su número de Fibonacci es grande, por ejemplo 100, la cantidad de veces que se llama a la función para calcular un mismo número llega a ser de miles, millones o tal vez más veces.

La solución es muy simple, aunque implica utilizar algo de memoria extra y consiste en ir guardando los cálculos ya realizados en dicha memoria. Cuando se tenga que hacer un cálculo nuevo, primero revisamos en la memoria si ya lo tenemos calculado, de ser así, simplemente lo usamos y si no, entonces lo hacemos y al terminarlo lo guardamos en la memoria.

En el código mostrado a continuación se puede ver que se utiliza un vector (o arreglo) para guardar los números de Fibonacci ya calculados, esto es, para hacer memoization.

```
1 // se inicializa con los valores del fibonacci de 0 y 1
2 vector<int> memo = {0, 1};
3
4 int fibonacciPD(int n){
5   if (n < memo.size()) // si n está en memo se obtiene su valor
6     return memo[n];
7   else{ // si no está, se calcula y se coloca el resultado en memo
8     int f = fibonacciPD(n-1) + fibonacciPD(n-2);
9     memo.push_back(f);
10    return f;
11  }
12 }
```

Ahora podemos calcular el Fibonacci de número mayores al 41 y lo regresa en un tiempo muy corto (por ejemplo, el 45 ya lo hace de inmediato).

En realidad, es posible eliminar la llamada recursiva manteniendo la relación de recurrencia. Esto se debe a que en la línea 5 podemos observar que cada llamada a fibonacciPD lo único que hace es verificar si ya está en el arreglo y como cada número depende de los resultados de números menores que él es posible sustituir la llamada a la función directamente por el acceso al arreglo. Esto lleva a una implementación iterativa usando un ciclo de menor a mayor (por ejemplo, un for) porque, como se dijo anteriormente, el Fibonacci de un número depende solo del Fibonacci de números menores que él, lo cual es mucho más rápido que las llamadas recursivas. El código de esta implementación se muestra a continuación usando un arreglo en lugar de un vector:

```
1 int fibonacciIterativoPD(int n){
2   int F[n];
3   F[0] = 0;
4   F[1] = 1;
5   for (int i = 2; i <= n; i++)
6     F[i] = F[i-1] + F[i-2];
7   return F[n];
8 }
```

Aunque no existen llamadas recursivas, la idea de la programación dinámica se mantiene intacta en esta implementación iterativa, ya que la relación de recurrencia de un número con respecto a dos números calculados previamente se encuentra implementada en el código.

Al analizar con detenimiento el problema se puede observar que, en realidad, en la implementación iterativa anterior no se requiere todo el arreglo, sino que basta con guardar los dos números anteriores al número cuyo Fibonacci se desea calcular. Además de que nos reduce el tiempo de ejecución como en la implementación anterior, también nos reduce al mínimo la cantidad de memoria extra utilizada para memoization. A continuación se muestra el código:

```
1  int fibonacciIterativo(int n){
2    if (n <= 1)
3      return n;
4    else{
5      int n1 = 0, n2 = 1, f;
6      for (int i = 2; i <= n; i++){
7        f = n1 + n2;
8        n1 = n2;
9        n2 = f;
10     }
11     return f;
12   }
13 }
```

Este último caso es muy particular del problema del cálculo de números de Fibonacci, lo que significa que cada problema debe ser analizado para ver cómo podemos hacer más eficiente la solución propuesta.

3.3.2 Análisis de complejidad

En programación dinámica no existe una forma general de obtener la complejidad como en divide y vencerás. Esta dependerá de la forma recursiva que tenga el problema que se está resolviendo. En este caso, aunque se trata de encontrar la solución de un problema dividiéndolo en subproblemas más simples no se puede aplicar el **método maestro** debido a que, como se describió anteriormente, en programación dinámica, la solución del subproblema puede o no tomarse para solucionar el problema mayor, a diferencia de divide y vencerás donde esta era parte de la solución del problema mayor.

Sin embargo, se pueden analizar varias cosas con respecto a la complejidad general de un problema resuelto con programación dinámica. La más importante es la relación entre la complejidad en espacio y la complejidad en tiempo.

Como ya se explicó anteriormente, para poder disminuir la complejidad en tiempo, la programación dinámica requiere ir guardando los resultados de los subproblemas que ya se resolvieron con el propósito de no hacer el cálculo más de una vez. Claramente, el uso de *memoization* aumenta la complejidad en espacio. El programador tiene que analizar si el aumento en la complejidad en espacio vale la pena con respecto a la mejora pro-vocada en la complejidad en tiempo. Si es así se debe usar pro-gramación dinámica.

La historia ha mostrado que en los casos en los que se ha usado programación dinámica, la mejora causada en el tiempo al usar *memoization* casi siempre es positiva, sin embargo, cada vez que se usa programación dinámica será necesario hacer cálculos de ambas complejidades: tiempo y espacio; y decidir si se debe usar o no.

En general, al analizar la complejidad en tiempo se puede observar que esta dependerá del número de estados únicos que tenga el problema en cuestión. Un estado único es aquel cuyo cálculo se hace una sola vez y se guarda en la estructura para hacer memoization. Por ejemplo, en el caso de Fibonacci analizado en la sección anterior podemos ver que un estado único es el número de Fibonacci de cada uno de los números desde 0 hasta n, que es el número de Fibonacci que se quiere obtener. Las acciones de verificar en la estructura para ver si ya se calculó el número de Fibonacci requerido y, en caso de que así sea, tomarlo y regresarlo como el valor buscado, si se usa una estructura de datos adecuada, se puede considerar constante. Así que la complejidad en tiempo dependerá del número de estados únicos, tal como se comentó anteriormente.

El cálculo de la complejidad en tiempo de Fibonacci recursivo no es tan simple, pero se puede establecer en $O(2^n)$, es decir, es exponencial. Sin embargo, cuando se usa programación dinámica el orden es igual al número de estados únicos y, en este caso, es claramente lineal, es decir, $O(n)$. Por otro lado, el cálculo de la complejidad en espacio de Fibonacci con programación dinámica es también $O(n)$ debido a que tienen que ir guardando todos los valores previamente calculados, mientras que la complejidad es espacio de la versión recursiva es así mismo proporcional a la profundidad máxima del árbol de recursión, que es el máximo número de nodos del árbol (llamadas a funciones) que estarán en la memoria en un momento dado, es decir, también es $O(n)$. Desde luego que si analizamos la implementación iterativa se observa que su complejidad en tiempo es $O(n)$, mientras su complejidad en espacio es constante, es decir, $O(1)$, porque solo requiere tres variables en todo momento.

En resumen, cuando se resuelva un problema usando programación dinámica se tiene que calcular la complejidad en tiempo

y en espacio y hacer el análisis del beneficio obtenido. La complejidad de cada problema dependerá de su función de recurrencia así como de la estructura de datos con la que se implementa la memoization.

3.3.3 Ejemplo de aplicación de la programación dinámica

En la sección 3.3.1 se hizo un ejemplo de implementación muy simple de la programación dinámica usando el cálculo de los números de Fibonacci. Ahora, se presenta la solución del problema de las monedas, cuya solución no se pudo obtener para todos los casos usando el enfoque de algoritmos avaros (sección 3.2).

El problema general de las monedas se puede describir como sigue.

Problema de las monedas

En un país X tienen n monedas de diferentes denominaciones: c_1, c_2, …, c_n, donde c_i es la denominación de la moneda i. Dada una cantidad entera de dinero C, encontrar el mínimo número de monedas con el que se puede formar dicha cantidad.

Lo primero que tenemos que hacer para usar programación dinámica es establecer la relación de recurrencia entre el problema actual y la solución de problemas más simples. Esta relación de recurrencia es muy particular de cada problema y se debe tomar un buen tiempo para analizarlo y poder establecer dicha relación.

En problema de las monedas, cada vez que usamos una de las n monedas, digamos la moneda i, con denominación ci, estamos aumentando el número de monedas en 1 y disminuyendo la cantidad a ser formada en la denominación de la moneda seleccionada, es decir, la nueva cantidad es C = C-c_i y repetimos el problema, ahora con la nueva cantidad. Debemos tomar en cuenta que en cada paso hay a lo más n opciones diferentes para formar la nueva cantidad debido a que hay n diferentes monedas. De todas las opciones tenemos que seleccionar la que nos entregue el menor número de monedas. Entonces, si la función que encuentra el menor número de monedas para una cantidad C la llamamos MinNúmMonedas, la relación recursiva la podemos definir como se muestra a continuación.

Relación recursiva para el problema de las monedas

$$MinNúmMonedas(C) = min\{MinNúmMonedas(C - c_1) + 1\ MinNúmMonedas(C - c_2) + 1\ \ldots\ MinNúmMonedas(C - c_n) + 1 \quad (3.20)$$

Donde **min** significa que se selecciona la opción que regrese el menor número de monedas y el **+1** se hace en cada caso porque se seleccionó una de las monedas.

La función recursiva (3.20) se podría implementar fácilmente en cualquier lenguaje de programación, sin embargo, tendríamos el mismo problema que con Fibonacci, es decir, se tendría que hacer el cálculo varias veces para la misma cantidad. El algoritmo recursivo en pseudocódigo lo podemos ver en el algoritmo 3.7.

Algoritmo 3.7 Mínimo número de monedas recursivo

Entrada: la cantidad C y el conjunto disponible de monedas S

Salida: el menor número de monedas del conjunto S para formar la cantidad C

```
minNúmMonedasRecursivo(C,S)
  if C == 0 return 0
    mínimo = ∞
    for i en S
     if C >= ci
       num = minNúmMonedasRecursivo(C-ci,S) + 1
       if num < mínimo
          mínimo = num
  return minimo
```

Aplicando programación dinámica podemos evitar el problema de la repetición usando memoization, es decir se van guardando los resultados ya obtenidos en una estructura de datos y, cada vez que se llame a la función, primero vemos si ya tenemos el cálculo en la estructura. Si lo tenemos, simplemente lo tomamos, si no, lo calculamos (solo una vez) y lo guardamos en la estructura.

El segundo paso en el uso de programación dinámica es la selección de una estructura de datos adecuada para hacer memoization. En el caso del problema de las monedas bastará un arreglo unidimensional de C+1 elementos, cuyos índices repre-

sentan las cantidades que se quieren formar, por lo que van de 0 a la cantidad C y donde el valor de la celda i será el mínimo número de moneda para formar la cantidad i. Inicializamos la estructura con 0 en la celda 0 y con ∞ en las restantes. La estructura para memoization se observa en la Figura 3.6.

Figura 3.6 Estructura de datos para hacer memoization en el problema de las monedas. Los índices representan cantidades de dinero y el valor en la celda el mínimo número de monedas para forma dicha cantidad

La primera llamada a la función se hace con la cantidad C y se van haciendo llamadas recursivas hacia atrás hasta llegar a 0. En el Algoritmo 3.8 se observa una implementación con programación dinámica iterativa (sin hacer recursión) donde se aprovecha la estructura de datos y la relación de recurrencia descrita anteriormente para ir haciendo los cálculos de 0 hasta C.

Algoritmo 3.8 Mínimo número de monedas con programación dinámica

Entrada: la cantidad C y el conjunto disponible de monedas S

Salida: el menor número de monedas del conjunto S para formar la cantidad C

```
minNúmMonedasPD(C, S)
  M[0] = 0
  for cantidad = 1 to C
    M[cantidad] = ∞
    for i en S
      if cantidad >= ci
          num = M(cantidad - ci, S) + 1
          if num < M[cantidad]
              M[cantidad] = num
  return M[C]
```

Se puede hacer una versión recursiva del Algoritmo 3.8, pero las llamadas recursivas son más lentas que los accesos a un arreglo y por eso se prefiere esta implementación iterativa donde se simula la relación de recurrencia al calcular un valor actual basado en valores calculados anteriormente.

3.4 Backtracking

Existen problemas para los cuales no se ha logrado encontrar un algoritmo eficiente para solucionarlos. Sin embargo, como son importantes, se tiene que buscar alguna forma de obtener su solución. Esta búsqueda es la base de la investigación en diseño de algoritmos.

En algunas ocasiones, la única herramienta con la que se cuenta es la búsqueda exhaustiva, esto es, buscar en todo el es-

pacio de soluciones un elemento que cumpla con las condiciones para ser la solución. Cuando estos problemas son combinatorios, esto es, cuando su solución es una permutación, combinación o subconjunto de un conjunto de valores que pueden tomar las variables involucradas, el espacio de búsqueda es discreto. En este caso, la solución podría ser encontrada revisando todas las posibles soluciones hasta encontrar alguna o algunas (dependiendo de lo que pida el problema) que cumplan con las condiciones para ser la solución deseada.

Un ejemplo de problema combinatorio es el visto en la sección 3.2.1, en donde se trata de encontrar el máximo número entero que se puede formar con un conjunto S de dígitos dado. Sus posibles soluciones son todas las permutaciones de los dígitos en el conjunto S. Si el conjunto S = {1,2,3}, las posibles soluciones (permutaciones) son: 123, 132, 213, 231, 312, 321. Ahora solo bastará con buscar la permutación que formó el número mayor, la cual es 321 y el problema está resuelto.

La búsqueda exhaustiva podría funcionar para solucionar cualquier problema combinatorio, sin embargo en algunos casos, los espacios de solución son infinitos o muy grandes y este proceso se haría imposible de realizar en un tiempo adecuado debido a que el número posible de soluciones crece exponencialmente con el número de valores posibles de las variables. En el caso del problema del número mayor con un conjunto de n dígitos, el número de permutaciones es $O(n!)$, es decir, a medida que n crece, el número de posibles soluciones se hace demasiado grande.

Backtracking es una mejora de la búsqueda exhaustiva, que va construyendo la solución paso a paso, en donde cada paso analiza los posibles valores que pueda tomar una variable. Utiliza una estructura de árbol, cuya raíz es el inicio del problema y cada

nivel está formado por los posibles valores que puede tomar una de las variables involucradas en el problema. Esto significa que el último nivel del árbol contiene todas las posibles soluciones del espacio, por tal razón a este árbol se le conoce como "Árbol del espacio de estados".

Backtracking no forma el árbol desde el inicio, sino que lo va formando usando la regla "Primero profundidad" (que es la regla usada para recorrer todos los nodos de un grafo en el algoritmo "Búsqueda primero en profundidad", DFS por sus siglas en inglés,). Los nodos interiores de este árbol tienen soluciones parciales al problema y las hojas contienen soluciones completas al problema. Al explorar un nodo se verifica que la solución parcial hasta ese momento cumpla con las condiciones de una posible solución, si es así este nodo es clasificado como "promisorio", por lo que se expande y se continúa el proceso de "primero en profundidad". Si, por el contrario, se encuentra que la solución parcial obtenida hasta ese momento no cumple con las condiciones para ser una solución al problema, ese nodo se clasifica como "no promisorio", por lo que se desecha, se regresa a su padre, es decir, se hace *backtracking* y se continúa el proceso con sus hermanos. En el caso que ya no haya más nodos por analizar se hace *backtracking* un nivel más arriba y así sucesivamente. El algoritmo general de *backtracking* se puede ver en el Algoritmo 3.9, el cual se implementa en forma recursiva para realizar el *backtraking* en forma automática.

Algoritmo 3.9 *Backtracking*

Entrada: el nodo raíz donde inicia la búsqueda. Para cada problema se debe conocer el conjunto de variables x_i y sus posibles valores V_i, donde $X = \{x_1:V_1, x_2:V_2, \ldots, x_n:V_n\}$.

Salida: una o más soluciones al problema, si es que existe alguna.

```
Backtracking(nodo):

    if not promisorio(nodo): // si el nodo no es promisorio

        return // backtracking

    if solución(nodo): // se encontró una solución

        imprime(nodo)

        return

    // si el nodo es promisorio y no es una solución se llama a la
función con sus hijos

    for i in hijos(nodo):

        Backtracking(i)
```

Donde:

- **promisorio(nodo):** regresa *true* si el nodo es promisorio
- **solución(nodo):** regresa *true* si nodo es la solución
- **hijos(nodos):** regresa el conjunto de todos los hijos de nodo
- **imprime(nodo):** imprime una solución

Como siempre, el Algoritmo 3.9 es muy general y se tiene que adaptar a cada problema específico, en cuanto a todas las funciones que se especifican, es decir: verificar si el nodo es óptimo, si es una respuesta, generar sus hijos o imprimir una respuesta.

3.4.1 Ejemplo de funcionamiento

Uno de los mejores problemas para entender el funcionamiento de *backtracking* es el de las 8 reinas descrito a continuación:

Problema de las 8 reinas

En un tablero de ajedrez (8X8 casillas) se deben colocar 8 reinas sin que se ataquen. Recuerda que una reina puede atacar vertical, horizontal o diagonalmente, recorriendo cuantas casillas requiera.

El problema se puede generalizar a n reinas, es decir, en un tablero de nXn casillas se colocan n reinas sin que se ataquen.

El problema de las 8 reinas ha sido ampliamente estudiado en inteligencia artificial y se sabe que tiene 92 posibles soluciones, pero si se considera que algunas de ellas son la misma solución simplemente rotando el tablero, en realidad, tiene 12 soluciones distintas.

Para ver el funcionamiento de *backtracking* veamos el problema de las 4-reinas. El problema parte de un tablero vacío, que sería la raíz del árbol, y en cada nivel se va agregando una reina. Las reinas serían las variables y los cuadros vacíos serían los valores que puede tomar. La Figura 3.7 muestra una representación del árbol hasta el primer nivel, es decir, hasta la primera variable que

es la primera reina y que puede tomar 16 posibles lugares porque el tablero inicialmente está vacío.

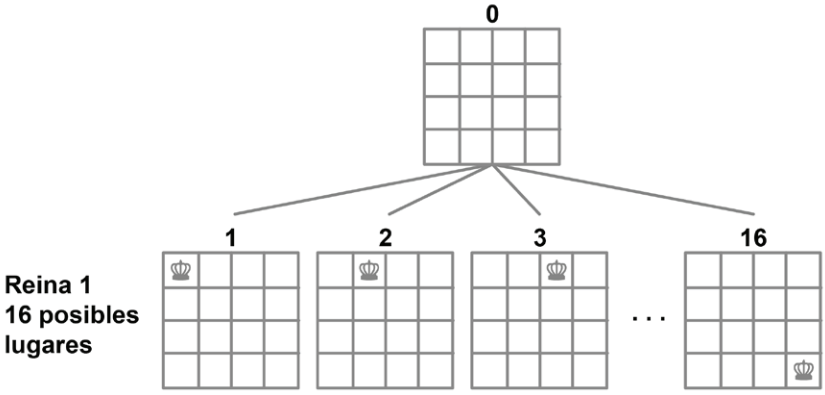


Figura 3.7 Árbol de búsqueda backtracking para el primer nivel del problema de las 4-reinas

En realidad, sabemos que si dos reinas se encuentran en la misma fila, misma columna o misma diagonal se van a atacar, así que le podemos ayudar un poco al algoritmo y restringir la colocación de las reinas a una en cada columna diferente. La Figura 3.8 muestra una representación del primer árbol hasta el primer nivel, es decir, con todos los valores que puede tomar la primera reina. En este caso solo son 4 valores posibles porque la primera reina solamente se puede colocar en la primera columna. Hasta este nivel el árbol cuenta con 5 nodos únicamente.

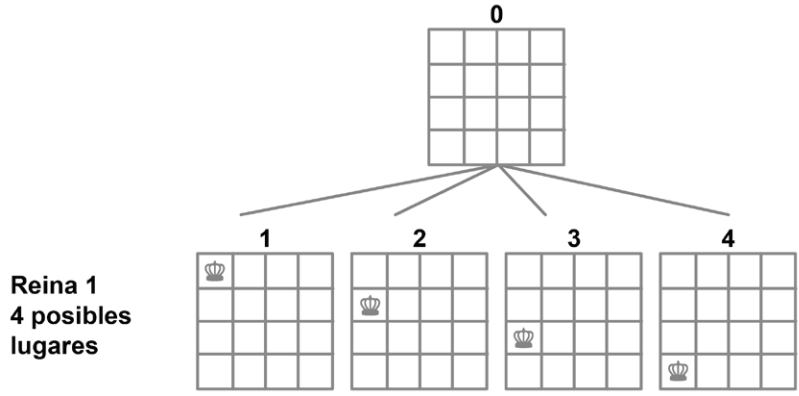


Figura 3.8 Árbol hasta el primer nivel con la restricción de tener una reina por columna

Pero, como se comentó anteriormente, backtracking no genera todo el árbol, ni todo el nivel a la vez, sino que va creando los nodos de acuerdo la regla primero en profundidad, partiendo de la raíz, lo que significa que, partiendo del nodo 0 crea el nodo 1, si es promisorio crea el primer nodo hijo del nodo 1, si no, crea el nodo 2 y así sucesivamente. El árbol parcialmente creado se puede ver en la Figura 3.9.

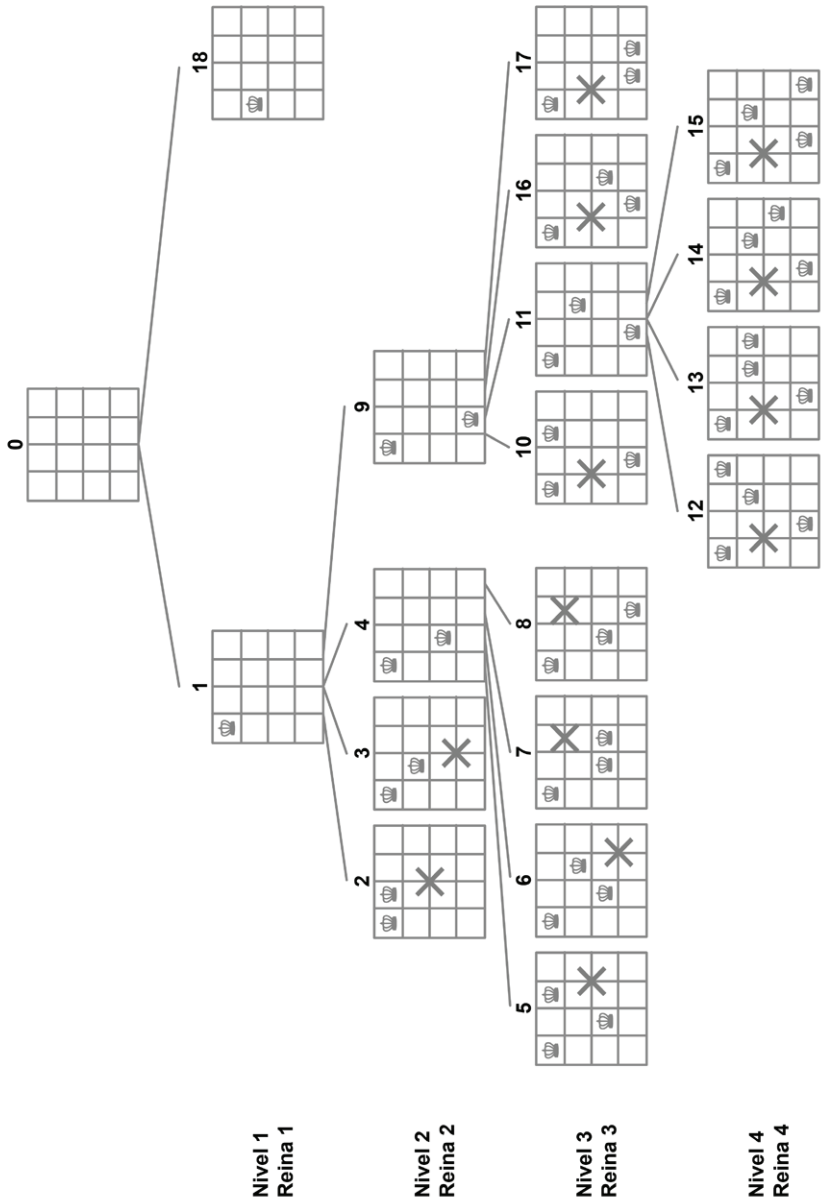


Figura 3.9 Árbol backtracking parcialmente generado.Las cruces rojas indican que no es nodo promisorio. Los números arriba de los nodos indican el orden en el que los nodos son creados

El proceso se puede explicar de la siguiente forma:

1. Inicia en el nodo 0.

a. Es un nodo promisorio

b. No es la solución

c. Se procesa con la regla primero en profundidad y se genera su primer hijo, el nodo 1.

2. Se analiza el nodo 1.

a. Es un nodo promisorio.

b. No es la solución.

c. Se procesa con la regla primero en profundidad y se genera su primer hijo, el nodo 2.

3. Se analiza el nodo 2.

a. No es un nodo promisorio porque viola las condiciones de la solución, ya que las dos reinas se atacan.

b. Se hace *backtracking* hacia su padre, el nodo 1.

4. Continúa el proceso primero en profundidad del nodo 1, se genera su segundo hijo, el nodo 3.

5. Se analiza el nodo 3.

a. No es un nodo promisorio porque viola las condiciones de la solución, ya que las dos reinas se atacan.

b. Se hace *backtracking* hacia su padre, el nodo 1.

6. Continúa el proceso primero en profundidad del nodo 1, se genera su tercer hijo, el nodo 4.

7. Se analiza el nodo 4.

a. Es un nodo promisorio.

b. No es la solución.

c. Se procesa con la regla primero en profundidad y se genera su primer hijo, el nodo 5.

8. Se analiza el nodo 5.

a. No es un nodo promisorio porque viola las condiciones de la solución, ya que las dos reinas se atacan.

b. Se hace *backtracking* hacia su padre, el nodo 4.

9. Continúa el proceso primero en profundidad del nodo 4, se genera su segundo hijo, el nodo 6.

10. Se analiza el nodo 6.

a. No es un nodo promisorio porque viola las condiciones de la solución, ya que las dos reinas se atacan.

b. Se hace *backtracking* hacia su padre, el nodo 4.

11. Continúa el proceso primero en profundidad del nodo 4, se genera su tercer hijo, el nodo 7.

12. Se analiza el nodo 7.

a. No es un nodo promisorio porque viola las condiciones de la solución, ya que las dos reinas se atacan.

b. Se hace *backtracking* hacia su padre, el nodo 4.

13. Continúa el proceso primero en profundidad del nodo 4, se genera su cuarto hijo, el nodo 8.

14. Se analiza el nodo 8.

a. No es un nodo promisorio porque viola las condiciones de la solución, ya que las dos reinas se atacan.

b. Se hace *backtracking*, hacia su padre, el nodo 4.

15. Continúa el proceso primero en profundidad del nodo 4, pero, como ya se generaron todos sus hijos, se hace *backtracking* a su padre, el nodo 1

16. Continúa el proceso primero en profundidad del nodo 1, se genera su cuarto hijo, el nodo 9.

17. Se analiza el nodo 9.

a. Es un nodo promisorio.

b. No es la solución.

c. Se proceso con la regla primero en profundidad y se genera su primer hijo, el nodo 10.

18. Se analiza el nodo 10.

a. No es un nodo promisorio porque viola las condiciones de la solución, ya que las dos reinas se atacan.

b. Se hace *backtracking* hacia su padre, el nodo 9.

19. Continúa el proceso primero en profundidad del nodo 9, se genera su segundo hijo, el nodo 11.

20. Se analiza el nodo 11.

a. Es un nodo promisorio.

b. No es la solución.

c. Se proceso con la regla primero en profundidad y se genera su primer hijo, el nodo 12.

21. Se analiza el nodo 12.

a. No es un nodo promisorio porque viola las condiciones de la solución, ya que las dos reinas se atacan.

b. Se hace *backtracking* hacia su padre, el nodo 11.

22. Continúa el proceso primero en profundidad del nodo 11, se genera su segundo hijo, el nodo 13.

23. Se analiza el nodo 13.

a. No es un nodo promisorio porque viola las condiciones de la solución, ya que las dos reinas se atacan.

b. Se hace *backtracking* hacia su padre, el nodo 11.

24. Continúa el proceso primero en profundidad del nodo 11, se genera su tercer hijo, el nodo 14.

25. Se analiza el nodo 14.

a. No es un nodo promisorio porque viola las condiciones de la solución, ya que las dos reinas se atacan.

b. Se hace backtracking hacia su padre, el nodo 11.

26. Continúa el proceso primero en profundidad del nodo 11, se genera su cuarto hijo, el nodo 15.

27. Se analiza el nodo 15.

a. No es un nodo promisorio porque viola las condiciones de la solución, ya que las dos reinas se atacan.

b. Se hace *backtracking* hacia su padre, el nodo 11.

28. Continúa el proceso primero en profundidad del nodo 11, pero, como ya se generaron todos sus hijos, se hace *backtracking* a su padre, el nodo 9.

29. Continúa el proceso primero en profundidad del nodo 9, se genera su tercer hijo, el nodo 16.

30. Se analiza el nodo 16.

a. No es un nodo promisorio porque viola las condiciones de la solución, ya que las dos reinas se atacan.

b. Se hace *backtracking* hacia su padre, el nodo 9.

31. Continúa el proceso primero en profundidad del nodo 9, se genera su cuarto hijo, el nodo 17.

32. Se analiza el nodo 17.

a. No es un nodo promisorio porque viola las condiciones de la solución, ya que las dos reinas se atacan.

b. Se hace *backtracking* hacia su padre, el nodo 9.

33. Continúa el proceso primero en profundidad del nodo 9, pero, como ya se generaron todos sus hijos, se hace *backtracking* a su padre, el nodo 1.

34. Continúa el proceso primero en profundidad del nodo 1, pero, como ya se generaron todos sus hijos, se hace *backtracking* a su padre, el nodo 0.

35. Continúa el proceso primero en profundidad del nodo 0, se genera su segundo hijo, el nodo 18.

36. Y así sucesivamente, hasta encontrar alguna solución, todas las soluciones o que se indique que la solución no existe.

Se puede observar que el recorrido del árbol es muy similar al que hace la búsqueda primero en profundidad (DFS por sus siglas en inglés), pero en este caso no es necesario llegar hasta la hoja, ya que si el proceso detecta que el nodo actual no puede llevarnos a una solución suspende la expansión y hace *backtracking*. Es una mejora a la búsqueda exhaustiva, una búsqueda más inteligente.

3.4.2 Análisis de complejidad

La búsqueda exhaustiva tiene que generar todas las posibles soluciones y de entre ellas encontrar la o las que cumplan con las características de la solución requerida. El *backtracking* es una búsqueda más inteligente, ya que cuando se da cuenta que la solución parcialmente formada no cumple con las características solicitadas detiene el proceso y no tiene que llegar a formar la solución completa. Eso significa que *backtracking* no disminuye la complejidad de la búsqueda exhaustiva en el peor caso, pero en promedio

sí lo hace, porque tiene la capacidad de detener la expansión de algunas ramas sin tener que llegar a las hojas, es decir, sin tener que formar toda la solución completa.

La complejidad de la implementación *backtracking* no es única para todos los problemas. Para un problema particular dependerá del problema en sí.

Como se comentó anteriormente, *backtracking* es muy utilizada para encontrar la solución de problemas combinatorios para los cuales no existe un algoritmo más eficiente que la búsqueda exhaustiva. Como *backtracking* es en promedio mejor que búsqueda exhaustiva, siempre se trata de hacer esta implementación en lugar de la exhaustiva cuando esto sea posible. Los problemas combinatorios normalmente tienen complejidades exponenciales o superiores en búsqueda exhaustiva y, en el peor de los casos, estas se mantienen para *backtracking*.

3.4.3 Ejemplo de implementación

Como un ejemplo muy simple de implementación de *backtracking* solucionamos el problema de la suma de subconjuntos que se define como:

Problema de la suma de subconjuntos

Dado un conjunto S de números enteros positivos, encontrar los subconjuntos que sumen la cantidad C. Si no se encuentra algún subconjunto que cumpla, se debe responder con el conjunto vacío.

Por ejemplo: si el conjunto S = {2, 3, 7, 9} y C = 9, debemos

entrar cuál es el subconjunto cuyos elementos suman 12. En este caso, la solución es: {2, 7} y {9}. Si C=25 no hay solución y la respuesta sería el conjunto vacío {}.

Para este problema, cada número de los n que contiene el conjunto S se puede usar para la suma o no, esto implica que para resolverlo por fuerza bruta, una tabla de verdad de 2^n renglones nos ayudaría a formar todas las posibles sumas. Si tomamos el conjunto S = {2, 3, 7, 9}, la tabla con sus primeros renglones sería como la que se muestra en la Tabla 3.1.

2	3	7	9	SUMA
F	F	F	F	NA
F	F	F	V	9
F	F	V	F	7
F	F	V	V	16
F	V	F	F	3
F	V	F	V	12
F	V	V	F	10
F	V	V	V	19
V	F	F	F	2
V	F	F	V	11
V	F	V	F	9
V	F	V	V	18
V	V	F	F	5
V	V	F	V	14
V	V	V	F	12
V	V	V	V	21

Tabla 3.1 Ejemplo de tabla de verdad como auxiliar en la formación de los subconjuntos

Es claro que la solución por búsqueda exhaustiva tiene orden $O(2^n)$, por lo que es mejor hacerla con *backtracking*.

La Tabla 3.1 también nos ayuda a pensar en la estructura del árbol, el cual sería binario, donde en cada rama se tomaría o no uno de los números y los nodos representarían la suma obtenida hasta ese momento. La Figura 3.10, muestra el árbol de búsqueda completo.

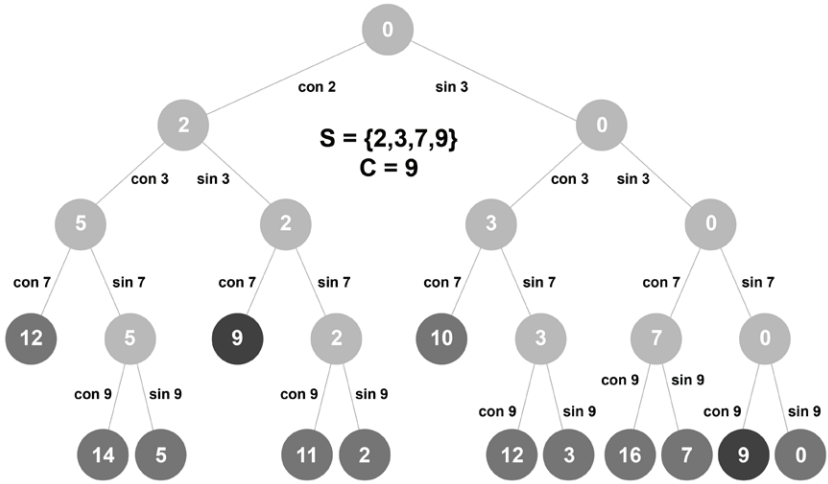


Figura 3.10 Árbol de búsqueda completo para el ejemplo de la suma de subconjuntos

Hay tres condiciones para hacer *backtracking*:

- Cuando la suma de los números que llevamos hasta ahora ya sea mayor que la cantidad C, en este caso, ya no es necesario continuar por esa rama.
- Cuando se llega al último nivel y la suma es diferente de la cantidad C.
- Cuando la suma sea igual a D, en este caso se encontró una solución.

Los nodos de color naranja en la Figura 3.10 muestran los lugares donde se hace *backtracking*.

Y hay una condición de finalización:

- Cuando se termine el árbol y no se haya encontrado ninguna solución.

Los nodos de color verde en la Figura 3.10 muestran las soluciones encontradas en caso de que el proceso continúe hasta encontrar todas las soluciones. De no ser así, se debería salir al encontrar la primera solución.

A continuación, se presenta una forma de implementarlo usando C++.

```
#include <iostream>
#include <set>
#include <vector>

using namespace std;

vector<int> S{2,3,7,9};  // conjunto de números
int C = 9;  // suma deseada

void imprimeConjunto(set<int> s){
  cout << "{";
  for (int i : s)
    cout << i << " ";
  cout << "}\n";
}

void backtracking(set<int> s, int suma, int nivel, int c){
  if (suma > c) return; // se termina la exploración de esa rama
  if (suma == c){       // es una solución
    imprimeConjunto(s);
    return;
  }
  if (nivel < S.size()){  // todavía no se llega al final
    backtracking(s, suma, nivel+1, c);
    s.insert(S[nivel]);
    backtracking(s, suma+S[nivel], nivel+1, c);
  }
}

int main(){
  backtracking({}, 0, 0, C);
}
```

En la línea 7 se define el conjunto S y en la 8 se define la cantidad C que se desea obtener en la suma.

En la línea 10 se define la función imprimeConjunto que recibe un conjunto e imprime sus elementos. Este se utilizará cuando se encuentre una solución.

La función principal backtracking inicia en la línea 17. Se puede observar que verifica exactamente las tres condiciones para hacer *backtracking*:

- La suma sea mayor a C (línea 18)
- La suma es igual a C (línea 19)
- Ya se llegó al último nivel (línea 23). Si no ha llegado al último nivel se hace la recursión en las líneas 24 y 26.

En la línea 24 se hace la recursión sin seleccionar el número de ese nivel (se llama a la función con el mismo conjunto y la misma suma). En la línea 26 se hace la recursión seleccionando al número de ese nivel (se llama con un nuevo conjunto que se forma agregando al conjunto actual, el número del nivel actual, en la línea 25, y con la nueva suma resultante de sumar a la suma actual el número seleccionado).

La llamada inicial se hace con el conjunto vacío, la suma igual a 0, el nivel igual a 0 y la cantidad C deseada (línea 31).

3.5 Ramificación y poda

Ramificación y poda o *Branch and Bound* (*B&B* en inglés) es un algoritmo similar a *backtracking*, donde la idea principal es que en el momento en que se encuentre que una solución parcial no puede ser la solución buscada se detiene el proceso (a esto se le

llama poda). Sin embargo, *B&B* utiliza esta idea para solucionar problemas de optimización.

Un problema de optimización es aquel en que se quiere encontrar la solución que maximice o minimice una función, muchas veces sujeta a ciertas restricciones. Como referencia, un problema de optimización es equivalente a encontrar el máximo o el mínimo de una función continua usando cálculo diferencial, derivadas parciales cuando son una función continua de varias variables y multiplicadores de Lagrange cuando se tienen restricciones, entre otros. *Branch and Bound* sirve para resolver este tipo de problemas, para funciones discretas de varias variables, es decir, para problemas de optimización combinatoria.

En el argot de los problemas de optimización, una **solución factible** es aquella que cumple con todas las **restricciones** que tenga el problema y la **solución óptima** es la solución factible que maximiza o minimiza la función dependiendo de lo que se busque. La función que se quiere maximizar o minimizar se le llama **función objetivo**.

Para hacer *B&B* es necesario que para cada nodo del árbol se pueda establecer un límite (inferior para los problemas de minimización y superior para los problemas de maximización) para el mejor valor de la función objetivo sobre cualquier solución que pueda ser obtenida agregando más variables a la solución parcial que se tenga en el nodo actual. No hay una técnica general para establecer esta cota, es decir, es muy particular para cada problema. Sin embargo, una forma que normalmente resulta adecuada es la de relajar el problema original, es decir, quitar o cambiar algunas de las restricciones de tal forma que tengamos un problema más sencillo de resolver y podamos encontrar una solución adecuada para iniciar el proceso en forma más rápida. Además, como se está tratando un problema de optimización, se

debe ir guardando el mejor valor que se haya encontrado hasta el momento.

En cada nodo se compara el límite de dicho nodo con el mejor valor que se haya obtenido hasta ese momento, el cual al inicio es ∞ si el problema es de minimización y -∞ si el problema es de maximización. Si el límite no es mejor que la mejor solución obtenida hasta este punto entonces el nodo no es promisorio y ahí se detiene la búsqueda con esa rama (se poda la rama), lo que significa que ninguna rama que parte de este nodo nos dará una mejor solución. También se puede podar una rama si la solución parcial en el nodo actual viola alguna de las restricciones establecidas por el problema, es decir, si representa una solución no factible.

Cuando se llega a una solución completa (normalmente en una hoja del árbol, pero puede ser antes de acuerdo con la manera en que se resuelva el problema en cada nodo) se compara con el mejor valor encontrado hasta el momento. Si es mejor se actualiza el mejor valor. Si no, la solución se desecha.

A diferencia de *backtracking*, donde se generaba un solo hijo del nodo a la vez, en B&B sedeben generar todos los hijos del **nodo más prometedor**, a esto se le conoce como ramificación (de ahí el nombre del método), de todos los que se encuentran en la **frontera del árbol** que no han sido podados. La **frontera del árbol** está formada por todos los nodos que no han sido expandidos, es decir, las hojas, hasta ese momento, a los cuales se les llama nodos **vivos**.

Para saber cuál es el nodo más prometedor de entre los nodos vivos, se comparan sus límites y el más prometedor será aquel con el mejor límite.

A la forma de explorar lo nodos que usa *backtracking* se le llama "primero en profundidad", es decir, el siguiente nodo a ser

analizado es el más profundo en el árbol (rompiendo el empate con alguna regla). A la forma de explorar que usa *B&B* se le conoce como "primero el mejor", porque se analiza el nodo que sea el que más promete, el que tenga el mejor límite, es decir, el mejor, al menos hasta ese momento.

El algoritmo general de *B&B* se puede ver en el algoritmo 3.10.

Algoritmo 3.10 *Brach and Bound*

Entrada: la frontera actual del problema formada por los nodos vivos y sus límites. Al inicio la frontera contiene solo la raíz del árbol y su límite. Cada problema tiene un conjunto de variables xi y sus posibles valores V_i, donde $X = \{x_1:V_1, x_2:V_2, \ldots, x_n:V_n\}$.

Salida: la solución óptima o *null* en caso de que la solución no exista.

MEJOR = ∞ si es un problema de minimización o -∞ si es de maximización

SOLUCIÓN = *null*, el nodo solución. Al final se debe imprimir esta SOLUCIÓN.

```
BranchAndBound(frontera): // la frontera contiene todos los nodos vivos
    nodo = mejorNodo(frontera): // el nodo con mejor límite en la frontera
    if not promisorio(nodo):
    // su límite no es mejor que el mejor valor conocido hasta ahora o viola alguna restricción
```

```
        return // hacer poda

    if solución(nodo):  // encontró una solución

        if factible(nodo): // si la solución del nodo es factible

          if límite(nodo) es mejor que MEJOR:

          // su límite es mejor que el mejor valor encontrado hasta
el momento

            MEJOR = límite(nodo)

            SOLUCIÓN = nodo

        return

    // si es un nodo promisorio, hacer ramificación

    for i in hijos(nodo): // si el nodo es una hoja no tiene hijos

        meterEnFrontera(i) // cada hijo del nodo actual lo mete a la
frontera

    if not vacía(frontera): // si la frontera no está vacía

        BranchAndBound(frontera) // llamada recursiva con la
nueva frontera
```

Donde:

- **mejorNodo(frontera):** obtiene el nodo en la frontera con el mejor límite.
- **promisorio(nodo):** regresa *true* si el nodo es promisorio, es decir, si su solución relajada existe y su límite es mejor que el mejor encontrado hasta el momento.
- **solución(nodo):** regresa *true* si nodo es la solución
- **factible(nodo):** regresa *true* si la solución contenida en el nodo es factible
- **hijos(nodos):** regresa el conjunto de todos los hijos de nodo la frontera está vacía.

3.5.1 Ejemplo de funcionamiento

Apliquemos el algoritmo de *B&B* a uno de los problemas más famosos de optimización: el **problema de asignación**, el cual se describe a continuación.

Problema de asignación

Dado un conjunto W de n trabajos, $W = \{w_1, w_2, \ldots, w_n\}$ y un conjunto de T de n trabajadores, $T = \{t_1, t_2, \ldots, t_n\}$, y una matriz C de tamaño (n x n), donde sus filas representan los n trabajadores y sus columnas los n trabajos, la cual contiene los costos cij de asignar el trabajador i al trabajo j encontrar la mejor asignación de trabajos a trabajadores, es decir, la de menor costo total (que se calcula como la suma de los costos individuales), donde cada trabajo solo puede ser asignado a un solo trabajador y a cada trabajador solo se le puede asignar un trabajo.

El nodo raíz debe tener una asignación y su límite, en este caso inferior, la cual se puede obtener relajando el problema original. Las restricciones de este problema dicen que cada trabajo solo se puede ser asignado a un solo trabajador y a cada trabajador solo se le puede asignar un trabajo. Podemos quitar la primera restricción para obtener un problema relajado que permita asignar más de un trabajo a un solo trabajador. Si es así, su solución se obtiene fácilmente seleccionando el costo más pequeño de cada fila. Es seguro que ningún problema que se derive de este nodo tendría un valor más pequeño que este.

Establezcamos el siguiente problema pequeño de asignación para efectos de la explicación del mecanismo *B&B*.

Se tienen 4 trabajadores y 4 trabajos y sus costos de asignación están dados por la siguiente matriz:

$$C = \begin{bmatrix} 5 & 9 & 10 & 3 \\ 2 & 6 & 12 & 1 \\ 7 & 4 & 4 & 8 \\ 11 & 16 & 2 & 14 \end{bmatrix} \quad (3.21)$$

El límite del nodo raíz (nodo 0) se obtiene seleccionando el menor costo de cada fila, es decir, 3+1+4+2 = 10. Se puede observar que esta asignación es solo una solución del problema relajado (sin la restricción de la asignación única) y no es una solución factible para el problema real porque el mismo trabajo 4 se le estaría asignando a más de un trabajador (al 1 y al 2). Por otro lado, esta solución inicial todavía no tiene ninguna asignación real, todas son ficticias y solo se hace para solucionar el problema relajado, es decir, para encontrar el mínimo costo posible de dicho problema relajado. El nodo raíz del árbol queda como el que se muestra en la Figura 3.11, donde, la casilla superior contiene el número de nodo, la del en medio contiene la asignación realizada en ese nivel (por el momento ninguna) y la casilla inferior contiene el límite inferior para ese nodo.

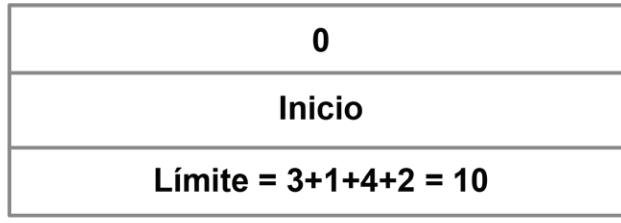


Figura 3.11 Nodo raíz del árbol de asignación

Ahora procedemos a la primera asignación real que se hará sobre el primer trabajador t1, el cual tiene 4 opciones (los 4 trabajos) en las que se expandirá el nodo inicial, por lo que el nivel 1 del árbol contiene esos 4 nuevos nodos, como se muestra en la Figura 3.12.

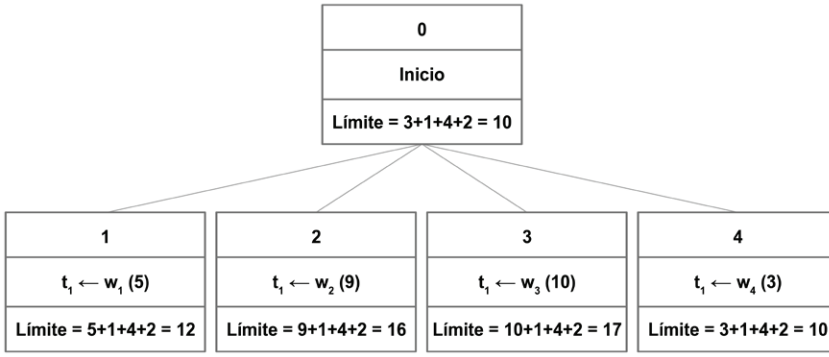


Figura 3.12 Árbol de asignación hasta el primer nivel, donde se asigna trabajo al trabajador 1

Si analizamos el nodo 1 de la Figura 3.12, observamos que corresponde a asignar el trabajo 1 (w_1) (al trabajador 1, porque es el primer nivel). En la matriz 3.21 se puede ver que el costo de dicha asignación es 5, por eso, en la casilla del nodo 1, después de la asignación aparece un 5 entre paréntesis. Esto hace que el límite se mueva, porque ahora hay que sustituir el 3 inicial por el 5, que es una asignación real, dejando el resto de los valores iguales porque corresponden a las otras asignaciones. Así, para el nodo 1 el nuevo límite es 12; para los nodos del 2 al 4 el análisis es el mismo y los valores se pueden observar en la figura 3.13.

Todos los nodos del nivel 1 son nodos vivos, porque todos tienen un límite menor que la mejor solución hasta este momento (ya que la mejor solución al inicio es ∞). Usando la regla primero el mejor, seleccionamos el que tenga el límite más bajo, el cual es

el nodo 4, con un límite de 10. El proceso se sigue de la misma forma para el nodo 4. Sus hijos quedarían en el nivel 2, lo que indica que se tendría que asignar trabajo al trabajador t_2. Ahora, solo quedan 3 trabajos por asignar para este trabajador, por eso el nodo 4 solo tendría 3 hijos. El trabajo w_4 no se puede asignar al trabajador 2, porque para el nodo 4 este trabajo fue asignado al trabajador 1. El resultado de su expansión se observa en la Figura 3.13.

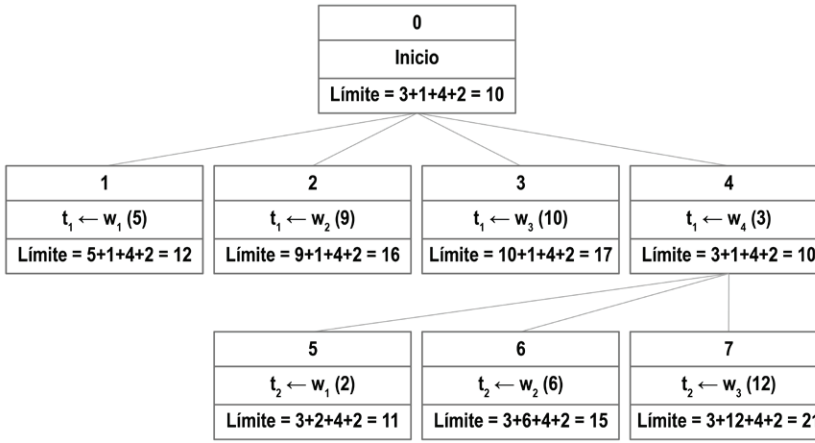


Figura 3.13 Árbol de asignación hasta el nivel dos, donde se asigna trabajo al trabajador 2

Los tres nodos nuevos siguen vivos porque sus límites son menores que la mejor solución encontrada hasta ahora (la cual sigue siendo ∞). De todos los nodos vivos (incluyendo los de todos los niveles) se selecciona el que tenga menor límite, el cual es el 5, con un límite de 11. En el siguiente nivel, el nivel tres, se asigna trabajo al trabajador 3 (t_3). Para esta nueva asignación solo quedan dos trabajos, ya que en esta rama el trabajo w_4 fue asignado al t_1 y el trabajo w_1 fue asignado al trabador t_2. Las opciones restantes para la expansión del nodo 5 se observan en la Figura 3.14.

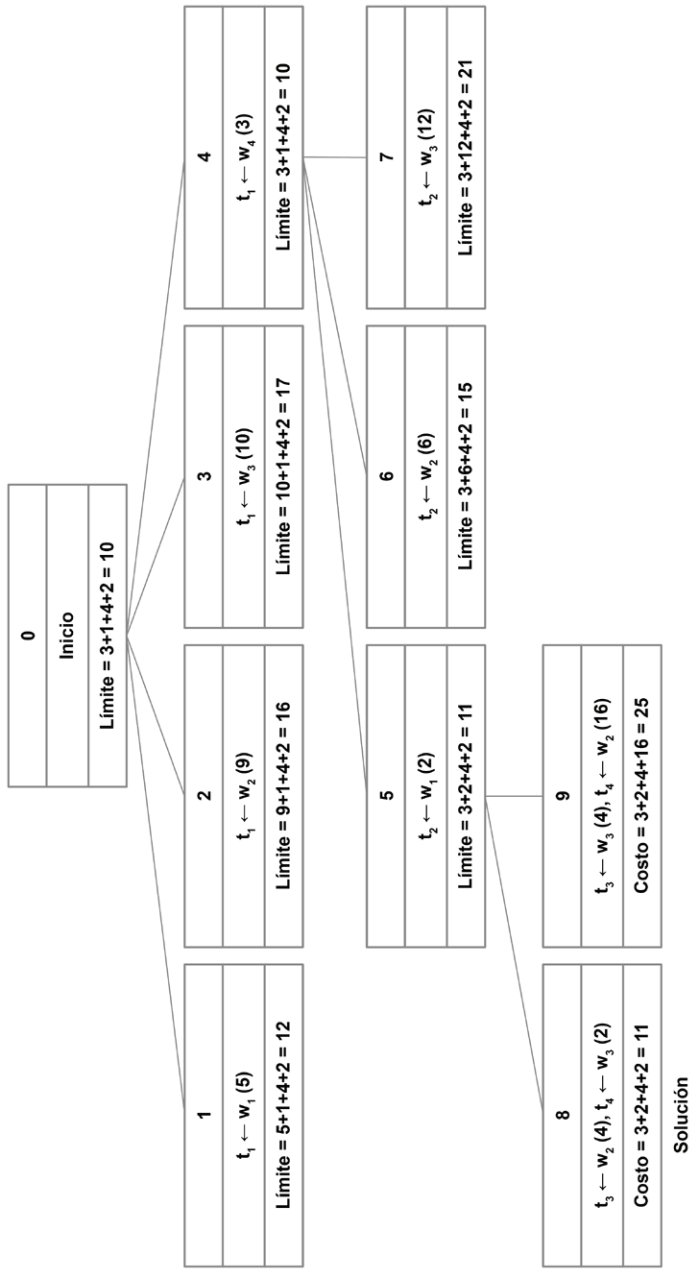


Figura 3.14 Árbol de asignación hasta el nivel 3, donde se asigna trabajo al trabajador 3 y, automáticamente, el trabajo restante es asignado al trabajador 4

En la Figura 3.14, el nivel 3 es especial, ya que al hacer la asignación al trabajador 3, automáticamente el trabajo que queda debe ser asignado al trabajador 4, por esa razón, cada nodo de ese nivel tiene dos asignaciones. Los dos nodos creados siguen vivos porque su límite es menor que el mejor valor encontrado hasta ahora (que sigue siendo ∞). Como en este nivel ya se tienen una asignación completa, el límite ya es el **costo** final de dicha asignación.

Si se continúa con el proceso, el nodo 8 de la Figura 3.14 es una solución y es menor que la mejor encontrada hasta ahora (que es ∞), por lo que es necesario cambiar la mejor solución encontrada a 11, que es el costo del nodo 8. Ahora se analiza el mejor de los nodos vivos restantes, el cual es el nodo 1, sin embargo, su límite es 12, lo que significa que ninguna de las ramas del árbol que partan de él tendrá un costo menor a 12, por lo que esa rama debe ser podada, el nodo ya no está vivo y el proceso continúa. Lo mismo va a pasar con el resto de los nodos porque todos tienen un límite mayor a la mejor solución encontrada hasta ahora. El proceso termina y la solución es la asignación hecha en el nodo 8, es decir [4,1,2,3], lo que significa que al trabajador 1 se le asignó el trabajo 4 (con un costo de 3), al trabajador 2 se le asignó el trabajo 1 (con un costo de 2), al trabajador 3 se le asignó el trabajo 2 (con un costo de 4) y al trabajador 4 se le asignó el trabajo 3 (con un costo de 2), lo que da un costo total final de 11 que es la solución óptima del problema.

El árbol final se observa en la Figura 3.15, donde los nodos podados se marcan con una 'X' roja.

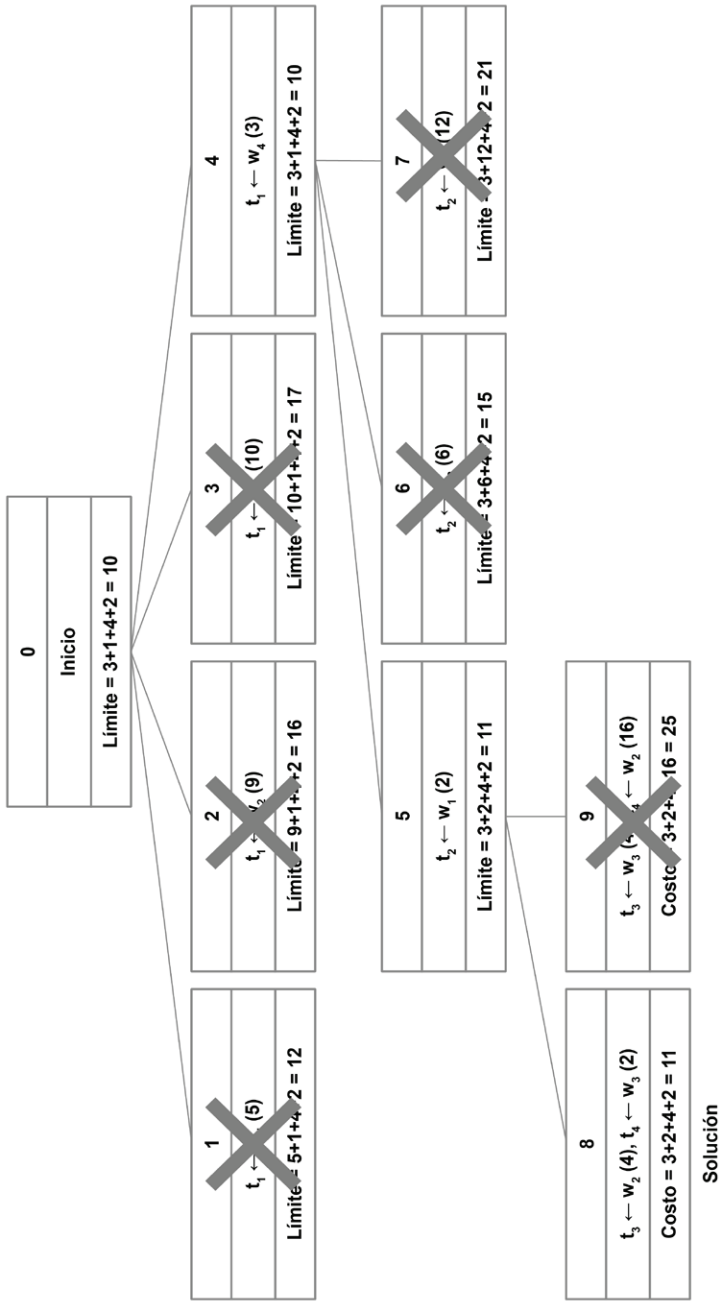


Figura 3.15 Árbol de asignación final

3.5.2 Análisis de complejidad

Como en el caso de *backtracking*, no existe una complejidad general para todos los problemas que se pueden resolver con *B&B*. Se requiere un análisis de la complejidad para cada caso particular. Sin embargo, muchos de los problemas que se resuelven con *B&B* tienen complejidades exponenciales o superiores en el caso de hacerla por medio de búsqueda exhaustiva.

Analizando la complejidad del problema de asignación se puede entender bien este procedimiento y la forma en la que ayuda el uso de *B&B*. Si una solución se representa por un vector de n elementos, donde el primero es el número de trabajo asignado al trabajador 1, el segundo es el número de trabajo asignado al trabajador 2 y así sucesivamente, se puede observar que una solución no es otra cosa más que una permutación de los n trabajos. La forma exhaustiva de resolver el problema implicaría encontrar todas las permutaciones, quitar las que violan alguna restricción, de las que quedan encontrar su costo total y seleccionar la menor. Este trabajo tendría un orden de $O(n!)$.

Al usar *B&B*, en el peor de los casos la complejidad no disminuiría, sin embargo, normalmente, se pueden podar varias ramas y eso disminuye la complejidad promedio del método. La cantidad de la disminución dependerá de varios factores, uno de los principales es la forma de plantear las variables para ir formado el árbol y el establecimiento de las condiciones que sirvan para realizar la poda.

3.5.3 Ejemplo de implementación

Para mostrar una forma de implementar este algoritmo se dará solución al problema de asignación visto en la sección anterior.

```
#include <bits/stdc++.h> // incluye cada librería estándar

using namespace std;

#define N 4 // número de trabajadores y trabajos
int C[N][N] =
        {{5, 9, 10, 3},
   {2, 6, 12, 1},
   {7, 4, 4, 8},
   {11, 16, 2, 14}};

struct Nodo{
  int limite;
  int nivel;
  int asignacion[N];
  bool asignado[N];
};

struct comp{
        bool operator()(const Nodo* a, const Nodo* b) const{
                return a->limite > b->limite;
        }
};

int mejor = INT_MAX; // es equivalente a inf para minimización
Nodo* mejorNodo = nullptr;

int calculaCosto(int asignacion[]){
  int sum=0;
  for (int i=0; i<N; i++)
    sum+=C[i][asignacion[i]];
  return sum;
}

Nodo* nuevoNodo(int nivel, int asignacion[N], bool asignado[N],
                int trabajo){
        Nodo* nodo = new Nodo;

        nodo->nivel = nivel+1;
        for (int i=0; i<N; i++){
                nodo->asignado[i] = asignado[i];
                nodo->asignacion[i] = asignacion[i];
        }
        if (nodo->nivel != 0){ // si no es la raíz
    nodo->asignado[trabajo] = true;
    nodo->asignacion[nodo->nivel-1] = trabajo;
        }
  nodo->limite = calculaCosto(nodo->asignacion);
        return nodo;
}

void imprimeAsignacion(){
  cout << "Costo mínimo: " << mejor << endl;
  for (int i=0; i<N; i++)
    cout << "Trabajador: " << i+1 << " Trabajo: "
         << mejorNodo->asignacion[i]+1 << endl;
}

void BranchAndBound(){
  // crear el min heap
```

```
61   priority_queue<Nodo*, vector<Nodo*>, comp> pq;
62   // calcular el límite del nodo raíz
63   int asignacion[N];
64   bool asignado[N];
65   for (int i=0; i<N; i++){
66     asignado[i] = false;
67     int menor = C[i][0];
68     int trabajo = 0;
69     for (int j=1; j<N; j++)
70       if (C[i][j] < menor){
71         menor = C[i][j];
72         trabajo = j;
73       }
74     asignacion[i] = trabajo;
75   }
76   // crear el nodo inicial (raíz)
77   Nodo* raiz = nuevoNodo(-1, asignacion, asignado, -1);
78   // meter el nodo raíz al heap
79   pq.push(raiz);
80   // inicia el proceso
81   while (!pq.empty()){ // si el heap no está vacío
82     Nodo* menor = pq.top(); // saco el nodo menor
83     pq.pop();
84     if (menor->limite < mejor){ // si el nodo está vivo
85       if (menor->nivel < N){ // faltan trabajos por asignar
86         for (int j=0; j<N; j++)
87           if (!menor->asignado[j]){ // trabajo j no ha sido asignado
88             // se crea un hijo por cada trabajo no asignado
89             Nodo* hijo = nuevoNodo(menor->nivel, menor->asignacion,
90                                    menor->asignado, j);
91             // se mete al heap
92             pq.push(hijo);
93           }
94       }
95       else{ // si ya se asignaron todos los trabajos (nodo hoja) y su
96             //costo es mejor que el mejor
97         mejor = menor->limite;
98         mejorNodo = menor;
99       }
100    }
101  }
102 }
103
104 int main(){
105   BranchAndBound();
106   imprimeAsignacion();
107 }
```

En esta implementación, los trabajos y los trabajadores se numeran del 0 al N-1, por cuestiones de facilidad en el manejo de arreglos de C++. Sin embargo, al imprimir la solución final sí se suma 1 (línea 55) para que corresponda con la explicación de la sección anterior.

La implementación se hace por medio de un árbol cuyo nodo se define en la línea 12. Cada nodo contiene:

- **límite:** es el costo límite de dicho nodo, excepto en el nodo hoja, donde es el costo.

- **nivel:** es el nivel del nodo en el árbol, donde, en el nivel i se asigna trabajo al trabajador i.

- **asignación:** es un arreglo de N enteros en donde la posición i contiene el número de trabajo que se asigna al trabajador i.

- **asignado:** es un arreglo de N valores booleanos en donde, la posición i contiene un *false* cuando el trabajo i no ha sido asignado y *true* cuando ya se asignó.

Para obtener siempre el mínimo en forma eficiente se utiliza una cola con prioridad (*min heap*), definida en la línea 61. Por defecto, al declarar un *heap* en C++ se genera un *max heap*. Para poderlo hacer min heap es necesario redefinir su comparador, lo cual se hace con la estructura de la línea 19, y se usa en la declaración de la línea 12.

El número de trabajadores **N** (y de trabajos) se define como una constante (línea 5) y la matriz de costo se declara como una variable global (líneas 6). Por facilidad se usan costos enteros, pero se pueden cambiar al tipo que se requiera.

El **mejor** valor encontrado se debe inicializar a ∞, lo que se simula inicializándolo con el máximo entero en C++ definido con la constante **INT_MAX** (línea 25). El nodo que contiene la respuesta se va a dejar también en una variable global llamada **mejorNodo** (línea 26).

Se utilizan algunas funciones auxiliares:

- **calculaCosto:** (línea 28) calcula el costo de una asignación dada, la cual recibe como parámetro por medio de un arreglo de **N** enteros.

- **nuevoNodo:** (línea 35) recibe los valores de su nodo padre y calcula los valores que corresponde al nodo hijo, regresándolo como un apuntador a nodo.

- **imprimeAsignacion:** (línea 52) imprime la asignación de costo mínimo (asignación final) y su costo correspondiente. No recibe parámetros porque utiliza las variables globales **mejor** y **mejorNodo**.

La implementación del algoritmo *Branch and Bound* se realiza en la función **BranchAndBound**. Se puede observar que sigue exactamente el Algoritmo 3.10.

Se hace una prueba utilizando la matriz de costos 3.2, usada en el ejemplo de la sección anterior, por lo que el resultado, al correr el programa, debe coincidir con la planteada en dicha sección, es decir, el costo mínimo es 11.

3.6 Ejercicios del capítulo 3

1. Realiza un programa que implemente la **búsqueda binaria** en un arreglo de enteros, usando el enfoque **divide y vencerás** con llamadas recursivas.

2. Calcula la complejidad del algoritmo del problema 1 usando el **método maestro**.

3. Realiza un programa que implemente el ordenamiento *Quick Sort*, usando el enfoque Divide y Vencerás con llamadas recursivas.

4. Calcula la complejidad del algoritmo del problema 3 usando el **método maestro**. Supón que siempre se divide en dos partes iguales.

5. Dado un arreglo que contiene números enteros. Realiza un programa que calcule la máxima suma que se puede obtener si se tiene la restricción de que no se pueden sumar número en casillas consecutivas. Utiliza programación dinámica.

6. Resuelve el problema 5 usando ramificación y poda.

7. Dada una expresión en lógica proposicional. Si se desea saber si existe alguna asignación de valores booleanos que hagan verdadera dicha expresión. ¿Qué enfoque utilizarías para hacer un programa que lo resuelva? Explica las razones principales.

8. La implementación del programa del problema 7 se complica un poco porque se debe evaluar una función booleana general. Afortunadamente, toda expresión booleana se puede escribir en Forma Normal Conjuntiva (FNC), que es la conjunción de disyunciones colocadas entre paréntesis y formadas con variables negadas o no. Por ejemplo, dadas las variables booleanas {a,b,c}, una expresión booleana en FNC sería:

$$(\neg a \vee b) \wedge (b \vee c)$$

Para la cual, en la siguiente tabla se dan dos posibles soluciones (las que hacen que la expresión sea *true*) y una no solución (la que hace que la expresión sea *false*):

a	b	c	$(\neg a \vee b) \wedge (b \vee c)$
False	True	True	True
True	True	False	True
True	False	False	False

Tabla 3.2

Dada una expresión en lógica proposicional en FNC, realiza un programa que regrese todas las asignaciones posibles que hacen verdadera dicha expresión o un letrero indicando que es una "falacia", en caso de que no haya combinación alguna que la haga verdadera. A este problema se le conoce como *Boolean satisfiability problem (SAT)*.

9. Un viajero hace un trayecto en coche de una ciudad A (colocada en el kilómetro 0 del trayecto) a una ciudad B (colocada en el kilómetro F del trayecto). En el camino hay gasolineras en algunos puntos del trayecto (indicadas por los kilómetros g_1, g_2, …, g_n). Si el tanque de gasolina del coche aguanta K kilómetros. Se desea obtener las gasolineras donde el conductor deberá parar a llenar su tanque, de tal forma que se detenga el menor número de veces. ¿Qué enfoque utilizarías para resolver este problema? Explica las razones.

10. Resuelve el problema 9 utilizando el enfoque de algoritmo avaro.

Capítulo 4. Manejo de strings

El manejo eficiente de *strings* es crucial en todas las aplicaciones donde exista texto o algo que se pueda manejar como tal. Este es el caso de la creación de herramientas para un editor de texto, análisis de texto en procesamiento del lenguaje natural, análisis de genomas en biología computacional, manejo de secuencias musicales, entre otras.

Generalmente, el manejo de texto involucra una gran cantidad de información, por lo que de inmediato los científicos de la computación empezaron a pensar en automatizar ciertas tareas del manejo de texto por medio de la computadora. Esto llevó a la necesidad de la creación de ciertos algoritmos que lograran resolver los problemas que se presentaron en la implementación de las tareas en una computadora.

Inicialmente, los editores computacionales de texto fueron una gran motivación porque todo el mundo tiene la necesidad de escribir. Las funciones para facilitar esta edición, tales como, búsqueda de un *substring*, búsqueda y reemplazamiento de palabras, detección de errores ortográficos, etc. fueron tan comunes que requerían hacerse en forma muy eficiente, lo cual motivó el desarrollo de mejores algoritmos.

En años recientes, la biología (biología molecular), al representar el genoma como una secuencia de caracteres llamados bases (A de adenina, C de citosina, G de guanina y T de timina), tuvo la necesidad de realizar el tratamiento y análisis de estas secuencia genómicas, las cuales son en realidad grandes *strings*, por lo que la adopción de los algoritmos para el manejo de *strings* como una de sus herramientas fundamentales se dio en forma muy natural, logrando así, con la ayuda de la computadora, tener un rápido desarrollo dando lugar a los que se conoce como bio-

logía computacional, biología molecular computacional o bioinformática. La aplicación y desarrollo en esta área fue tan grande que, en la actualidad, es muy común encontrar varios capítulos que tratan sobre algoritmos para el manejo de *string* en libros relacionados con la biología computacional.

Los algoritmos para el manejo de *string* forman una clase muy especial y por eso los estudiaremos en una sección por separado complementando los enfoques analizados en el capítulo 3.

4.1 Notación especial para *strings*

Un *string* **S** está formado por caracteres colocados uno detrás del otro, es decir, es una secuencia ordenada de caracteres. El primer carácter en el *string* es el de más a la izquierda y ocupa la posición 1, el siguiente la posición 2 y así sucesivamente. Esta es la convención que usaremos en las definiciones. Sin embargo, se debe tener cuidado en las implementaciones que se presentarán en este capítulo debido a que la posición inicial de un *string* será la 0 para estar de acuerdo en la forma de tratar los arreglos en C++, que es el lenguaje que se utilizará en la implementación de los algoritmos.

De acuerdo a lo anterior, un *string* en computación se puede ver como un arreglo de caracteres y la mayor parte de los lenguajes de programación los tratan así.

La **longitud** de un *string* **S** está dada por el número de caracteres que contiene y se representa como **|S|**.

Si la longitud del *string* **S** es **n**, sus caracteres van a estar en las posiciones 1 a la **n** del *string*, iniciando en la extrema izquierda y es equivalente a decir que **S = S[1..n]**. Por ejemplo, si **S = abc**, su longitud es 3 y el carácter que está en la posición 1 es la **a**, en la posición 2 es la **b** y en la posición 3 (igual a su longitud) es la **c**.

Un ***substring*** del *string* **S** es cualquier secuencia de caracteres que se encuentre en el *string* S. El *substring* S**[i..j]** es el *string* formado con los caracteres de **S** que están de la posición **i** a la **j**, inclusive. Por ejemplo: en el *string* **S = abcaabbcc**, el *substring* **S[3..6] = *caab*** y el *substring* **S[5..8] = abbc**. Si **i >j**, **S[i..j]** representa el ***string nulo***, es decir, el *string* formado por 0 caracteres.

Dos definiciones muy importantes al tratar *strings* son las de **prefijo** y **sufijo** que se muestran a continuación:

Prefijo

Dado un *string* **S**, un prefijo del mismo es cualquier *substring* **S[1..i]**, es decir, es un *substring* que inicia en la posición **1** y termina en la posición **i** de **S**.

Por ejemplo: si el *string* **S = abcde**, sus prefijos son: **a, ab, abc, abcd** y **abcde**.

Sufijo

Dado un *string* **S**, con **|S|=n**, un sufijo del mismo es cualquier *substring* **S[i..n]**, es decir, es un *substring* que inicia en la posición **i** y termina en la posición **n**, que es el final **S**.

Por ejemplo, si el *string* **S=abcde**, sus sufijos son: **e, de, cde, bcde** y **abcde**.

Para un *string* **S**, se dice que un *substring*, prefijo o sufijo es **propio** si no es iguales a **S**.

4.2 Función Z (Z-function)

La función Z se define sobre un *string* de la siguiente forma:

Función Z

Dado un *string* **S** de longitud **n**, la función Z es un arreglo de longitud **n** para el cual, la celda en la posición **i>1** contiene un número que indica el máximo número de caracteres que a partir de la posición **i** del *string* **S** coinciden con los caracteres del inicio del mismo *string* **S**, es decir, el valor en la celda **i** es la longitud máxima del prefijo (el prefijo más largo) del *substring* **S[i..n]** que es también un prefijo del *string* completo **S**.

De la definición anterior es claro que la función Z solo se calcula para los caracteres que no son el primero del *string*. El desarrollo que se realiza a continuación se colocará un valor de 0 al inicio porque esa posición no es parte de la definición.

Por ejemplo, si se tiene el *string* **S = aaabaaab** su función Z estará definida como en la Figura 4.1.

string	**a**	**a**	**a**	**b**	**a**	**a**	**a**	**b**
posición	1	2	3	4	5	6	7	8
Valores Z	**0**	**2**	**1**	**0**	**4**	**2**	**1**	**0**

Figura 4.1 Función Z para el string **S = aaabaaab**. El número arriba de la celda es la posición y el número en la celda **i** es el valor de la función Z para la posición **i**

Para la función **Z** en la posición **2**, el *string* a partir de ahí sería **aabaaab** y coincide con el inicio del *string* original **aaabaaab** solo en las dos primeras letras **a**, por lo que su función Z es 2. En la posición 5, el *string* a partir de ahí es **aaab**, el cual coincide en 4 letras a partir del inicio del *string* original, por lo que su función Z es 4.

4.2.1 Cálculo de la función Z

La forma más simple de calcular la función Z es utilizar fuerza bruta. Para cada posición i del *string* se van contando los caracteres que coinciden. El algoritmo es el mostrado en el Algoritmo 4.1.

Algoritmo 4.1 Cálculo de la función Z de un *string*

Entrada: el *string* S al que se le quiere calcular su función Z.

Salida: el arreglo con los valores de la función Z para el *string* S.

```
1. Crear arreglo A e inicializarlo a 0.
2. For i=1 to S.length:
3.         For j=i to S.length:
4.           Si S[j] = S[j-i], hacer A[i] = A[i] + 1
5.           Si no, salir del For
6. Regresa A
```

La implementación en C++ es totalmente directa a partir del algoritmo, solo debemos recordar que la posición inicial del arreglo en C++ es 0, por lo que el arreglo que nos regresa el valor que se encuentra en la celda 0 es equivalente al de la celda 1 de la definición. Un posible código se muestra a continuación.

```
1   vector<int> ZNaive(string S){
2     int n = S.length();
3     vector<int> A(n,0);
4     for (int i=1; i<n; i++)
5       for (int j=i; j<n; j++)
6         if (S[j] == S[j-i])
7           A[i]++;
8         else
9           break;
10    return A;
11  }
```

Este algoritmo tiene una complejidad $O(n^2)$. Una forma de mejorar esta implementación es recordar las coincidencias que se han tenido hasta i-1, con lo que se puede obtener una complejidad $O(n)$. Esto sería algo similar a la memoization de la programación dinámica. Dejamos al lector la implementación de este algoritmo.

4.3 Problema de la coincidencia de un patrón

El problema de la coincidencia de un patrón en un texto (*Pattern Matching Problem* en inglés), también conocido como búsqueda de un *substring* en un texto (*Substring Search* en inglés) se define de la siguiente forma:

Problema de la coincidencia de un patrón (*Pattern Matching Problem*)

Dado un *string* **P**, llamado el patrón (*pattern*) y un *string* más grande **T**, llamado el texto, encontrar la primera ocurrencia del patrón P en el texto T y regresar la posición del texto T donde inicia el patrón P o -1 en caso de que el patrón P no exista en el texto T.

Por ejemplo: si **P = aca** y **T = bacacabcaca**, el patrón **P** se encuentra en **T** iniciando en las posiciones 2, 4 y 9. Observa que las ocurrencias pueden estar sobrepuestas (*overlap*) tal como sucede en las ocurrencias que inician en 2 y 4. Para este caso, la solución al problema de la coincidencia de un patrón será 2, que es la posición de inicio de la primera ocurrencia de P en T.

La forma más simple de resolver este problema es por fuerza bruta, llamada algoritmo *naive*, el cual consiste en ir moviendo a la derecha (*shifting*) una ventana que contiene el patrón (conocida como ventana de deslizamiento, *sliding window*) casilla por casilla sobre el *string* T y en cuenta se encuentra una coincidencia se regresa el número de la casilla de inicio. El algoritmo se puede ver en el Algoritmo 4.2.

Algoritmo 4.2 Algoritmo naive del problema de la coincidencia exacta

Entrada: el *string* patrón P de longitud m y el *string* texto T de longitud n.

Salida: la posición en T del inicio de la primera ocurrencia del patrón P o -1 si no existe P en T.

1. For i=1 to T.length − P.length:
2. Si el *string* T[i..i+m] es igual a P
3. Regresar i
4. Regresa -1

Una forma de implementar este algoritmo se muestra a continuación usando C++.

```
1   int PenTnaive(string P, string T){
2     int n = T.length();
3     int m = P.length();
4     for (int i=0; i<n-m+1; i++){
5       bool bandera = true;
6       for (int j=0; j<m; j++)
7         if (P[j] != T[i+j]){
8           bandera = false;
9           break;
10        }
11      if (bandera)
12        return i;
13    }
14    return -1;
15  }
```

Recordemos que se usan arreglos que inician en la posición 0, por lo que en la línea 4 se debe sumar 1 a la resta.

Se usa una bandera booleana que inicia en ***true*** y si en algún momento se rompe la coincidencia (línea 7) hace la bandera ***false*** y se sale del ciclo (línea 9). Si al terminar el ciclo la bandera es ***true*** se regresa la posición i (línea 12) y el proceso termina. Si termina el ciclo y no se reconoció el patrón, regresa -1 (línea 14).

La complejidad de este algoritmo es *O(mn)*. Existen varias formas de mejorar este algoritmo para hacerlo en tiempo lineal. Una de ellas es el uso de la función Z calculada en tiempo lineal, que logra una complejidad también lineal sobre la longitud del texto, es decir, *O(n)*. Para lograrlo, basta concatenar el patrón al inicio del texto, separando caracteres por un carácter que no esté en el texto, normalmente, para efectos de una explicación teórica se usa el signo de pesos ($).

Por ejemplo, si **P = aca** y **T = bacacabcaca** hacemos un nuevo *string* **T'** formado con la concatenación de P y T separados por un $, es decir, **T'** = P$T = **aca$bacacabcaca**. Ahora, calculamos su función Z con alguno de los algoritmos vistos anteriormente y obtenemos el arreglo mostrado en la Figura 4.2.

string	a	c	a	$	b	a	c	a	c	a	b	c	a	c	a
posición	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Valores Z	9	3	1	8	4	5	2	7	9	3	1	8	4	5	2

Figura 4.2 Función Z para el string **aca$bacacabcaca**

Después de este proceso, las ocurrencias estarán indicadas en el arreglo de la función Z por las posiciones que tengan un valor de Z igual a la longitud del patrón y restándole la longitud del patrón más 1 (por el $). En este caso, como la longitud del patrón

es 3, las ocurrencias están en las posiciones **6 − 4 = 2, 8 − 4 = 4 y 13 − 4 = 9** del *string* **T** original. La solución estaría dada por la primera encontrada, es decir 2.

El algoritmo KMP, también resuelve este problema en tiempo lineal sobre la longitud del texto y, además, tiene varias ventajas en algunas aplicaciones en donde se requiere resolver este problema como parte de la solución del problema mayor, por ejemplo, cuando se busca un conjunto de patrones en un texto T.

4.4 Algoritmo Knuth-Morris-Pratt (KMP)

Una mejor forma de resolver el problema de la coincidencia de un patrón es utilizando el famoso algoritmo KPM (por las iniciales de los nombres de los autores), el cual fue propuesto por Donald Knuth y en forma independiente por James Morris y Vaughan Pratt en 1970. Los tres juntos publicaron el algoritmo en 1977.

Recordemos que la forma *naive* de resolver este problema requiere que el patrón se vaya probando como una ventana que se recorre en el texto un carácter a la vez. La idea principal del algoritmo KPM, para aumentar su velocidad, es que, cuando se tenga una no coincidencia, no se tenga que recorrer solo un carácter y volver a empezar, sino que se pueden recorrer tantos caracteres como se garantice que el patrón no estará ahí.

La idea básica es:

- Se van comparando los caracteres a partir de la posición **i** del texto con los del patrón.
- Cuando uno no coincide, en el *substring* del patrón que se tiene antes de la no coincidencia, buscamos el máximo prefijo que al mismo tiempo sea sufijo de dicho *substring*.

- Si no existe, no nos tenemos que regresar ni un carácter en el texto, solo nos regresamos en el patrón a su posición inicial.

- Si existe un prefijo que es sufijo y la longitud de este prefijo es k, continuamos a partir de la posición i del texto, pero no tenemos que regresar a la posición 1 del patrón, sino a la posición 1+k.

- Este proceso se hace tan hacia atrás como sea necesario pudiendo llegar a iniciar el patrón en su posición 1.

Por ejemplo, si parte del texto es **T = abcabx**… y el patrón es **P = abcaby**, iniciamos comparando los caracteres uno a uno y todos van siendo iguales hasta que llegamos a la posición 6, en donde falla porque tenemos una **x** en el texto y una **y** en el patrón. En el método naive tendríamos que iniciar todo desde la posición 2 del texto y la 1 del patrón, pero podemos observar que en el *substring* anterior a la falla, es decir, en **abcab**, existe el prefijo **ab**, con longitud 2, que al mismo tiempo es sufijo del mismo *string* (**ab**c**ab**), lo que nos dice que es seguro que los 2 caracteres anteriores al fallo coinciden completamente con el inicio del patrón, por lo que las siguientes comparaciones se pueden hacer a partir de la 6 del texto y la posición 3 del patrón.

Este comportamiento en el algoritmo ahorra muchas comparaciones, pero para lograrlo es necesario hacer un preprocesamiento del patrón.

4.4.1 Preprocesamiento del patrón

Este preprocesamiento se hace solo sobre el patrón usando un arreglo que inicia en la posición 0, que sería la posición inicial del *string*. Al final del preprocesamiento, el arreglo contendrá el

número de caracteres que se debe regresar en el patrón para seguir con la comparación con el texto. El Algoritmo 4.3 nos muestra este preprocesamiento:

Algoritmo 4.3 Algoritmo para el preprocesamiento del patrón

Entrada: el *string* patrón **P** de longitud **m**.

Salida: un arreglo en donde la celda en la posición i (**0 < i < m-1**) contiene la longitud del máximo prefijo que es también sufijo en el *string* **P[1..i].**

1. Crear un arreglo **V** de **n** posiciones, iniciando en la posición 0 (0-arreglo).

2. Inicializar **j = 0, i = 1 y V[0] = 0**.

3. Mientras **i** sea menor que **n**: (solo llega hasta n-1)

4. Si **P[i] == P[j]:**

5. Hacer **V[i] = j + 1**

6. **i = i + 1** y **j = j + 1**

7. Si no, (los caracteres son diferentes)

8. Si **j == 0:**

9. **V[i] = 0**

10. Hacer **i = i + 1**

11. Si no, (**j** es diferente de 0):

12. Hacer **j = V[j-1]**

Este algoritmo tiene una complejidad proporcional a la longitud del patrón, esto es, *O(m).*

Sigámoslo con el ejemplo mostrado en la Figura 4.3 para que sea más claro.

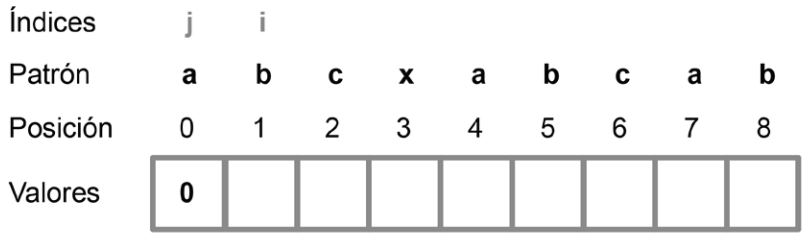


Figura 4.3 Patrón inicial ejemplo para preprocesamiento

La Figura 4.3 muestra la inicialización para el preprocesamiento del patrón. El valor de la posición 0 es siempre 0 y se colocan dos índices: la j en la posición 0 y la i en la posición 1. Los pasos son los siguientes:

1. Como el carácter en la posición **i** (posición 1, es una **b**) no es igual al de la posición **j** (posición 0, es una **a**) se coloca un 0 en el valor de la posición **i** (posición 1). Como **j** está en 0 solo incrementa **i**, pasando a la posición 2. Figura 4.4b.

2. Los caracteres en las posiciones **i** (posición 2, es una **c**) e **j** (posición 0, es una **a**) vuelven a no ser iguales entonces se procede de la misma forma, colocando un 0 en el valor de la posición **i** (posición 2) y, como **j** es 0, solo se incrementa **i**, pasando a la posición 3. Figura 4.4c.

3. Una vez más, los caracteres en **i** (posición 3, es una **x**) y **j** (posición 0, es una **a**) no son iguales, se coloca un 0 en la posición 3 y, como **j** es 0, solo se incrementa **i**, pasando a la posición 4. Figura 4.4d.

4. Ahora los caracteres en las posiciones **i** (posición 4) y **j** (posición 0) son iguales (ambos son **a**), por lo que en el valor de la posición **i** (posición 4) se coloca la posición de **j** (0) más 1, es decir, 1. Observe que se le suma 1 a la posición de **j** no al valor que están en la posición **j**. Como sí coincidieron, se incrementan ambos: **i** y **j**, pasando i a la posición 1 y **j** a la posición 5. Figura 4.4e.

El 1 en la posición 4 significa que en el *string* anterior de la posición 1 a la 4 del patrón, esto es, el *string* **abcxa**, existe un *substring* de 1 solo carácter que es prefijo y sufijo al mismo tiempo. En este caso se trata del *substring* **a** de un solo carácter.

5. Ahora el carácter en **i** es igual al carácter en **j** (ambos son **b**), por lo que el valor en la casilla 5 es la posición de **j** (posición 1) más 1, esto es, 2. Como son iguales se incrementan los dos índices pasando **j** a la posición 2 e **i** a la posición 6. Figura 4.4f.

El 2 en la posición 5 significa que en el *string* anterior de la posición 1 a la 5 del patrón, esto es, el *string* **abcxab**, existe un *substring* de 2 caracteres que es prefijo y sufijo al mismo tiempo. En este caso se trata del *substring* **ab** de dos caracteres.

6. Los caracteres en las posiciones **i** (posición 2) y **j** (posición 6) son iguales (son una **c**), por lo que en el valor de la posición 6 se coloca la posición del índice **j** (posición 2) más 1, es decir, 3. Como son iguales se incrementan los dos índices pasando **j** a 3 e **i** a 7. Figura 4.4g.

El 3 en la posición 6 significa que en el *string* anterior, de la posición 1 a la 6 del patrón, esto es, el *string* **abcxabc**, existe un *substring* de 3 caracteres que es prefijo y sufijo al mismo tiempo. En este caso se trata del *substring* **abc** de tres caracteres.

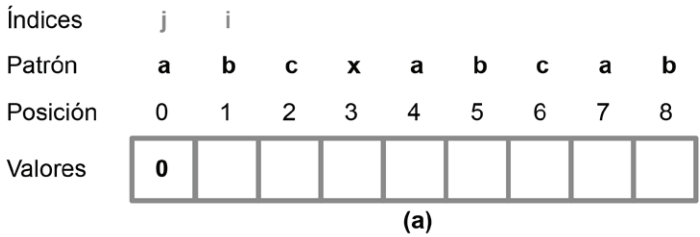
7. Los caracteres en las posiciones **j** (posición 3, es una **x**) e **i** (posición 7, es una **a**) no son iguales. Como **j** no es cero se tiene que mover a la posición indicada por el valor en la posición **j-1**. Como **j** es 3, j-1 será 2, y el valor en la posición 2 es 0, es decir, **j** se debe mover a la posición 0. No se incrementa **i**, que sigue en la posición 7. Figura 4.4h.

Observa que, todavía no se coloca valor en la posición **i** (posición 7).

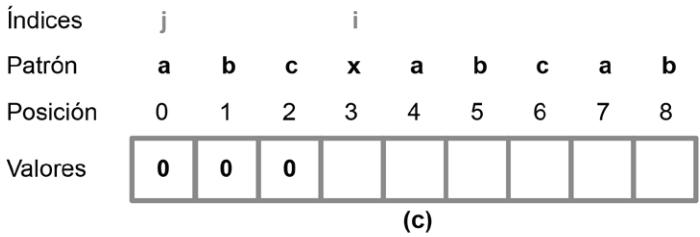
8. Los caracteres en las posiciones **i** (posición 7) y **j** (posición 0) son iguales (ambos son **a**), por lo que se coloca como valor de la posición 7, la posición de **j** (posición 0) más 1, esto es 1. Se incrementan los dos índices pasando **i** a 8 y **j** a 1. Figura 4.4i.

9. Los caracteres en las posiciones **i** (posición 8) y **j** (posición 1) son iguales (ambos son **b**). Se coloca como valor de la posición 8, la posición **j** (posición 1) más 1, esto es 2. Como ya **i** llegó al último carácter del *string*, el proceso termina. Figura 4.4j.

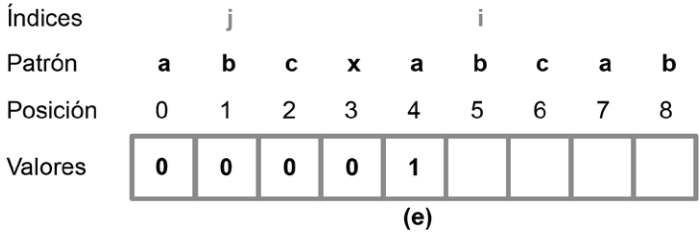
La Figura 4.4 contiene gráficamente todos los pasos anteriores.



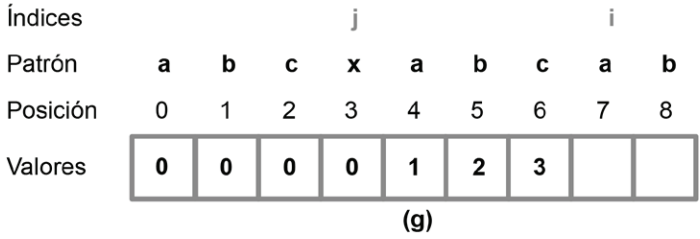
(a)



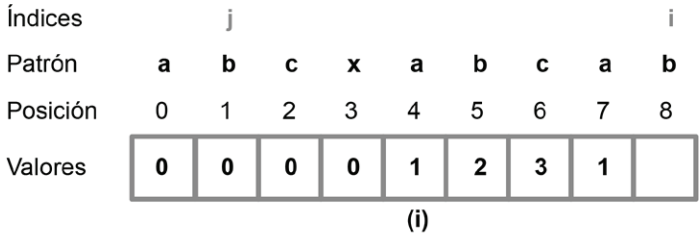
(c)



(e)



(g)



(i)

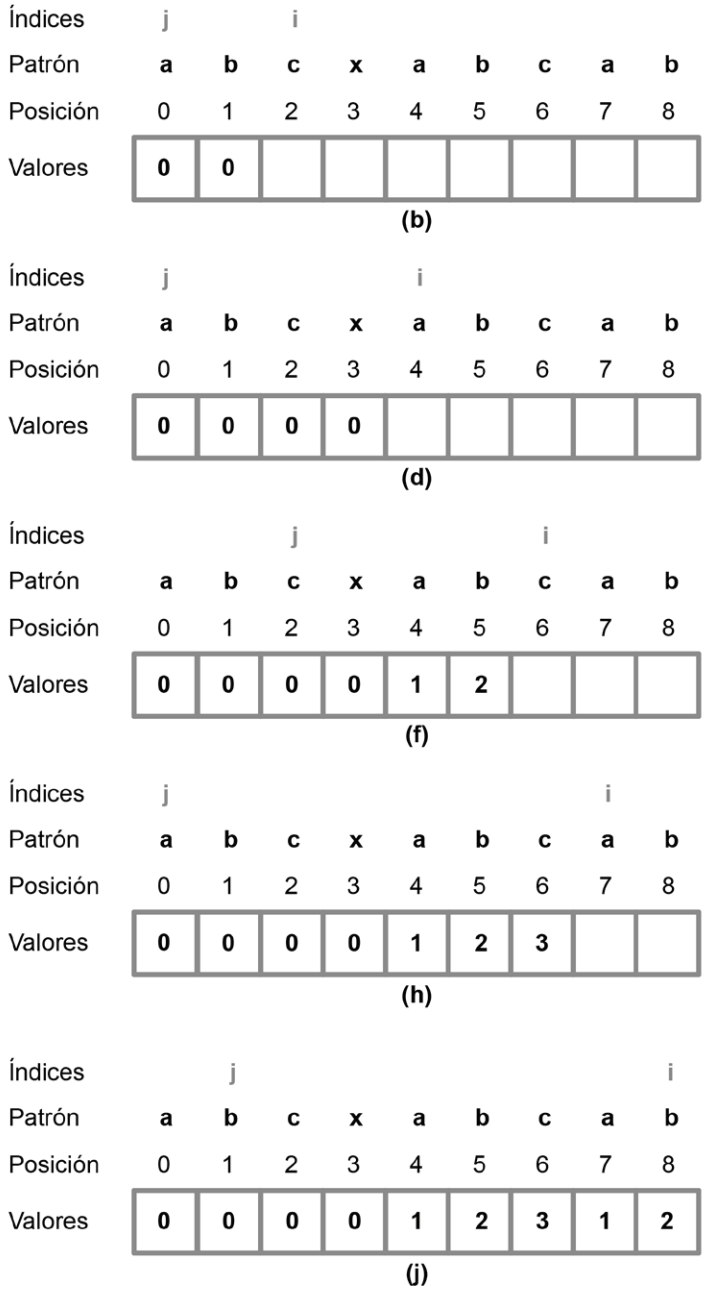


Figura 4.4 Ejemplo completo del uso del Algoritmo 4.3

A continuación se presenta una implementación de este algoritmo, la cual se hizo prácticamente en forma directa del Algoritmo 4.3.

```
 1 vector<int> preProcesoPatron(string P){
 2   int m = P.length();
 3   vector<int> V(m);
 4   int i=1, j=0;
 5
 6   V[0] = 0;
 7   while (i < m){
 8     if (P[i] == P[j]){
 9       V[i] = j + 1;
10       i++; j++;
11     } else
12         if (j == 0){
13           V[i] = 0;
14           i++;
15         }
16         else
17           j = V[j-1];
18   }
19   return V;
20 }
```

4.4.2 Algoritmo KMP completo

Una vez preprocesado el patrón P a ser buscado en un texto T, el algoritmo es muy simple, ya que al arreglo V obtenido del preprocesamiento de P nos dice a qué posición se debe regresar en el patrón para seguir comparando con el texto, una vez que encuentre un fallo en la comparación, es decir, que dos caracteres que se están comparando no sean iguales. Veamos el Algoritmo 4.4.

Algoritmo 4.4 Algoritmo KMP

Entrada: el *string* patrón **P** de longitud **m**, el *string* texto **T** de longitud **n**.

Salida: la posición en T del inicio de la primera ocurrencia del patrón P o -1 si no existe P en T.

1. Hacer el arreglo V igual al resultado del preprocesamiento de P.

2. Hacer **posición = 0** (indica la posición donde inicia P en T, cuando se encuentre).

3. Hacer **j = 0** (índice en el patrón P) e **i = 0** (índice del texto T)

4. Mientras **i < n:**

5. if T[i] == P[j]:

6. j = j +1

7. i = i + 1

8. if j == m (ya terminó el patrón)

9. regresar posición (se encontró y se termina el proceso)

10. else:

11. if j == 0:

12. i = i + 1

13. posicion = i (actualizamos la posible posición de inicio)

14. else:

15. posición = i – V[j – 1] (actualizamos la posible posición de inicio)

16. j = V[j – 1] (se regresa j al valor indicado en la posición j-1)

17. regresar -1 (no se encontró)

La complejidad de este algoritmo es lineal sobre la longitud del texto, esto es: *O(n)* y como la complejidad del preprocesamiento es lineal sobre la longitud del patrón, esto es: *O(m)*, la complejidad de KMP es lineal, sobre los dos procesos, es decir: *O(m+n)*, mejor que *O(mn)* del algoritmo *naive*.

Veamos trabajar el algoritmo con el ejemplo mostrado en la Figura 4.5, donde **P = abcabc** y **T = xabcabxabcabcx**. Se puede observar que el patrón P ya fue preprocesado y sus valores están dados en el arreglo de color naranja. Las posiciones iniciales de ambos *strings* están en 0. Se inicializan los índices **i** (índice del texto) y **j** (índice del patrón) a 0.

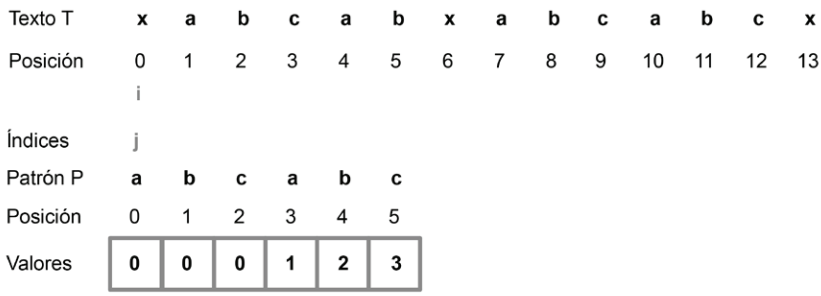


Figura 4.5 Inicialización de ejemplo de aplicación de algoritmo KMP

Los pasos para resolver el problema del ejemplo de la Figura 4.5 son los siguientes:

1. Los caracteres en las posiciones i (posición 0 de T, es una x) y j (posición 0 de P, es una a) no son iguales. Como la j = 0, solo se incrementa i pasando a la posición 1 y se actualiza la posible posición de inicio al valor de i, esto es, posición = 1. Figura 4.6b.

2. Los caracteres en las posiciones i (posición 1 de T, es una a) y j (posición 0 de P, es una a) son iguales. Se incrementan tan-

to i como j, pasando a las posiciones 2 y 1, respectivamente. Figura 4.6c.

3. Los caracteres en las posiciones i (posición 2 de T, es una b) y j (posición 1 de P, es una b) son iguales. Se incrementan tanto i como j, pasando a las posiciones 3 y 2, respectivamente.

4. Los caracteres en las posiciones i (posición 3 de T, es una c) y j (posición 2 de P, es una c) son iguales. Se incrementan tanto i como j, pasando a las posiciones 4 y 3, respectivamente.

5. Los caracteres en las posiciones i (posición 4 de T, es una a) y j (posición 3 de P, es una a) son iguales. Se incrementan tanto i como j, pasando a las posiciones 5 y 4, respectivamente.

6. Los caracteres en las posiciones i (posición 5 de T, es una b) y j (posición 4 de P, es una b) son iguales. Se incrementan tanto i como j, pasando a las posiciones 6 y 5, respectivamente. Figura 4.6d.

7. Los caracteres en las posiciones i (posición 6 de T, es una x) y j (posición 5 de P, es una c) no son iguales. Como j no es 0, se actualiza la posible posición a i − V[j-1], esto es 6 − 2 = 4, y se actualiza j a V[j-1], esto es, j = 2. Figura 4.6e.

8. Los caracteres en las posiciones i (posición 6 de T, es una x) y j (posición 2 de P, es una c) no son iguales. Como j no es 0, se actualiza la posible posición a i − V[j-1], esto es 6 − 0 = 6, y se actualiza j a V[j-1], esto es, j = 0. Figura 4.6f.

9. Los caracteres en las posiciones i (posición 6 de T, es una x) y j (posición 0 de P, es una a) no son iguales. Como la j = 0, solo se incrementa i pasando a la posición 7 y se actualiza la posible posición de inicio al valor de i, esto es, posición = 7. Figura 4.6g.

10. Los caracteres en las posiciones i (posición 7 de T, es una a) y j (posición 0 de P, es una a) son iguales. Se incrementan tanto i como j, pasando a las posiciones 8 y 1, respectivamente.

11. Los caracteres en las posiciones i (posición 8 de T, es una b) y j (posición 1 de P, es una b) son iguales. Se incrementan tanto i como j, pasando a las posiciones 9 y 2, respectivamente.

12. Los caracteres en las posiciones i (posición 9 de T, es una c) y j (posición 2 de P, es una c) son iguales. Se incrementan tanto i como j, pasando a las posiciones 10 y 3, respectivamente.

13. Los caracteres en las posiciones i (posición 10 de T, es una a) y j (posición 3 de P, es una a) son iguales. Se incrementan tanto i como j, pasando a las posiciones 11 y 4, respectivamente.

14. Los caracteres en las posiciones i (posición 11 de T, es una b) y j (posición 4 de P, es una b) son iguales. Se incrementan tanto i como j, pasando a las posiciones 12 y 5, respectivamente. Como j = 5, que es la última posición del patrón, significa que el patrón se encontró por completo en T. Se termina el proceso regresando la posición, que hasta este momento es 7, de acuerdo con la última actualización en el paso 9, lo que significa que la posición de inicio donde se encontró el patrón es 7. Figura 4.6h.

Algunos de los pasos anteriores (los repetitivos, donde coinciden los caracteres y se incrementan tanto i como j, fueron omitidos) del proceso de aplicación del algoritmo KPM se pueden observar gráficamente en la Figura 4.6.

Texto T	x	a	b	c	a	b	x	a	b	c	a	b	c	x
Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	i													
Índices	j							Posición = 0						
Patrón P	a	b	c	a	b	c								
Posición	0	1	2	3	4	5								
Valores	0	0	0	1	2	3								

(a)

Texto T	x	a	b	c	a	b	x	a	b	c	a	b	c	x
Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13
			i											
Índices		j						Posición = 1						
Patrón P	a	b	c	a	b	c								
Posición	0	1	2	3	4	5								
Valores	0	0	0	1	2	3								

(c)

Texto T	x	a	b	c	a	b	x	a	b	c	a	b	c	x
Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13
							i							
Índices			j					Posición = 6 - 2 = 4						
Patrón P	a	b	c	a	b	c								
Posición	0	1	2	3	4	5								
Valores	0	0	0	1	2	3								

(e)

Texto T	x	a	b	c	a	b	x	a	b	c	a	b	c	x
Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13
								i						
Índices	j							Posición = 7						
Patrón P	a	b	c	a	b	c								
Posición	0	1	2	3	4	5								
Valores	0	0	0	1	2	3								

(g)

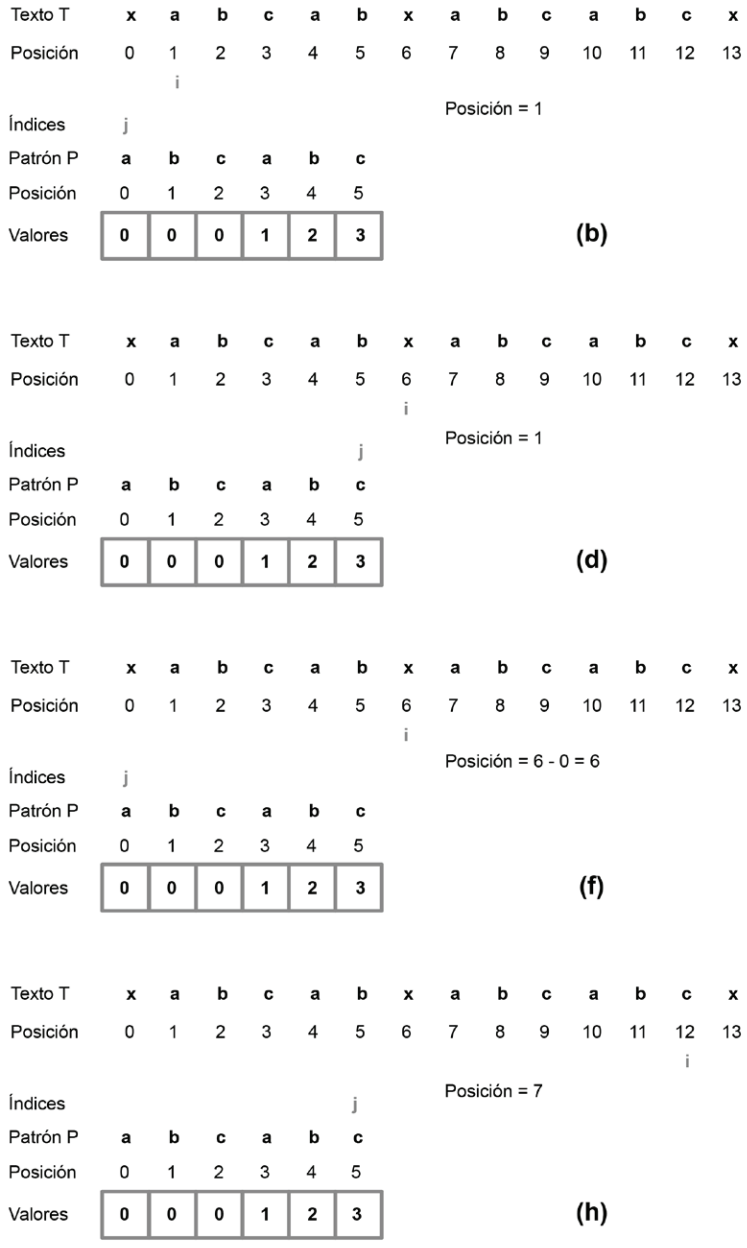


Figura 4.6 Ejemplo del proceso de aplicación del algoritmo KMP

A continuación se muestra una posible implementación del Algoritmo 4.4.

```
int KMP(string P, string T){
  int m = P.length(), n = T.length();
  vector<int> V = preProcesoPatron(P);
  int posicion = 0, i = 0, j = 0;
  while (i < n){
    if (T[i] == P[j]){
      i++;
      j++;
      if (j == m)
        return posicion;
    }
    else
      if (j == 0){
        i++;
        posicion = i;
      }
      else{
        posicion = i - V[j-1];
        j = V[j-1];
      }
  }
  return -1;
}
```

4.5 El problema del palíndromo más largo

Un **palíndromo** es un *string* que se lee igual de izquierda a derecha que de derecha a izquierda, por ejemplo, el *string* **aabcbaa** es un palíndromo. En algunas aplicaciones es necesario encontrar los *substrings* que son palíndromos dentro de un *string* dado o el palíndromo más grande que exista como *substring* de un *string* dado. A este último problema se le conoce como el problema del palíndromo más largo y se define a continuación.

Problema del palíndromo más largo

Dado un *string* **S**, se desea encontrar el *substring* del mismo que forma un palíndromo y que es el más largo que se pueda encontrar, es decir, el que está formado con el mayor número de caracteres.

Revisemos dos formas de resolverlo: primero la forma *naive*, usando fuerza bruta y, posteriormente, una forma más eficiente realizada con el algoritmo de Manacher.

4.5.1 Algoritmo naive para el palíndromo más largo

La forma *naive* de resolver el problema del palíndromo más largo que *substring* de un *string* **S** dado es usando fuerza bruta, en donde se recorren todas las posiciones **i** del *string*, para cada una de ellas se trata de expandir hacia la izquierda y a la derecha lo más que se pueda hasta encontrar el *string* más grande, lo cual se debe hacer tanto para *strings* de longitud impar como para los de longitud par. Este algoritmo tiene una complejidad $O(n^2)$, donde **n** es la longitud del *string* y se presenta en el Algoritmo 4.5.

Algoritmo 4.5 Palíndromo más largo usando fuerza bruta

Entrada: el *string* **S** de longitud **n**, donde se desea encontrar el palíndromo más largo.

Salida: una pareja **(m,inicio)**, donde **m** es la longitud del palíndromo más largo e **inicio** el inicio del palíndromo en S.

1. Inicializar la longitud del palíndromo **m = 1** e **inicio=0**. (longitud e inicio del palíndromo)

2. Para i = 1 hasta n:

3. (Para *substrings* de longitud impar)

4. Hacer **j = 1**

5. Si **i-j >= 1** y **i-j <= n** y **S[i-j] == S[i+j]**

6. **j = j + 1** e ir a 5

7. Si la **longitud = 2j-1** del palíndromo formado es mayor que m: (guardar palíndromo)

8. **m = longitud** e **inicio = i-j+1**

9. (Para *substring* de longitud par)

10. Hacer **j = 0**

11. Si **i-j-1 >= 1** y **i-j <= n** y **S[i-j-1] == S[i+j]**

12. **j = j + 1** e ir a 11

13. Si la **longitud = 2*j** del palíndromo formado es mayor que **m:** (guardar palíndromo)

14. **m = longitud** e **inicio = i-j**

15. regresar (**inicio,m**)

Una posible implementación se muestra a continuación. Hay que recordar que en la implementación en C++, los *strings* inician en 0, por lo que los límites cambiarán un poco.

```
1  pair<int,int> palindromeLargoNaive(string S){
 2   int n = S.length();
 3   int m = 1, j=0, inicio = 0;
 4   string s = "";
 5
 6   for (int i=0; i<n; i++){
 7     // para substrins de longitud impar con centro en i
 8     j = 1;
 9     while (i-j >= 0 && i+j < n && S[i-j] == S[i+j])
10       j++;
11     if (2*j-1 > m){
12       m = 2*j-1;
13       inicio = i-j+1;
14     }
15     j = 0;
16     while (i-j-1 >= 0 && i+j < n && S[i-j-1] == S[i+j])
17       j++;
18     if (2*j > m){
19       m = 2*j;
20       inicio = i-j;
21     }
22   }
23   pair<int,int> res(inicio,m);
24   return res;
25 }
```

En las líneas 9 y 16, las condiciones colocadas en ese orden aprovechan el *lazy evaluation* de C++ al evaluar los **and (&&)**, es decir, como la única forma de que la condición completa sea verdadera es que todas las partes sean verdaderas (porque están unidas con **&&**) las va evaluando de izquierda a derecha y si no se cumple alguna, ahí termina la evaluación y regresa ***false***. Esto hace que si alguna resta es negativa o alguna suma es mayor a **n**, la comparación de los caracteres nunca se llega a dar (si se diera, marcaría error al salirse de los límites del *string*).

4.5.2 Algoritmo de Manacher

El algoritmo de Manacher, llamado así porque fue propuesto por Glenn K. Manacher en 1975, logra resolver el problema del

palíndromo más largo de un *string* **S** en una forma muy eficiente, con una complejidad *O(n)* proporcional a la longitud **n** del *string*.

El algoritmo de Manacher es un poco complicado de entender. De la misma forma que el algoritmo *naive*, se basa en el cálculo del máximo número de caracteres que forman un palíndromo centrado en la posición **i**, sin embargo tiene tres grandes diferencias:

- Como se observó en el algoritmo naive, hay que hacer procesos diferentes si se trata de encontrar palíndromos de tamaño impar, los cuales están centrados en la posición i, o si se trata de un palíndromo de tamaño par, el cual está centrado en el carácter derecho de los dos caracteres que están en el centro. El algoritmo de Manacher basa su trabajo en un nuevo string donde se agregan posiciones intermedias, incluyendo una al inicio y otra al final del *string*, lo que garantiza que todo el proceso se puede hacer en un *string* de longitud impar.

- Guarda los valores calculados en un arreglo **L**, el cual tiene la longitud del nuevo *string*.

- Los valores en el arreglo **L** se van calculando de izquierda a derecha y el algoritmo, para una posición i aprovecha los valores ya calculados (los cuales están a la izquierda de **i**) para calcular los que siguen (que están a la derecha de **i**). De esta forma baja el número de comparaciones que debe hacer teniendo ahora una complejidad lineal *O(n)*.

El algoritmo inicia con la construcción del nuevo *string*. Como se comentó, dado un *string* S al que se le quieren encontrar el *substring* que es un palíndromo de máxima longitud se crea un

string T agregando un carácter especial en medio de cada carácter original, además de uno al inicio y otro al final del *string* original, lo cual garantiza que el nuevo *string* T siempre tiene una longitud impar. El carácter que se agrega puede ser cualquiera que no pertenezca al alfabeto del *string* analizado. Veamos un par de ejemplos:

- Para un *string* de longitud par, por ejemplo: **S = baab** de longitud 4, al agregar el carácter especial *pipe* ("**|**"), el nuevo *string* **T = |b|a|a|b|** tiene una longitud de 9, que sería el tamaño del arreglo L.

- Para un *string* de longitud impar, por ejemplo: **S = bacab** de longitud 5, al agregar el carácter especial *pipe*, el nuevo *string* **T = |b|a|c|a|b|** tiene una longitud de 11, que sería el tamaño del arreglo L.

El valor en la posición **i** del arreglo **L** indica la longitud (en número de caracteres) que tiene el palíndromo más grande centrado en **i**, medido a la derecha o a la izquierda de i. En la Figura 4.7 se puede observar un ejemplo de un *string* y su arreglo **L**.

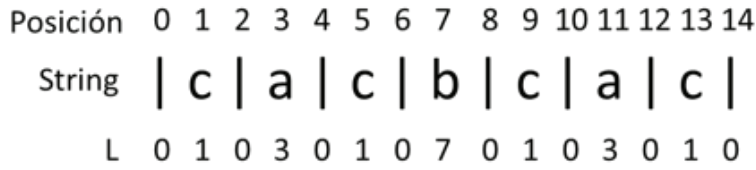


Figura 4.7 Ejemplo de un string para el algoritmo de Manacher y su correspondiente arreglo L

En la Figura 4.7, si tomamos la posición **11** vemos que el valor de **L** en 11 es **L[11]=3**, lo que significa que 3 caracteres a la izquierda y a la derecha de la posición 11, junto con el carácter de la posición 11 forman el máximo palíndromo que se puede

formar tendiendo como centro la posición 11, el cual es **|c|a|c|** de longitud 7. En la posición 2, el valor de L[2]=0, lo que significa que el máximo palíndromo centrado en 2 no tiene caracteres a su derecha ni a su izquierda, es decir, el máximo palíndromo centrado en 2 está formado solo por el carácter en la posición 2, que es **|**. El objetivo del algoritmo de Manacher es calcular todos los valores del arreglo **L** para un *string* dado.

Manacher se dio cuenta que la propiedad de simetría de un palíndromo podría ayudar en el cálculo de **L** aprovechando los valores que ya se habían calculado anteriormente. Sin embargo, las condiciones que se deben cumplir para poder disminuir la cantidad de cálculos no son triviales y se deben estudiar con mucho cuidado para entenderlas. Manacher encontró que solo había 4 casos posibles y al implementarlos se reduce la complejidad. Antes de ver los 4 casos, establezcamos una notación que nos ayudará a su entendimiento, y podemos ver en la Figura 4.8.

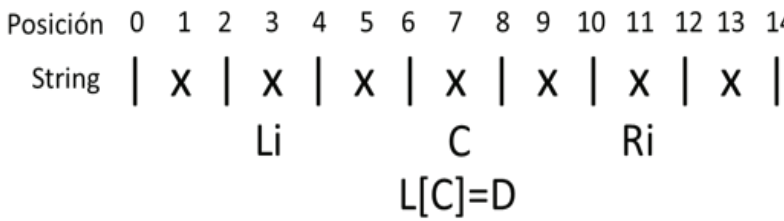


Figura 4.8 Notación usada en los casos del algoritmo de Manacher

Notación:

- **C:** última posición calculada de **L**. Todos los valores en posiciones menores o iguales en **C** en **L** se conocen.
- **Ri:** posición a la derecha de **C**, a la cual se le quiere calcular su valor **L**.

- **Li:** posición a la izquierda de **C** que está separada exactamente el mismo número de caracteres de **C** que **Ri**. Esto implica que **Ri − C = C − Li.**

- **D:** el valor ya calculado de **L** en la posición **C**, es decir, **L[C] = D**.

Ahora sí, veamos los casos, considerando que la longitud del *string* original es **n** por lo que la última posición del *string* aumentado es la **2n:**

- **Caso 1: L[Ri] = L[Li]**, se cumple cuando **L[Li] < (C + D) − Ri**.

- **Caso 2: L[Ri] = L[Li]**, se cumple cuando **L[Li] = (C + D) − Ri** y **(C + D) = 2n**.

- **Caso 3: L[Ri] >= L[Li]**, se cumple cuando **L[Li] = (C + D) − Ri y (C + D) < 2n**. En este caso, no conocemos exactamente el valor que debemos colocar en **L[Ri]**, sin embargo, sabemos que ese valor debe ser al menos **L[Li]**, por lo que podemos intentar expandir el palíndromo carácter por carácter a partir de ahí hasta encontrar el valor exacto de **L[Ri]**.

- **Caso 4: L[Ri] >= L[Li]**, se cumple cuando **L[Li] > (C + D) − Ri y (C + D) < 2n**. En este caso, no conocemos exactamente el valor que debemos colocar en **L[Ri]**, sin embargo, sabemos que ese valor debe ser al menos **L[Li]**, por lo que podemos intentar expandir el palíndromo carácter por carácter a partir de ahí hasta encontrar el valor exacto de **L[Ri]**.

- Hacemos **C = Ri**, siempre que el palíndromo centrado en **Ri** se expanda más allá de **C+D**.

El algoritmo de Manacher se presenta en el Algoritmo 4.6.

Algoritmo 4.6 Algoritmo de Manacher

Entrada: el *string* **S** de longitud **n**, donde se desea encontrar el palíndromo más largo.

Salida: una pareja **(inicio, m)**, donde **m** es la longitud del palíndromo más largo e **inicio** el inicio del palíndromo en S.

1. Crear **T** aumentando **S** con **|**, al inicio, al final y en medio de cada carácter

2. Hacer **N** igual a la longitud de **T**

3. Crear el arreglo **L** de longitud **N** y colocar **L[0]=0** y **L[1]=1**

4. Hacer **maxLong = 0** y **maxCentro = 0** (longitud y centro del máximo palíndrome)

5. Hacer **C = 1** (última posición a la que se le calculó el valor de L)

6. Hacer **Ri = 0** y **Li = 0**

7. Para **Ri = 2** hasta **N-1:**

8. Calcular **Li = C − (Ri − C)**

9. Hacer **expansión = false** (**true** si se requiere expansión)

10. Si **Ri** está dentro del límite del máximo palíndrome centrado en C:

11. Si se cumple el Caso 1:

12. Hacer **L[Ri] = L[Li]**

13. Si no, si se cumple el Caso 2:

14. Hacer **L[Ri] = L[Li]**

15. Si no, si se cumple el Caso 3:

16. Hacer **L[Ri] = L[Li]** y **expansión = true**

17. Si no, si se cumple el caso 4:

18. Hacer **L[Ri] = (C+L[C]) − Ri** y **expansión = true**

19. Si no,

20. Hacer **L[Ri] = 0** y **expansión = true**

21. Si **expansión = true:** // expandir el palíndromo todo lo que se pueda

22. Mientras **Ri + L[Ri] < N and Ri − L[Ri] > 0 and T[Ri − L[Ri] − 1) == T[Ri + L[Ri] + 1):**

23. Hacer **L[Ri] = L[Ri] + 1**

24. Si el nuevo palíndrome se expande más allá de **C:**

25. Hacer **C = Ri**

26. Si **L[Ri] > maxLong:**

27. Hacer **maxLong = L[Ri]** y **maxCentro = Ri** (se encontró un nuevo máximo palíndromo)

28. (calcular el inicio del máximo palíndromo encontrado en el *string* original **S**)

29. Hacer **inicio = (maxCentro − maxLong) / 2** (porque la longitud de **T** es el doble de la de **S**)

30. (calcular la longitud **m** del máximo palíndromo encontrado en el *string* original **S**)

31. Hacer **m = maxLong**

32. Regresar **(inicio, m)**

Veamos un ejemplo, paso a paso, sobre cómo aplicar el Algoritmo 4.6 a un *string*. Sea el *string* **S = xcbcc**. El resultado final se vería como en la Figura 4.9.

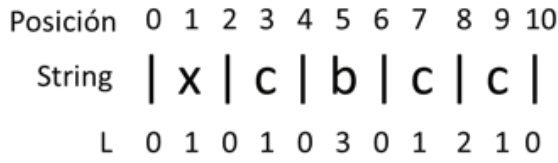


Figura 4.9 Resultado final de la aplicación del algoritmo de Manacher a S = xcbcc

0. Iniciamos el algoritmo con: L[0]=0, L[1]=1, C=1 (última posición calculada), Ri = 2 (posición a ser calculada), maxLong = 1 (máxima longitud del palíndromo encontrada hasta ahora) y maxCentro = 1 (centro del máximo palíndromo encontrado hasta ahora).

1. Li = 0. Como Ri = 2 está dentro de la posición extrema del palíndromo centrado en C=1 de longitud L[C] = 1, vemos el caso que corresponde.

2. Al calcular el valor L[Li] – [(C+L[C]) – Ri] = 0 – [(1 + 1) – 2] = 0, pero C + L[C] = 1 + 1 = 2 es menor que la longitud del *string* T, se trata del CASO 3.

3. Hacemos L[Ri] = L[Li] = 0 y vamos a expansión.

4. Ya no se puede hacer más expansión porque Ri – L[Ri] < 0, así que L[Ri] = 0.

5. Como el nuevo palíndromo centrado en 2 no se extiende más allá del palíndromo centrado en C, se mantiene C = 1.

6. Hacemos Ri = Ri + 1 = 3 y repetimos el ciclo.

7. Li = -1. Como Ri = 3 está fuera de la posición extrema del palíndromo centrado en C=1 de longitud L[C] = 1, hacemos L[3] = 0 y comenzamos expansión.

8. La expansión nos lleva a 1 carácter (el |, que está a ambos lados de la posición 3) dando como resultado el palíndromo **|c|**, lo que indica que L[3] = 1.

9. Como el nuevo palíndromo se sale de los límites del anterior palíndromo se hace C = Ri = 3.

10. No se encontró un palíndromo más grande (fue del mismo tamaño) que el que se tenía, por lo que **maxCentro** y **maxLong** permanecen en 1.

11. Hacemos Ri = Ri + 1 = 4 y repetimos el ciclo.

12. Li = 2. Como Ri = 4 está dentro de la posición extrema del palíndromo centrado en C=3 de longitud L[C] = 1, vemos el caso que corresponde.

13. Al calcular el valor L[Li] – [(C+L[C]) – Ri] = 0 – [(3 + 1) – 4] = 0, pero C + L[C] = 3 + 1 = 4 es menor que la longitud del *string* T, se trata del CASO 3.

14. Hacemos L[Ri] = L[Li] = 0 y vamos a expansión.

15. Ya no se puede hacer más expansión porque T[3] != T[5], así que L[Ri] = 0.

16. Como el nuevo palíndromo centrado en 4, de longitud L[4] = 0, no se extiende más allá del palíndromo centrado en C de longitud L[C] = 1, se mantiene C = 3.

17. Hacemos Ri = Ri + 1 = 5 y repetimos el ciclo.

18. Li = 1. Como Ri = 5 está fuera de la posición extrema del palíndromo centrado en C=3 de longitud L[C] = 1, hacemos L[5] = 0 y comenzamos expansión.

19. La expansión nos lleva a 3 caracteres dando como resultado el palíndromo **|c|b|c|**, lo que indica que L[5] = 3.

20. Como el nuevo palíndromo se sale de los límites del anterior palíndromo se hace C = Ri = 5.

21. Como se encontró un palíndromo más grande (de longitud 3) que el que se tenía hasta el momento (de longitud 1), se actualizan **maxCentro=5** y **maxLong=3**.

22. Hacemos Ri = Ri + 1 = 6 y repetimos el ciclo.

23. Li = 4. Como Ri = 6 está dentro de la posición extrema del palíndromo centrado en C=5 de longitud L[C] = 3, vemos el caso que corresponde.

24. Al calcular el valor L[Li] – [(C+L[C]) – Ri] = 0 – [(5 + 3) – 6] = – 2, se trata del CASO 1 y simplemente se hace L[Ri] = L[Li]=0.

25. No se cambia C ni se modifica el máximo palíndromo.

26. Hacemos Ri = Ri + 1 = 7 y repetimos el ciclo.

27. Li = 3. Como Ri = 7 está dentro de la posición extrema del palíndromo centrado en C=5 de longitud L[C] = 3, vemos el caso que corresponde.

28. Al calcular el valor L[Li] – [(C+L[C]) – Ri] = 1 – [(5 + 3) – 7] = 0, pero C + L[C] = 5 + 3 = 8 es menor que la longitud del *string* T, se trata del CASO 3.

29. Hacemos L[Ri] = L[Li] = 0 y vamos a expansión.

30. La expansión nos lleva a 1 carácter (el |, que está a ambos lados de la posición 7), dando como resultado el palíndromo **|c|**, lo que indica que L[7] = 1.

31. No se cambia C ni se modifica el máximo palíndromo.

32. Hacemos Ri = Ri + 1 = 8 y repetimos el ciclo.

33. Li = 2. Como Ri = 8 está dentro de la posición extrema del palíndromo centrado en C=5 de longitud L[C] = 3, vemos el caso que corresponde.

34. Al calcular el valor L[Li] – [(C+L[C]) – Ri] = 0 – [(5 + 3) – 8] = 0, pero C + L[C] = 5 + 3 = 8 es menor que la longitud del *string* T, se trata del CASO 3.

35. Hacemos L[Ri] = L[Li] = 0 y vamos a expansión.

36. La expansión nos lleva a 2 caracteres dando como resultado el palíndromo **|c|c|**, lo que indica que L[8] = 2.

37. Como el nuevo palíndromo se sale de los límites del anterior palíndromo se hace C = Ri = 8.

38. No se modifica el máximo palíndromo (3) porque se encontró uno menor (2).

39. Hacemos Ri = Ri + 1 = 9 y repetimos el ciclo.

40. Li = 7. Como Ri = 9 está dentro de la posición extrema del palíndromo centrado en C=8 de longitud L[C] = 2, vemos el caso que corresponde.

41. Al calcular el valor L[Li] – [(C+L[C]) – Ri] = 1 – [(8 + 2) – 9] = 0, pero C + L[C] = 8 + 2 = 10 es igual que la longitud del *string* T, se trata del CASO 2 y simplemente se hace L[Ri] = L[Li]=1.

42. No se cambia C ni se modifica el máximo palíndromo.

43. Hacemos Ri = Ri + 1 = 10 y repetimos el ciclo.

44. Li = 6. Como Ri = 10 está fuera de la posición extrema del palíndromo centrado en C=8 de longitud L[C] = 2, hacemos L[10] = 0 y comenzamos expansión.

45. No se puede hacer más expansión porque la posición 10 es la final, lo que indica que L[5] = 0.

46. Como el nuevo palíndromo se sale de los límites del anterior palíndromo se hace C = Ri = 10.

47. Ya se llegó a la posición final Ri = 10, el proceso termina y se procede a regresar el máximo palíndromo encontrado.

48. Como T es el doble de longitud que S, para encontrar el inicio del máximo palíndromo encontrado en **S** al **maxCentro** se le resta el **maxLong** y se hace división entera entre 2, esto da **inicio = (maxCentro – maxLong)/2 = (5 – 3)/2 = 1**, es decir, el palíndromo de longitud máxima inicia en la posición 1 del *string* **S**.

49. Nuevamente, recordando que la longitud de T es el doble de la de S, la longitud del máximo palíndromo en S es simplemente la maxLong encontrada, es decir, **longitud = maxLongitud = 3**.

50. Se regresa **(inicio, longitud) = (1, 3)**, que inidica qe el máximo palíndromo encontrado en **S = xcbcc** es el *string* **cbc**.

Se puede observar que en el *string* aumentado, los caracteres del *string* original siempre ocupan las posiciones impares (1, 3, 5, etc.), esto nos va a ayudar cuando se presente el *substring* que forma el palíndromo más largo, ya que solo se deben presentar los caracteres originales. Una posible implementación se presenta a continuación.

```
string aumenta(string S){
  string s = "";
  for (char c:S){
    s = s+"|"+c;
  }
  return s+"|";
}

pair<int,int> manacher(string S){
  pair<int,int> res; // resultado (inicio, longitud)
  if (S.length() == 0) // S es nulo
    return res;
  string T = aumenta(S); // llamar a función
  int N = T.length();
  // longitud y centro del máximo palíndromo encontrado
  int maxLong=1, maxCentro=1; // Hasta ahora posición 1
  int L[N];
  int C = 1;
  int Li = 0, Ri = 0;
  bool expansion = false; // true si requiera expansión

  L[0]=0; L[1]=1;
  for (Ri=2; Ri<N; Ri++){
    expansion = false;
    Li = C - (Ri-C);
    if ((C+L[C])-Ri >= 0){
      if(L[Li] < (C+L[C])-Ri) // Caso 1
        L[Ri] = L[Li];
      else if(L[Li] == (C+L[C])-Ri && (C+L[C]) == N-1) // Caso 2
        L[Ri] = L[Li];
      else if(L[Li] == (C+L[C])-Ri && (C+L[C]) < N-1){ // Caso 3
        L[Ri] = L[Li];
        expansion = true; // requiere expansión
      }
      else if(L[Li] > (C+L[C])-Ri){ // Case 4
        L[Ri] = (C+L[C])-Ri;
        expansion = true; // requiere expansión
      }
    }
    else{
      L[Ri] = 0;
      expansion = true; // requiere expansión
    }
    if (expansion) // hacer la expansión hasta donde se pueda
      while ((Ri + L[Ri]) < N && (Ri - L[Ri]) > 0
             && T[Ri+L[Ri]+1] == T[Ri-L[Ri]-1])
        L[Ri]++;
    if (Ri + L[Ri] > (C + L[C]))
      // si el nuevo palíndromo se expande más allá de C
      C = Ri;
```

```
51      if(L[Ri] > maxLong){
52        // Guardar longitud y centro del palíndromo más grande,
53        // hasta ahora
54        maxLong = L[Ri];
55        maxCentro = Ri;
56      }
57    }
58    // obtener inicio y longitud del máximo palíndromo encontrado
59    // recordando que la longitud de T es el doble de la de S
60    res.first = (maxCentro - maxLong)/2; // inicio en S
61    res.second = maxLong; // longitud en S
62    return res;
63  }
```

Si el *string* S es nulo, se regresa (0,0) (líneas 11 y 12). La función aumenta (línea 1) se llama en la línea 13. Las posiciones 0 y 1 en L siempre inician con L[0] = 0 y L[1] = 1 (línea 22), independientemente del *string* (que no sea nulo).

Para cada nueva posición de **Ri** (línea 23), se debe calcular **Li** (25) y se coloca la variable **expansión** a ***false*** (línea 24). En el proceso, se analizan los casos (líneas 27, 29, 31 y 35), se expande cuando sea necesario (línea 44), se cambia el valor de C cuando sea necesario (línea 48) y, si se encontró un palíndromo mayor que el que se tenía hasta ahora (línea 51), se guardan los nuevos valores de su centro y su longitud en las variables **maxCentro** y **maxLongitud** (líneas 54 y 55), respectivamente. Al final se calcula el inicio y la longitud (líneas 60 y 61) del máximo palíndromo encontrado en el *string* original S, para lo cual es necesario recordar que la longitud de T es el doble de la longitud de S.

En la implementación anterior todavía se pueden hacer algunas mejoras. Por ejemplo, como en el *string* aumentado, los caracteres del *string* original siempre ocupan las posiciones impares (1, 3, 5, etc.) y, además, los caracteres con los que aumentamos el *string* original (en este caso el **|**), siempre son iguales, cuando se requiere hacer una expansión, se pueden comparar solo los caracteres en el *string* original y luego multiplicarlos por 2. Si estas comparaciones se pueden hacer sobre el *string* original, en reali-

dad, no hace falta crear el *string* aumentado, basta con hacer **L** del tamaño del *string* aumentado. Por otro lado, algunas operaciones se repiten mucho y se dejaron para mayor claridad, pero en realidad se pueden hacer una vez, asignarla a una variable y usarla posteriormente. Tal es el caso del límite a la derecha **C+L[C]** (líneas 26, 27, 29, 31, 35, 36 y 48) entre otros.

4.6 Hash Strings

Los algoritmos de *hashing* son muy útiles en muchas áreas de las ciencias computacionales, sobre todo porque, usando una buena función de *hash*, es posible reducir la búsqueda a un tiempo casi constante $O(1)$.

El problema que se desea resolver es el de la comparación de dos *strings* que se define a continuación.

Problema de la comparación de dos *strings*

Dados dos *strings* S_1 de longitud n_1, y S_2, de longitud n_2, indicar si son iguales o no.

La forma *naive* de resolver este problema es usando fuerza bruta, esto es recorrer posición por posición los dos *strings* y comparar sus caracteres. Si todos los caracteres son iguales, se puede concluir que los dos *strings* son el mismo y si en algún carácter hay diferencia se detiene el proceso y se concluye que los *strings* son diferentes. Este algoritmo tiene un orden $O(\min(n_1,n_2))$, donde $\mathbf{n_1}$ y $\mathbf{n_2}$ son las longitudes de los dos *strings* respectivamente.

Es posible utilizar algoritmos *hash* para poder comparar dos *strings* en forma más eficiente. Con una buena función *hash* el orden de este algoritmo será $O(1)$.

La idea principal de una función *hash* para *strings* es una función que convierta un *string* en un entero. Desde luego, la función debe cumplir la condición de que si dos *strings* son iguales, les debe asignar el mismo número entero. Cuando la función le asigna el mismo número a dos *strings* diferentes se dice que hubo una colisión. La función ideal sería aquella que a cada *string* le asigne un entero diferente. En realidad, es muy complicado hacer una función con tal condición, pero es suficiente con una función donde la probabilidad de asignarle el mismo entero a dos *strings* diferentes es muy baja y ese tipo de funciones sí se han podido generar. Una de las más comúnmente usadas para *strings* es la llamada *Polynomial Rolling Hash Function*, pero hay algunas más y le recomendamos al lector que investigue sobre ellas.

Sea el *string* **S** de longitud **n**, al que se le quiere aplicar la función *hash*, y sean **S[0], S[1], …, S[n-1]** las representaciones en enteros de cada uno de los caracteres que lo forman, la función *Polynomial Rolling Hash Function* para el *string* S se define como:

$$prhf(S) = (S[0] + S[1]p + S[2]p^2 + \cdots + S[n-1]p^{n-1}) \bmod m = \left(\sum_{i=0}^{n-1} S[i]p^i\right) \bmod m \quad (4.1)$$

Donde **p** y **m** son números enteros positivos que se deben seleccionar. Una forma común de seleccionar **p** es que sea un número primo cercano al número de caracteres en el alfabeto, por ejemplo, para los caracteres en español, que son 27 (sin incluir **ch** y **ll**), el número primo más cercano es el 31, por lo que **p = 31** es una buena opción. Si se incluyen también las mayúsculas, el conjunto sube a 54 y entonces **p = 53** es una buena opción. La **m** debe ser un número muy grande debido a que la probabilidad de que haya colisión es aproximadamente **1/m**. Una buena se-

lección ha sido un número primo muy grande, por ejemplo: **m = 10^9 + 9**, lo que nos da una probabilidad de colisión de alrededor de 10^{-9}.

Una probabilidad de colisión de **10^{-9}** es muy baja, pero eso sucede si se hace una sola comparación. En algunos problemas es necesario comparar un *string* con un conjunto de otros strings. Si ese conjunto de strings es de 10^6 *strings*, la probabilidad de que suceda una colisión aumenta a 10^{-3} y si se comparan un conjunto de 106 strings entre ellos mismo la probabilidad se acerca peligrosamente a 1.

Una forma muy común y simple de obtener mejores probabilidades de colisión, incluso con muchas comparaciones, es hacer dos funciones *hash* con diferentes **p** y **m**. Se aplican las dos funciones hash y solo se declaran iguales los *strings* si en ambas son iguales. En este caso, si **m** para la segunda función es también cercana a 10^9, aplicar las dos funciones es equivalente a tener una m aproximadamente igual a 10^{18}.

Hay varias formas de asignar un entero a los caracteres de un alfabeto. Una de ellas podría ser, **a = 1, b = 2, …, z = 27**. (no se recomienda iniciar en 0 porque los *strings* formados solo por letras a tendrían el mismo valor). Hay algunos programadores que trabajan directamente con el ASCII de los caracteres, aunque se trabaja con números muy grandes.

La Ecuación 4.1 indica que primero se hace la suma y al final se hace el módulo con **m**, sin embargo, de esta forma se corre el riesgo de obtener un número muy grande que se salga del tipo de dato entero que se está manejando. Como para implementarlo se va a usar un ciclo, una mejor forma de hacerlo es tomar el módulo con **m** en cada paso del ciclo.

A continuación, se presenta una posible implementación de la Función 4.1, suponiendo que el alfabeto solo consta de letras minúsculas del alfabeto en español y usando la asignación de números consecutivos para las letras minúsculas desde el 1 para la **a**, hasta el 27 para la **z**. Esta es una implementación totalmente directa de la función 4.1 con la modificación del módulo en cada paso ya comentada; Sin embargo, existen métodos para hacer este cálculo de forma más eficiente.

```
1 long long prhf(string S){
2   int n = S.length();
3   int p = 31;
4   int m = 1e9 + 9;
5   long long valorHash = 0;
6   long long potencia = 1;
7
8   for (int i=0; i<n; i++){
9     valorHash = (valorHash + (S[i] - 'a' + 1) * potencia) % m;
10    potencia = (potencia * p) % m;
11  }
12  return valorHash;
13 }
```

Podemos ver que la potencia $\mathbf{p^i}$ se va calculando paso a paso por medio de multiplicaciones, lo que corresponde a una especie de memoization. Esto es más eficiente que calcular la potencia completa en cada momento con la función **pow(p,i)**. El **mod m** se hace en cada paso, no solo para el valor *hash* sino también para **p**, con lo que se evita que en algún paso se obtenga un número mayor al permitido por el tipo de dato en el que se va guardando, al hacer esto se puede trabajar con *strings* más grandes.

4.7 Arreglo de sufijos (*Suffix Array*)

El arreglo de sufijos (Suffix Array) es una estructura de datos que fue propuesta por Udi Manber y Gene Myers en 1990. Es muy utilizada en varias aplicaciones relacionadas con el análisis de strings. Se define de la siguiente forma:

Arreglo de sufijos

Dado un *string* **S**, de longitud n, su arreglo de sufijos es un arreglo de enteros que contiene las posiciones que tienen los **n+1** sufijos en el *string* **T = S$**, ordenados lexicográficamente (alfabéticamente), considerando **$** como el primer carácter del alfabeto.

La forma *naive* de crearlo se indica en el Algoritmo 4.7.

Algoritmo 4.7 Creación de un arreglo de sufijos

Entrada: el *string* **S** de longitud **n**.

Salida: el arreglo de sufijos

1. Formar el *string* de trabajo **T = S$**.
2. Formar todos los sufijos de T.
3. Ordenar lexicográficamente los sufijos encontrados considerando a $ como el primer símbolo del alfabeto.
4. Formar el arreglo A con las posiciones de inicio en el string T de cada uno de los sufijos ordenados lexicográficamente.
5. Regresar A

Por ejemplo: dado **S = bcdeabcedeb** de longitud **n = 11** agregamos el $ al final del *string* **T = bcdeabcedeb$**, con lo que obtenemos los sufijos mostrados en la tabla 4.1.

Sufijo	Posición en T
$	12
b$	11
eb$	10
deb$	9
edeb$	8
cedeb$	7
bcedeb$	6
abcedeb$	5
eabcedeb$	4
deabcedeb$	3
cdeabcedeb$	2
bcdeabcedeb$	1

Tabla 4.1 Sufijos del string **T= bcdeabcedebS**

Se ordenan lexicográficamente los sufijos obtenidos y quedan como los mostrados en la Tabla 4.2.

Sufijo	Posición en T
$	12
abcedeb$	5
b$	11
bcdeabcedeb$	1
bcedeb$	6
cdeabcedeb$	2
cedeb$	7
deabcedeb$	3
deb$	9
eabcedeb$	4
eb$	10
edeb$	8

Tabla 4.2 Sufijos del string **T= bcdeabcedeb$** ordenados lexicográficamente

De acuerdo con la Tabla 4.2, el arreglo de sufijos de A para el *string* **S** está formado por los números de la columna "Posición en T" que indican la posición en la que cada sufijo inicia dentro del *string* S y queda como se muestra a continuación:

$$A = [12, 5, 11, 1, 6, 2, 7, 3, 9, 4, 10, 8]$$

La posición 12 del *string* **S** no existe (es el **$** en el *string* **T**), lo que significa que dicho sufijo se refiere al *string* nulo siendo este el primero del arreglo de sufijos.

Una posible forma de implementar este algoritmo es la siguiente:

```
 1  vector<int> suffixArrayNaive(string S){
 2    int n = S.length()+1;
 3    vector<string> sa;
 4    vector<int> A(n);
 5
 6    S = S+"$";
 7    for (int i=0; i<n; i++)
 8      sa.push_back(S.substr(n-i-1, i+1));
 9    sort(sa.begin(), sa.end());
10    for (int i=0; i<n; i++)
11      A[i] = n - sa[i].size() + 1;
12    return A;
13  }
```

En esta implementación se utiliza la función **substr(a,b)** (línea 8) para obtener los sufijos a partir del *string*, donde **a** es la posición de inicio del *substring* y **b** es el número de caracteres que lo forman. Además, se utiliza la función **sort(a,b)** de la librería algorithms para ordenar un vector de sufijos (línea 9), donde **a** es la posición inicial y **b** es la posición final del tramo del arreglo que se desea ordenar; en este caso, **a=sa.begin()**, es el elemento inicial del arreglo y **b=sa.end()** es el último elemento del arreglo para que ordene el arreglo completo.

El algoritmo *naive* para construir el arreglo de sufijos requiere primero ordenar los **n** sufijos. El número de comparaciones entre sufijos para poder ser ordenados tiene una complejidad de $O(nlogn)$, donde **n** es el número de sufijos a ser ordenados. Sin embargo, en cada comparación se deben comparar todos los caracteres de cada sufijo, lo cual tiene una complejidad de $O(n)$, por lo que la complejidad total del algoritmo *naive* es $O(n^2\ logn)$.

Desde luego, hay mejores formas de formar el arreglo de sufijos, algunas de las cuales logran hacerlo en la menor complejidad posible (son óptimos) que es $O(n)$. Recomendamos al lector investigar sobre estos algoritmos.

4.8 *Trie*

El *trie* es una estructura de datos muy útil para resolver problemas relacionados con *strings*, es también conocido como **árbol de prefijos**. El nombre proviene de la palabra RE**TRIE**VAL, que significa recuperación. Internamente funciona como un árbol ordenado, donde cada nodo representa un carácter de una palabra insertada en el ***trie*** y los nodos hijos de este son caracteres que aparecen después de él en la palabra. La raíz del árbol es el carácter vacío, ya que se puede considerar que este aparece al principio de cualquier palabra. En la figura 4.10 se puede apreciar la estructura de datos *trie* generada con las palabras: loco, loca, lápiz, poca y pozo.

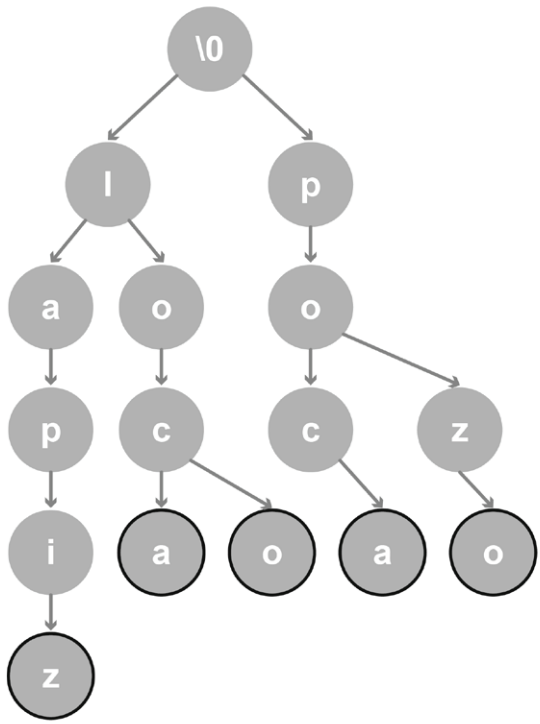


Figura 4.10 Ejemplo de un *trie*

Al construir un *trie* se requiere almacenar en cada nodo lo siguiente:

- Letra que representa.
- Si es o no la última letra de una palabra.
- Nodo padre.
- Nodos hijos.

Además, se requieren dos métodos más:

- **getChild**: para saber si un nodo es hijo de ese nodo
- **addChild**: para agregar un nuevo hijo al nodo.

El constructor del *trie* quedaría de la siguiente forma:

```
1  class trie{
2    public:
3      root = new nodeTrie('\0', null);
4      void insert(string word);
5      bool search(string word);
6
7    private:
8      nodeTrie root;
9  };
```

El nodo del *trie* quedaría de la siguiente forma:

```
 1 class nodeTrie{
 2   public:
 3     nodeTrie(char c, nodeTrie p){ parent = p; letter = c; }
 4     nodeTrie getChild(char c) { return child[c-'a']; }
 5       nodeTrie addChild(nodeTrie node)
 6             { child[node.content - 'a'] = node; }
 7
 8   private:
 9     char letter;
10     bool end;
11     nodeTrie parent;
12     nodeTrie[] child = new nodeTrie[26];
13 };
```

El algoritmo para el método de ***insert*** del *trie* es:

Algoritmo 4.8 método *insert* del *trie*

Entrada: la palabra que se desea insertar.

Salida: nada

1. Inicializar el nodeTrie llamado current en root.

2. Para cada carácter de la palabra:

2.1 Se inicializa la variable c con el carácter correspondiente

2.2 Se inicializa el nodoTrie sub con el hijo de current para ese carácter.

2.3 ¿sub != nullptr?

2.3.1 Sí,

2.3.1.1 current toma el valor de sub

2.3.2 No,

2.3.2.1 A current le agregas un nuevo hijo con los valores de c y current.

2.3.2.2 Al nodeTrie current le asignas el hijo de current para el carácter c.

2.4. ¿Llego al final de la palabra?

2.4.1 Sí,

2.4.1.1 El valor de end del nodeTrie current se vuelve true.

La implementación del método de ***insert*** del *trie* quedaría de la siguiente forma:

```
1 void trie::insert(string word){
2     nodeTrie current = root;
3     for(int i=0; i < word.length(); i++){
4         char c = word[i];
5         nodeTrie sub = current.getChild(c);
6         if (sub != nullptr){
7             current = sub;
8         }
9         else {
10            current.addChild(new nodeTrie(c, current));
11            current = current.getChild(c);
12        }
13        if (i == word.length()-1){
14            current.end = true;
15        }
16    }
17 }
```

El algoritmo para el método de ***search*** del *trie* es:

Algoritmo 4.9 método *search* del *trie*

Entrada: la palabra que se desea insertar

Salida: un boleano que dice si se encontró o no la palabra

1. Inicializar el nodeTrie llamado current en root.

2. Para cada carácter de la palabra:

   2.1 Se inicializa la variable c con el carácter correspondiente

   2.2 Se inicializa el nodoTrie sub con el hijo de current para ese carácter.

   2.3 ¿sub == nullptr?

      2.3.1 Sí,

         2.3.1.1 Regresa falso

3. Regresa el contenido de end del nodeTrie current.

La implementación del método de ***search*** del *trie* quedaría de la siguiente forma:

```
1  bool trie::search(string word){
2      nodeTrie current = root;
3      for(int i=0; i < word.length(); i++){
4          char c = word[i];
5          nodeTrie sub = current.getChild(c);
6          if (sub == nullptr){
7              return false;
8          }
9      }
10     return current.end;
11 }
```

4.9 Ejercicios del capítulo 4

1. Obtén el prefijo de 5 caracteres y el sufijo de 4 caracteres para el *string* **S = abcdabcde**.

2. Aplicando el algoritmo 4.1, obtenga la función-Z para el *string* **S = abcdabceabd** y comprueba su valor con la implementación de dicho algoritmo.

3. Obtén los valores del preprocesamiento del patrón **P = abcaba** utilizando el algoritmo 4.3 y compruébalo con su implementación.

4. Aplicando al algoritmo KMP (algoritmo 4.4), indica cuál sería su salida para el patrón **P = abcaba**, en el texto **T = abcabcabcababcd**. Comprueba tu resultado con la implementación de dicho algoritmo.

5. Realiza una comparación en tiempo de procesamiento de los algoritmos 4.2 y 4.3, para resolver el mismo problema del problema anterior. ¿Es muy grande la diferencia? ¿Corresponde esta diferencia a lo que indica el análisis de sus complejidades? Comenta tus resultados.

6. Usando el algoritmo de Manacher (algoritmo 4.6), obtén los valores del arreglo **L** para el *string* **S = ababacdbcacbddcabb** y comprueba tu respuesta modificando un poco la implementación mostrada para dicho algoritmo.

7. Usando el algoritmo de Manacher (algoritmo 4.6), obtén la posición de inicio y la longitud (en número de caracteres) del palíndromo más grande que se encuentre en él. Comprueba tu resultado mediante la implementación de dicho algoritmo.

8. Realiza una comparación en tiempo de procesamiento de los algoritmos 4.5 y 4.6, para resolver el mismo problema del problema anterior. ¿La diferencia es tan grande como lo indican sus complejidades? Comenta tus resultados.

9. Usando la *Polynomial Rolling Hash Function*, obtén los valores hash para los strings **S1 = abcd** y **S2 = ebcd**. Comprueba tu resultado con la implementación de la función mostrada en la sección 4.6.

10. Usando el algoritmo 4.7, obtén el arreglo de sufijos del *string* **S = ddbaacbdcab**. Comprueba tus resultados con la implementación de dicho algoritmo.

11. Usando los algoritmos 4.8 y 4.9, forma un *trie* con las siguientes palabras: **cosa, cocina, cocinero, cartera, carta** y **básico**. No tomes en cuenta los acentos. Comprueba tu resultado con la implementación de dichos algoritmos.

Capítulo 5. Aplicación de técnicas de diseño de algoritmos

5.1 Problema del *substring* común más largo

El problema del *substring* común más largo (***Longest Common Substring*** en inglés) tiene como objetivo encontrar el *substring* común más largo de dos o más *strings* (normalmente son dos), esto es, encontrar aquel *substring* continuo que puede coincidir en todos los *strings* de entrada no necesariamente del mismo tamaño. Este problema puede arrojar múltiples soluciones.

Por ejemplo, teniendo los *strings*: "AABCABA" y "CABCBABACC" el tamaño de la subsecuencia más larga es **3** en varios *substrings*: "ABC", "CAB" y "ABA", en la Figura 5.1 se muestra el resultado de este ejemplo.

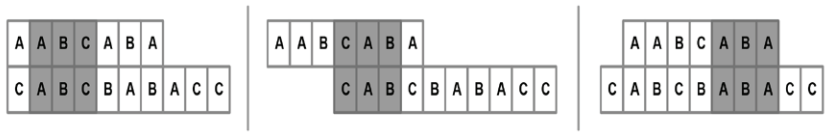


Figura 5.1 Ejemplo del Longest Common Substring

El algoritmo de ***Longest Common Substring*** compara cada uno de los caracteres de ambos *strings* llevando un acumulado por cada combinación continúa utilizando la técnica de programación dinámica.

Planteamiento de la solución recursiva para programación dinámica

Sea **LCS*(i,j,m,c)*** la solución recursiva en donde en **m** deja el valor máximo acumulado de la longitud del *substring* más largo teniendo **i** caracteres del **string1** y **j** caracteres del **string2**.

La solución a encontrar es **LCS(i,j,m,0)**, la cual puede obtenerse de la siguiente forma:

$$LCS(i,j,m,c) = \begin{cases} LCS(i-1,j-1,c+1,c+1), & S1[i] = S1[j] \text{ and } m = c \\ LCS(i-1,j-1,m,c+1), & S1[i] = S1[j] \text{ and } m > c \\ \max(LCS(i-1,j,m,0), LCS(i,j-1,m,0)), & S1[i] \neq S1[j] \end{cases} \quad (5.1)$$

A su vez esto se generaliza para cualquier i y cualquier j, parando cuando i o j lleguen a -1.

El resultado estará en la variable **m**.

A continuación, se muestra el algoritmo ***Longest Common Substring*** memorizando y utilizando la técnica de programación dinámica:

Algoritmo 5.1 Algoritmo de *Longest Common Substring*

Entrada: los dos *string* de los que se desea obtener el *Longest Common Substring*.

Salida: la longitud del *substring* común más largo.

1. Inicializar una matriz LCS de tamaño, longitud del string1 por longitud del string2

2. Inicializar una variable max con 0;

3. Para i desde 0 hasta n-1 (longitud string1)

   3.1 ¿El carácter en el string1 posición i es igual al carácter en la posición 0 del string2?

   3.2 Sí,

      3.2.1 Asignar 1 a la matriz LCS en la posición [i][0]

      3.2.2 Asignar 1 a la variable max

   3.3 No,

      3.3.1 Asignar 0 a la matriz LCS en la posición [i][0]

4. Para j desde 0 hasta m-1 (longitud string2)

   4.1 ¿El carácter en el string1 posición 0 es igual al carácter en la posición j del string2?

   4.2 Sí,

      4.2.1 Asignar 1 a la matriz LCS en la posición [0][j]

      4.2.2 Asignar 1 a la variable max

4.3 No,

4.3.1 Asignar 0 a la matriz LCS en la posición [0][j]

5. Para i desde 1 hasta n-1 (longitud string1)

5.1 Para j desde a hasta m-1 (m es la longitud string2)

5.1.1 ¿El carácter i del string1 es igual al carácter j del string2?

5.1.2 Sí,

5.1.2.1 Asignar LCS en la posición [i-1][j-1] más 1 a la posición [i][j]

5.1.2.2 ¿Es LCS en la posición [i][j] mayor a la variable max?

5.1.2.3 Sí,

5.1.2.3.1 Asignar LCS en la posición [i][j] a max.

5.1.3 No,

5.1.3.1 Asignar 0 a LCS en la posición [i][j]

6. Regresa la variable max

La implementación del algoritmo de ***Longest Common Substring*** quedaría de la siguiente forma:

```
int LCS(string S1, string S2){
        int LCS[S1.length()][S2.length()];
        int n = S1.length();
        int m = S2.length();
        int max = 0;
        for (int i=0; i<n; i++){
                if (S1[i] == S2[0]){
                        LCS[i][0] = 1;
                        max = 1;
                }
                else{
                        LCS[i][0] = 0;
                }
        }
        for (int j=0; j<m; j++){
                if (S1[0] == S2[j]){
                        LCS[0][j] = 1;
                        max = 1;
                }
                else{
                        LCS[0][j] = 0;
                }
        }
        for (int i=1; i<n; i++){
                for (int j=1; j<m; j++){
            if (S1[i] == S2[j]){
                LCS[i][j] = LCS[i-1][j-1] + 1;
                     if (LCS[i][j] > max)
                      max = LCS[i][j];
            }
            else
                LCS[i][j] = 0;
                }
        }
        return max;
}
```

Complejidad del algoritmo de *Longest Common Substring*	
Clasificación del problema	P
Tiempo de ejecución (peor caso)	O(n*m)
Memoria adicional (costo mínimo)	O(n*m)

Si se desea reconstruir el primer *substring* que tiene el valor m se requiere guardar la última posición en donde se modificó la variable m y posteriormente reconstruir buscando la ruta del *substring.*

5.2 Problema de la subsecuencia común más larga

El problema de la subsecuencia común más larga (LCS, ***Longest Common Subsequence*** en inglés) consiste en encontrar la subsecuencia más larga común entre 2 o más *strings* (comúnmente solo son 2).

Por ejemplo, teniendo los *strings* "AABCABA" y "CABCBABACC" el tamaño del *substring* más largo es 6 con la subsecuencia: "A, B, C, A, B, A", la Figura 5.2 se muestra el resultado de este ejemplo.

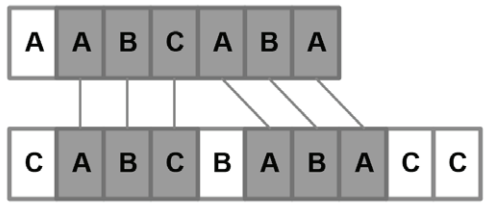


Figura 5.2 Ejemplo del Longest Common Subsequence

El algoritmo de ***Longest Common Subsequence*** compara cada uno de los caracteres de ambos *strings*, sumándole 1 en caso de coincidir al acumulado anterior, si no se queda con el máximo de las dos posibles secuencias de ambos *strings*. Para esto utiliza la técnica de programación dinámica.

Planteamiento de la solución recursiva para programación dinámica

Sea **LCS*(i,j)*** el valor acumulado de la longitud de la subsecuencia más larga teniendo **i** caracteres del **string1** y **j** caracteres del **string2**.

La solución a encontrar es **LCS(i,j),** la cual puede obtenerse de la siguiente forma:

$$LCS(i,j) = \begin{cases} LCS(i-1, j-1) + 1, & S1[i] = S1[j] \\ \max(LCS(i-1, j), LCS(i, j-1)), & S1[i] \neq S1[j] \end{cases} \quad (5.2)$$

A su vez esto se generaliza para cualquier i y cualquier j.

A continuación, se muestra el algoritmo de ***Longest Common Subsequence*** memorizando y utilizando la técnica de programación dinámica:

Algoritmo 5.2 Algoritmo de Longest Common Subsequence

Entrada: los dos *string* de los que se desea obtener el *Longest Common Subsequence*.

Salida: la longitud del subsequence común más largo.

1. Inicializa una matriz LCS de tamaño, longitud del string1 por longitud del string2

2. ¿El carácter en la posición 0 del string1 es igual al carácter en la posición 0 del string2?

3. Sí,

   3.1 A la matriz LCS en la posición [0][0] se le asigna 1

4. No,

   4.1 A la matriz LCS en la posición [0][0] se le asigna 0

5. Para i desde 1 hasta n-1 (longitud string1)

   5.1 ¿El carácter en el string1 posición i es igual al carácter en la posición 0 del string2?

   5.2 Sí,

      5.2.1 Asignar 1 a la matriz LCS en la posición [i][0]

   5.3 No,

      5.3.1 Asignar el contenido de la matriz LCS en la posición[i-1][0] a la matriz LCS en la posición [i][0]

6. Para j desde 1 hasta m-1 (longitud string2)

6.1 ¿El carácter en el string1 posición 0 es igual al carácter en la posición j del string2?

6.2 Sí,

6.2.1 Asignar 1 a la matriz LCS en la posición [0][j]

6.3 No,

6.3.1 Asignar el contenido de la matriz LCS en la posición[0][j-1] a la matriz LCS en la posición [0][j]

7. Para i desde 1 hasta n-1 (longitud string1)

7.1 Para j desde a hasta m-1 (m es la longitud string2)

7.1.1 ¿El carácter i del string1 es igual al carácter j del string2?

7.1.2 Sí,

7.1.2.1 Asignar LCS en la posición [i-1][j-1] más 1 a la posición [i][j]

7.1.3 No,

7.1.3.1 Asignar el máximo de LCS en la posición [i-1][j] y la posición [i][j-1], a la matriz LCS en la posición [i][j]

8. Regresa el contenido de la matriz LCS en la posición [n-1][m-1]

La implementación del algoritmo de ***Longest Common Subsequence*** quedaría de la siguiente forma:

```
1  int LCS(string S1, string S2){
2          int n = S1.length();
3          int m = S2.length();
4          int LCS[n][m];
5          LCS[0][0] = (S1[0] == S2[0]) ? 1 : 0;
6          for (int i=1; i<n; i++){
7                  LCS[i][0] = (S1[i] == S2[0]) ? 1 : LCS[i-1][0];
8          }
9          for (int j=1; j<m; j++){
10                 LCS[0][j] = (S1[0] == S2[j]) ? 1 : LCS[0][j-1] ;
11         }
12         for (int i=1; i<n; i++){
13                 for (int j=1; j<m; j++){
14             LCS[i][j] = (S1[i] == S2[j]) ? LCS[i-1][j-1] + 1 :
15                                 max(LCS[i][j-1], LCS[i-1][j]);
16                 }
17         }
18         return LCS[n-1][m-1];
19 }
```

Complejidad del algoritmo de *Longest Common Subsequence*	
Clasificación del problema	P
Tiempo de ejecución (peor caso)	*O(n*m)*
Memoria adicional (costo mínimo)	*O(n*m)*

Si se desea reconstruir la subsequencia con que se obtiene el valor de salida se requiere ir reconstruyendo a partir del valor de en LCS[n-1][m-1] hacia atrás, guardar y posteriormente reconstruir buscando la ruta de la subsequencia.

5.3 Problema del camino más corto

En muchas ocasiones, requerimos saber la ruta más corta de un punto a otro, ya sea en un viaje, algún proceso de logística o algún proceso de conectividad, todo esto dado una red ca-

rretera o red de conexiones, las cuales pueden estar con diferentes situaciones climáticas o de falta de conexión que en ciertos momentos tengan conectividades deshabilitadas. Es muy importante que este proceso se realice lo más rápido posible, ya que, para la mayoría de los casos, el tiempo es muy valioso para poder transmitir información o viajar entre ellas. Existen 2 algoritmos muy importantes que nos ayudan a solucionar este problema:

- ***Dijkstra:*** este algoritmo da el camino más corto de un punto al resto de los puntos en el grafo.
- ***Floyd-Warshall:*** este algoritmo da el camino más corto de todos los puntos a todos los puntos en el grafo.

5.3.1 Algoritmo de Dijkstra

El algoritmo de ***Dijkstra***, también conocido como el algoritmo de caminos mínimos, fue desarrollado por el científico en computación **Edsger Dijkstra** para determinar el camino más corto para un grafo ponderado dado un nodo de arranque al resto de los nodos utilizando la técnica de Algoritmos Avaros.

La idea básica del algoritmo es ir buscando los caminos más cortos que parten del nodo origen y que llevan a cada uno de los nodos restantes; cuando llega al último nodo, el algoritmo se detiene. Este algoritmo solo funciona para grafos con aristas con costos positivos.

En el ejemplo de la Figura 5.3, partiendo del nodo ***Rey***, el algoritmo va seleccionando los nodos de la siguiente forma:

- En el primer paso, selecciona al arco ***Rey-Lar***, ya que es el vecino con menor costo 5, logrando llegar al nodo ***Lar***.

- En el segundo paso, selecciona al arco ***Lar-Mty***, dado que el acumulado para ir de ***Rey*** a ***Mty*** pasando por ***Lar*** es de 6 y es de menor costo hasta este punto, logrando llegar en este paso a ***Mty***.

- En el tercer paso, selecciona al arco ***Rey-Vic***, que ya es el de menor costo hasta este punto 7, logrando llegar al nodo ***Vic***.

- Como cuarto paso, selecciona el arco ***Vic-Mat***, porque el acumulado para ir de ***Rey*** a ***Mat*** pasando por ***Vic*** es de 9 y es de menor costo hasta este punto, logrando llegar en este paso a ***Mat***, que es el último nodo del grafo.

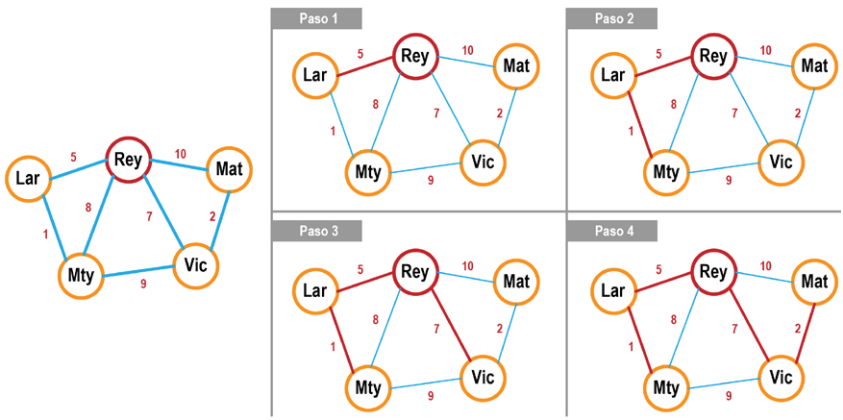


Figura 5.3 Ejemplo del algoritmo de Dijkstra

A continuación, se muestra el algoritmo de ***Dijkstra*** memorizando y utilizando la técnica de algoritmos avaros:

Algoritmo 5.3 Algoritmo de Dijkstra

Entrada: grafo ponderado en algún tipo de representación y el nodo origen.

Salida: costos mínimo a cada nodo.

1. Inicializa el vector de costos en INF excepto el costo al origen el cual se inicializa en cero.

2. Inicializa una fila priorizada (con prioridad costo menor) con el nodo de origen.

3. Mientras la fila priorizada no esté vacía.

   1.1 Sacar el nodo del frente (v) de la Fila Priorizada.

   1.2 Para cada vecino de la Fila Priorizada

      1.2.1 ¿Costo pasado por el nodo v mejora para llegar al vecino?

      1.2.2 Sí,

         1.2.2.1 Actualiza el costo en el vecino

4. Regresa el vector de costos.

La implementación del algoritmo de ***Dijkstra*** quedaría de la siguiente forma:

```
1 /* G =  es un vector de vector de enteros que
2         representa la lista de adjacencia
3 */
4 vector<int> Dijkstra(Graph& G, int source){
5     vector<int> Cost;
6     priority_queue<Vertex, vector<Vertex>, greater<Vertex>> queue;
7     Cost.assign(G.size(), INF);
8     Cost[source] = 0;
9     Vertex vx(0, source);
10    queue.push(vx);
11    while(!queue.empty()){
12        int u = queue.top().second;  // Vertex ID
13        queue.pop();
14        for (int i = 0; i < G[u].size(); i++){
15            Edge e = G[u][i];  // Edge es un par(destino, costo)
16            int v = e.first;
17            int w = e.second;
18            if(Cost[v] > Cost[u] + w){
19                Cost[v] = Dist[u] + w;
20                Vertex vtx(Cost[v], v);
21                queue.push(vtx);
22            }
23        }
24    }
25    return Cost;
26 }
```

Complejidad del algoritmo de Dijkstra	
Clasificación del problema	P
Tiempo de ejecución (peor caso)	$O(\|E\|+\|V\| \log \|V\|)$
Memoria adicional (costo mínimo)	$O(\|V\|)$

Si se desea reconstruir la ruta para conseguir esos costos mínimos se requiere ir guardando el nodo ***u*** cuando pasar por este sea mejor costo que el que se tiene actualmente (línea 18 del código) en un vector, el cual es inicializado con el nodo de origen (*source*).

5.3.2 Algoritmo de Floyd

El algoritmo de Floyd también conocido como el algoritmo de Floyd-Warshall, sirve para encontrar los caminos más cortos de todos los puntos a todos los puntos de un grafo ponderado con costos positivos y negativos (siempre y cuando no haya ciclos negativos). Este algoritmo utiliza la técnica de programación dinámica y fue desarrollado por Robert Floyd en 1962, pero básicamente es el mismo algoritmo desarrollado por Stephen Warshall para encontrar la cerradura transitiva de un grafo.

El algoritmo se basa en una matriz de adyacencia donde se encuentra representado el grafo que va iterando por cada nodo k viendo si mejora el costo de un punto i al punto j pasado por k, o es mejor el costo con el que esa ruta cuenta actualmente. La Figura 5.4 muestra este paso, el cual es la base del algoritmo.

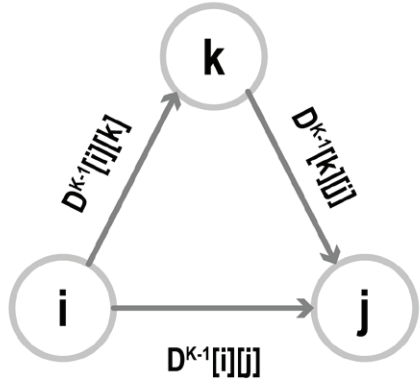


Figura 5.4 Selección del mejor costo para ir i a j pudiendo pasar por k

En el ejemplo de la Figura 5.5 se va iterando los nodos de la siguiente forma:

- Arranca con la matriz de adyacencia original del grafo, la cual llamaremos $\mathbf{D^0}$.

- En la primera iteración, que es de $\mathbf{D^0}$ a $\mathbf{D^1}$ revisa para todos los puntos si pasando por ***Rey*** mejora el costo. Por ejemplo: para ir de ***Lar-Vic*** tiene **INF**, pero pasando por ***Rey*** se obtiene un **12**, por lo tanto, cambia su valor.

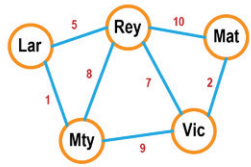
- En la segunda iteración, que es de $\mathbf{D^1}$ a $\mathbf{D^2}$ revisa para todos los puntos si pasando por ***Lar*** mejora el costo. Por ejemplo: para ir de ***Mty-Mat*** tiene **18**, pero pasando por ***Lar*** se obtiene un **16**, por lo tanto, cambia su valor.

- En la tercera iteración, que es de $\mathbf{D^2}$ a $\mathbf{D^3}$ revisa para todos los puntos si pasando por ***Mty*** mejora el costo. Por ejemplo: para ir de ***Lar-Vic*** tiene **11**, pero pasando por ***Mty*** se obtiene un **10**, por lo tanto, cambia su valor.

- En la cuarta iteración, que es de $\mathbf{D^3}$ a $\mathbf{D^4}$ revisa para todos los puntos si pasando por ***Vic*** mejora el costo. Por ejemplo: para ir de ***Rey-Mat*** tiene **10**, pero pasando por ***Vic*** se obtiene un **9**, por lo tanto, cambia su valor.

- En la quinta iteración, que es de $\mathbf{D^4}$ a $\mathbf{D^5}$ revisa para todos los puntos si pasando por ***Mat*** mejora el costo. Para este punto no hay ningún cambio, ya que nadie mejora, por ejemplo, para ir de ***Rey-Vic*** tiene un costo de **7**, y pasando por ***Mat*** es de **12**, por lo tanto, no conviene y se queda como estaba en $\mathbf{D^4}$.

- En este punto se consigue la matriz de costos mínimos que, como son 5 nodos, llamaremos $\mathbf{D^5}$.



K = 0

	1	2	3	4	5
1	0	5	8	7	10
2	5	0	1	INF	INF
3	8	1	0	9	INF
4	7	INF	9	0	2
5	10	INF	INF	2	0

K = 1

	1	2	3	4	5
1	0	5	8	7	10
2	5	0	1	12	15
3	8	1	0	9	18
4	7	12	9	0	2
5	10	15	18	2	0

K = 2

	1	2	3	4	5
1	0	5	6	7	10
2	5	0	1	12	15
3	6	1	0	9	16
4	7	12	9	0	2
5	10	15	16	2	0

K = 3

	1	2	3	4	5
1	0	5	6	7	10
2	5	0	1	10	15
3	6	1	0	9	16
4	7	10	9	0	2
5	10	15	16	2	0

K = 4

	1	2	3	4	5
1	0	5	6	7	9
2	5	0	1	10	12
3	6	1	0	9	11
4	7	10	9	0	2
5	9	12	11	2	0

K = 5

	1	2	3	4	5
1	0	5	6	7	9
2	5	0	1	10	12
3	6	1	0	9	11
4	7	10	9	0	2
5	9	12	11	2	0

Figura 5.5 Ejemplo del algoritmo de Floyd

Planteamiento de la solución recursiva para programación dinámica

Sea D(k,i,j) encontrar el costo menor para ir del nodo i al nodo j pudiendo pasar por los nodos desde 1 hasta k.

La solución a encontrar es D(k,i,j), la cual puede obtenerse de la siguiente forma:

$$D(k,i,j) = \begin{cases} D(k-1,i,j), & NO\ pasar\ por\ k \\ D(k-1,i,k) + D(k-1,k,j), & pasar\ por\ k \end{cases} \quad (5.3)$$

El algoritmo de **Floyd** es:

Algoritmo 5.4 Algoritmo de Floyd

Entrada: grafo ponderado utilizando una representación de matriz de adyacencias (D).

Salida: matriz con costos mínimo de todos los puntos a todos.

1. Para k desde 1 hasta n

1.1 Para i desde 1 hasta n

1.1.1 Para j desde 1 hasta n

1.1.1.1 ¿Es el costo acumulado pasando de i->k + k->j mejor que el que tiene actualmente de i->j?

1.1.1.2 Sí,

1.1.1.2.1 Actualiza el costo de D en la posición i->j con el acumulado de i->k + k->j

La implementación del algoritmo de **Floyd** quedaría de la siguiente forma:

```
/*
   D   = es la matriz de Adyacencias del Grafo
   MAX = es el maximo de nodos que se pueden tener
   n   = el numero de nodos
*/
void floyd(int D[MAX][MAX], int n){
    for (int k=0; k<n; k++){
        for (int i=0; i<n; i++){
            for (int j=0; j<n; j++){
                if (D[i][k] != INT_MAX && D[k][j] != INT_MAX &&
D[i][k]+D[k][j] < D[i][j]){
                    D[i][j] = D[i][k]+D[k][j];
                }
            }
        }
    }
}
```

Complejidad del algoritmo de Floyd	
Clasificación del problema	P
Tiempo de ejecución (peor caso)	$O(\|V\|^3)$
Memoria adicional	*Ninguna*

No se requiere memoria adicional, ya que todo se realiza sobre la misma matriz de adyacencias y cuando se encuentra en la iteración k, la columna k y el renglón k no sufren alteraciones.

Si se desea reconstruir la ruta para conseguir esos costos mínimos se requiere ir guardando en una matriz adicional la k cuando minimice pasar por k en lugar de lo que se tiene en ese punto.

5.4 Problema de la mochila

Supón que se tiene un conjunto de n objetos, cada uno de los cuales tienen un peso y un valor asociados a él. Además, se cuenta con una mochila capaz de soportar ciertas unidades de peso sin que se rompa. Se desea guardar en la mochila aquellos objetos de forma tal que maximice el valor acumulado de ellos sin que exceda la capacidad de peso de la mochila. Formalmente:

Formalmente el problema de la mochila

Sea **S** = $\{obj_1, obj_2, \ldots, obj_n\}$

Sea p_i el peso de cada objeto **i** y v_i el valor de cada objeto **i**

Sea P el peso máximo que soporta la mochila.

El problema de la mochila consiste en encontrar un subconjunto **A** de **S** tal que:

• La sumatoria de los v_i de los objetos de **A** sea lo máximo posible.

• Siempre y cuando, la sumatoria de los p_i de los objetos de **A** sea menor o igual a **P**.

La Figura 5.6 muestra un ejemplo del problema de la mochila, en donde con una mochila de capacidad de soportar 30 unidades de peso y 3 objetos:

• Objeto1 con valor de $50 y peso de 5

• Objeto2 con valor de $60 y peso de 10

• Objeto3 con valor de $140 y peso de 20

Lo óptimo es poner en la mochila los objetos 2 y 3, ya que ellos acumularán un valor $200 con un peso 30 unidades. Cualquier otra combinación rompería la mochila o sería menor el valor a acumular.

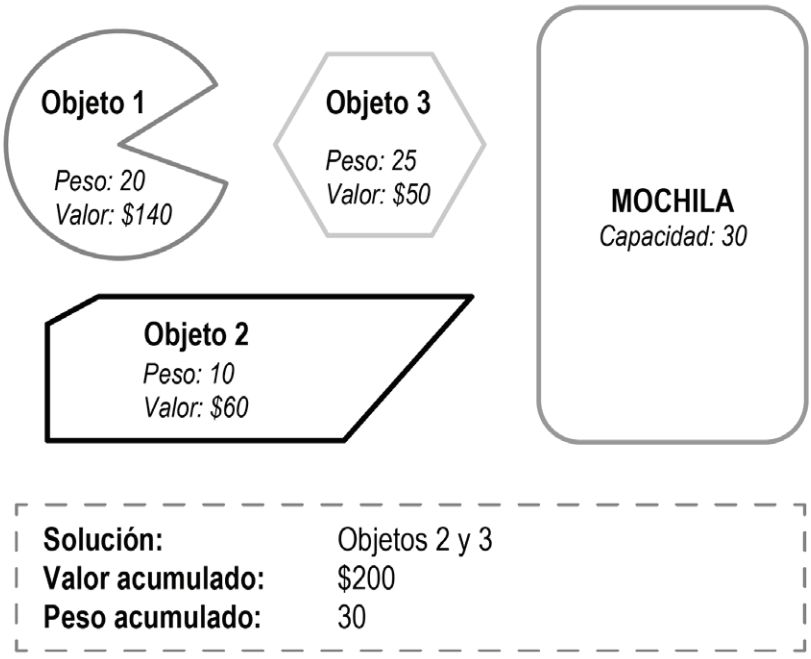


Figura 5.6 Ejemplo del problema de la mochila

5.4.1 Algoritmo con programación dinámica

Para la solución de programación dinámica primero se tiene que pensar en forma recursiva, para el problema de la mochila sería pensar en el último caso, sea A el subconjunto de objetos que maximiza el valor acumulado en la mochila (solución del problema):

- Si A no contiene al objeto n, A es igual a la solución óptima si el problema tuviera solo los primeros n-1 objetos.

- Si A contiene al objeto n, el valor total acumulado de A es igual a $\mathbf{v_n}$ más el valor óptimo del subconjunto formado por los primeros **n-1** objetos con la restricción de que el peso de la mochila no exceda de $\mathbf{P}$-$\boldsymbol{p_n}$ (para respetar el peso del objeto **n**).

Planteamiento de la solución recursiva para programación dinámica

Sea $\mathbf{V}(\boldsymbol{i},\boldsymbol{p})$ el valor acumulado óptimo para los objetos de ***1*** a ***i***, sin exceder al peso ***p***.

La solución a encontrar es $\mathbf{V(n,P)}$, la cual puede obtenerse de la siguiente forma:

$$V(n,P) = \begin{cases} V(n-1,P), & p_n > P \\ \max(V(n-1,P), v_n + V(n-1,P-p_i)), & p_n \leq P \end{cases} \quad (5.4)$$

A su vez esto se generaliza para cualquier i y cualquier p.

El algoritmo de la **mochila con programación dinámica** es:

Algoritmo 5.5 Algoritmo de mochila con programación dinámica

Entrada: vector con los valores y pesos de cada objeto, así como el peso que soporta la mochila.

Salida: el valor máximo que puede guardar la mochila donde la suma de los pesos de los objetos no rompa la mochila.

1. Inicializar una matriz con la cantidad de objetos+1 de renglones y el peso de la mochila+1 de columnas.

2. Para i desde 0 hasta la cantidad de objetos

2.1 Inicializar con 0 la posición de la matriz en el renglón i columna 0.

3. Para j desde 0 hasta el peso de la mochila

3.1 Inicializar con 0 la posición de la matriz en el renglón 0 columna j.

4. Para i desde 1 hasta cantidad de objetos

4.1 Para j desde 1 hasta n

4.1.1 ¿Es el peso del objeto i mayor que j?

4.1.1.1 Sí,

4.1.1.1.1 Asigna a la matriz en la posición [i][j] lo que tiene la matriz en la posición [i-1][j].

4.1.1.2 No,

4.1.1.2.1 Asigna a la matriz en la posición [i][j] el máximo de lo que tiene la matriz en la posición [i-1][j] o lo que contiene la matriz en la posición [i-1][j-peso del objeto i]+valor del objeto i.

5. Regresa el contenido de la matriz en la posición [cantidad de objetos][peso de la mochila].

La implementación del algoritmo de la mochila con programación dinámica quedaría de la siguiente forma:

```
/*
    datos = es el vector de objetos con valor y peso
    PESO  = es el peso que soporta la mochila
*/
struct obj{
    int valor, peso;
};
int mochDP(vector<obj> &datos, int PESO){
    int N = datos.size();
    int mat[N+1][PESO+1];
    for (int i=0; i<=N; i++){
        mat[i][0] = 0;
    }
    for (int j=0; j<=PESO; j++){
        mat[0][j] = 0;
    }
    for (int i=1; i<=N; i++){
        for (int j=1; j<=PESO; j++){
            if (datos[i-1].peso > j){
                mat[i][j] = mat[i-1][j];
            }
            else{
                mat[i][j] = max(mat[i-1][j],
                datos[i-1].valor + mat[i-1][j-datos[i-1].peso]);
            }
        }
    }
    return mat[N][PESO];
}
```

Complejidad del algoritmo de la mochila con programación dinámica	
Clasificación del problema	NP-Complete
Tiempo de ejecución (peor caso)	$O(nP)$
Memoria adicional	$O(nP)$

Se requiere memoria adicional del tamaño de la cantidad de objetos por el peso que soporta la mochila.

5.4.2 Algoritmo de divide y vencerás

Para la solución de **divide y vencerás** sería el mismo planteamiento de programación dinámica, pero en lugar de ir desde el caso base hasta el buscado se llamaría recursivamente.

Algoritmo 5.6 Algoritmo de mochila con divide y vencerás

Entrada: vector con los valores y pesos de cada objeto, así como la N (cantidad de objetos) y P (peso de la mochila) que se están evaluando.

Salida: el valor máximo que puede guardar la mochila con N objetos y el peso de entrada.

1. ¿Es N igual a 0 o el peso del objeto es mayor que P?

   1.1 Sí,

      1.1.1 Regresa 0.

   1.2 No,

      1.2.1 Regresa el valor máximo de la llamada recursiva con N-1 y P o la llamada recursiva con N-1 y P-peso del objeto i + valor del objeto i.

La implementación del algoritmo de la mochila con **divide y vencerás** quedaría de la siguiente forma:

```
/*
    datos = es el vector de objetos con valor y peso
    N     = es la cantidad de objetos.
    PESO  = es el peso que soporta la mochila
*/
struct obj{
    int valor, peso;
};
int mochDyV(vector<obj> &datos, int N, int PESO){
    if (N == 0 || datos[N-1].peso > PESO){
        return 0;
    }
    return max(mochDyV(datos, N-1, PESO),
    datos[N-1].valor+mochDyV(datos, N-1, PESO-datos[N-1].peso));
}
```

Complejidad del algoritmo de la mochila con divide y vencerás	
Clasificación del problema	NP-Complete
Tiempo de ejecución (peor caso)	$O(2^N)$
Memoria adicional	Ninguna

La variante de **divide y vencerás** algunas veces es más rápida que la de programación dinámica, por lo que es importante que, al conocer la cantidad de objetos y el peso de la mochila, se ejecute la menor de **NP** o $\mathbf{2^N}$.

5.4.3 Algoritmo de backtracking

El problema de la mochila al ser un problema de selección puede ser resuelto con la técnica de *backtracking*, pero al ser un problema de optimización tendremos que tomar esto en consideración para poder implementarlo.

Lo primero que hay que diseñar es el árbol de búsqueda de soluciones, el cual tendrá la siguiente conformación:

- El árbol tendrá n+1 niveles como máximo, donde n es la cantidad de objetos.

- Cada nivel indica un objeto a incluir en la mochila.

- Cada nodo tiene dos hijos: el de la izquierda que indica que si incluye al siguiente objeto y el de la derecha que indica que no incluye al siguiente objeto.

- Cada nodo tendrá que almacenar el valor acumulado hasta ese punto, el peso acumulado hasta ese punto y el valor posible a acumular.

El criterio de selección del nodo será:

- Si el peso acumulado de los objetos incluidos no excede a la capacidad de la mochila.

- Si el valor posible a acumular es mayor al mejor valor acumulado hasta ese momento.

Para la estimación del valor posible a acumular se sabe que el nodo del nivel i ya se conoce el peso y valor acumulado, así que se debe analizar cuántos de los siguientes objetos se podrían acumular sin exceder el peso de la mochila asumiendo que el objeto en el nivel **k** es el que excede la mochila. Estratégicamente conviene que los objetos estén ordenados en forma descendente de acuerdo con su valor proporcional ($valor_i$ / $peso_i$). El cálculo del valor posible será:

- El valor acumulado por los objetos ya incluidos, más...

- El valor de los objetos **i+1** hasta **k-1** (que caben sin exceder el peso soportado por la mochila) más…

- El valor proporcional del objeto **k** por el peso que resta de la mochila.

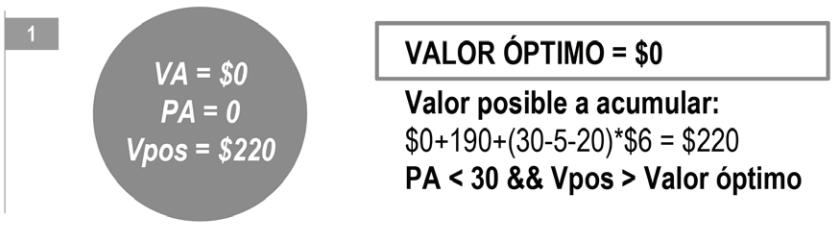
Tomando el ejemplo de la Figura 5.6 en donde se tiene una mochila capaz de soportar 30 unidades de peso y tres objetos:

- Objeto1 con valor de $50 y peso de 5
- Objeto2 con valor de $60 y peso de 10
- Objeto3 con valor de $140 y peso de 20

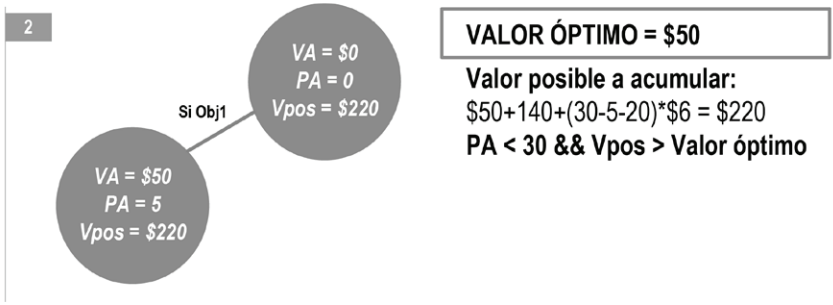
Se tendrá un árbol de máximo 4 niveles después de calcular su valor proporcional y ordenarlos descendentemente se puede calcular la solución con *backtracking*. La Figura 5.7 muestra paso a paso el procedimiento que se realiza con *backtracking* con los siguientes datos de entrada:

PESO MOCHILA = 30
Objeto$_1$, valor$_1$: $50, peso$_1$: 5, valor$_1$/peso$_1$ = $10
Objeto$_3$, valor$_3$: $140, peso$_3$: 20, valor$_3$/peso$_3$ = $7
Objeto$_2$, valor$_2$: $60, peso$_2$: 5, valor$_2$/peso$_2$ = $6

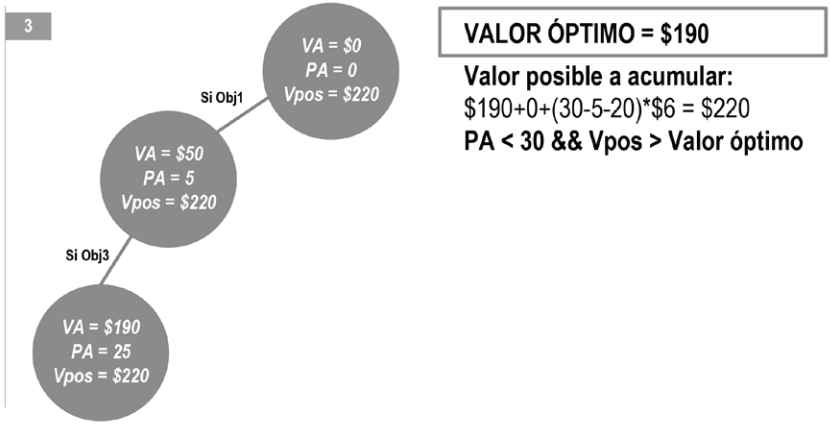
El paso 1 será generar en nodo de arranque.



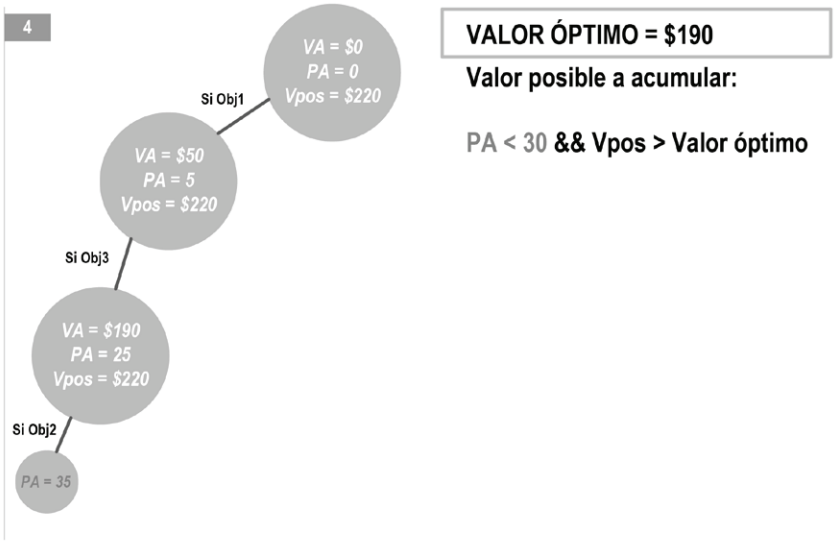
El paso 2 será generar el nodo donde se incluye al Objeto1 y actualiza el valor óptimo.



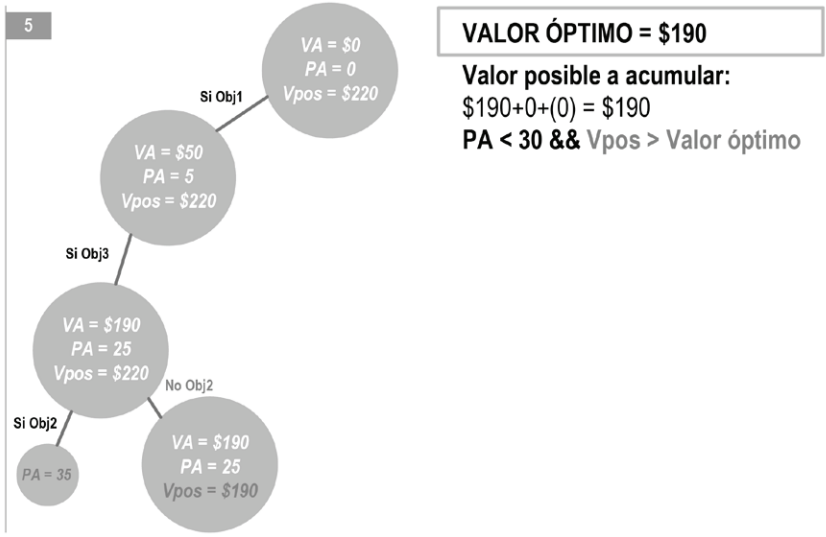
El paso 3 genera el nodo donde se incluye al Objeto3 dado que se incluyó al Objeto1 y actualiza el valor óptimo.



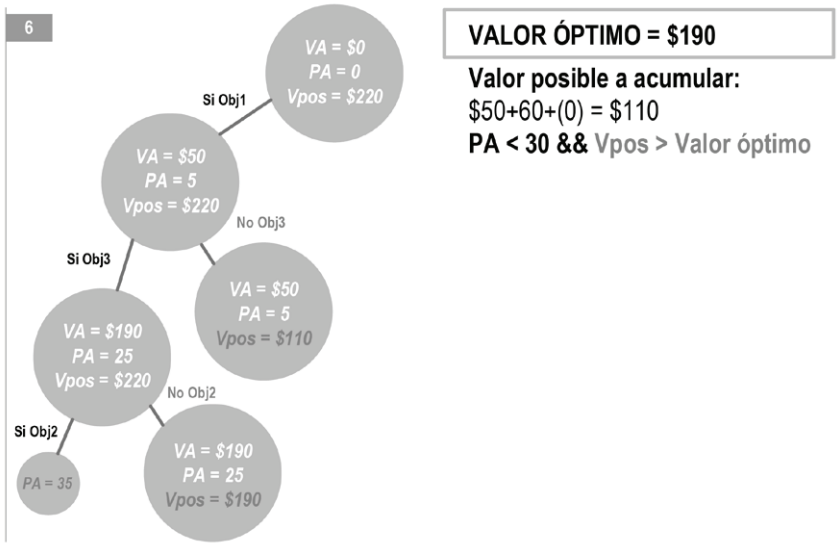
El paso 4 genera el nodo donde se intenta incluir al Objeto2 dado que se incluyó al Objeto3 y Objeto1, pero este nodo sobrepasa el peso de la mochila *(5+10+20 > 30)*.



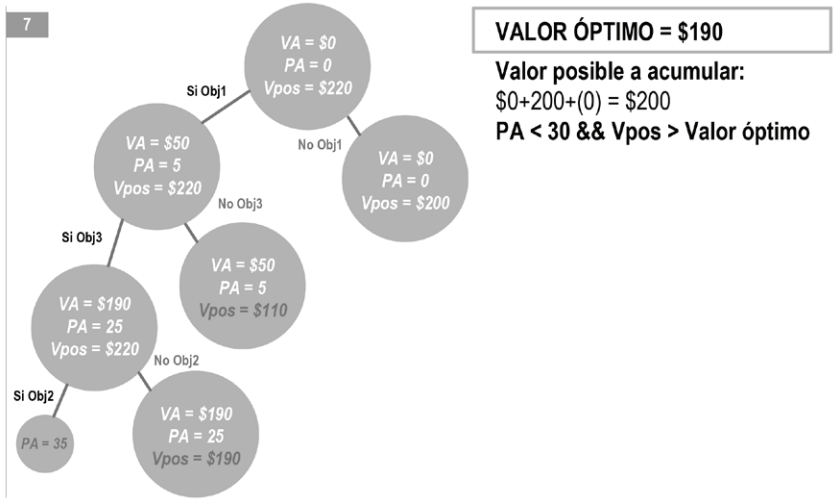
El paso 5 es no incluir al Objeto2 dado que están incluidos los Objetos1 y Objeto3.



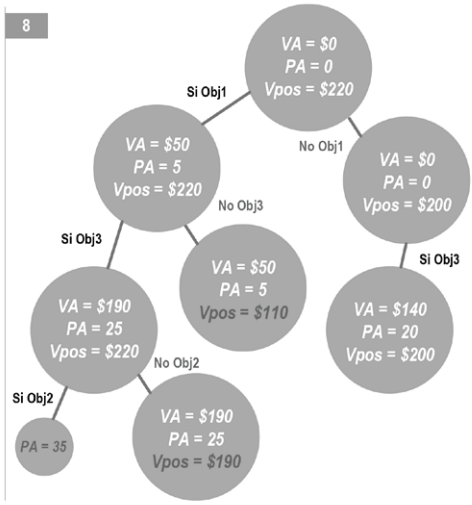
El paso 6 es no incluir al Objeto3 dado que está incluido el Objeto1 y el valor posible no mejora el valor óptimo.



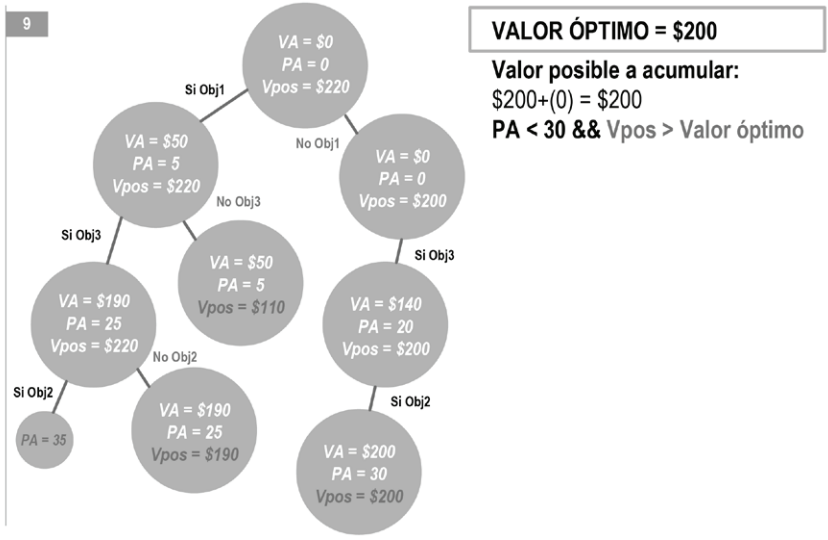
El paso 7 es no incluir al Objeto1.



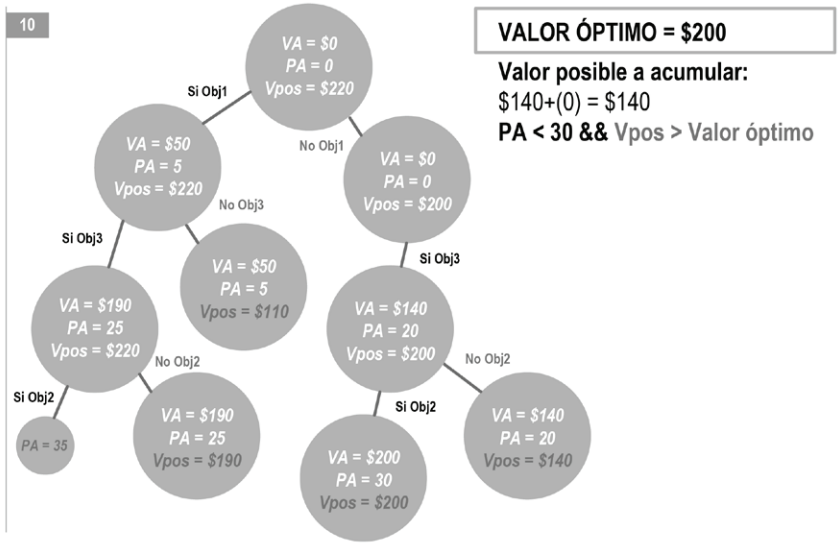
El paso 8 es incluir al Objeto3 dado que no se incluyó el Objeto1.



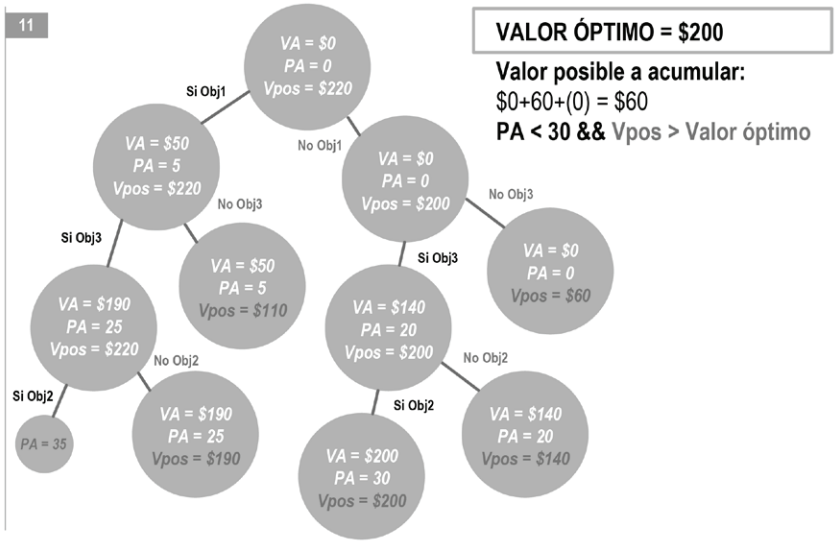
El paso 9 es el de incluir el Objeto2 dado que se incluyó al Objeto3 y no se incluyó al Objeto1, como el valor acumulado mejora el valor óptimo, se actualiza.



El paso 10 es no incluir el Objeto2 dado que se incluyó al Objeto3 y no se incluyó al Objeto1.



El paso 11 es no incluir al Objeto3 dado que no se incluyó al Objeto1.



Como el valor posible es menor que el valor óptimo para el proceso y se obtiene la solución.

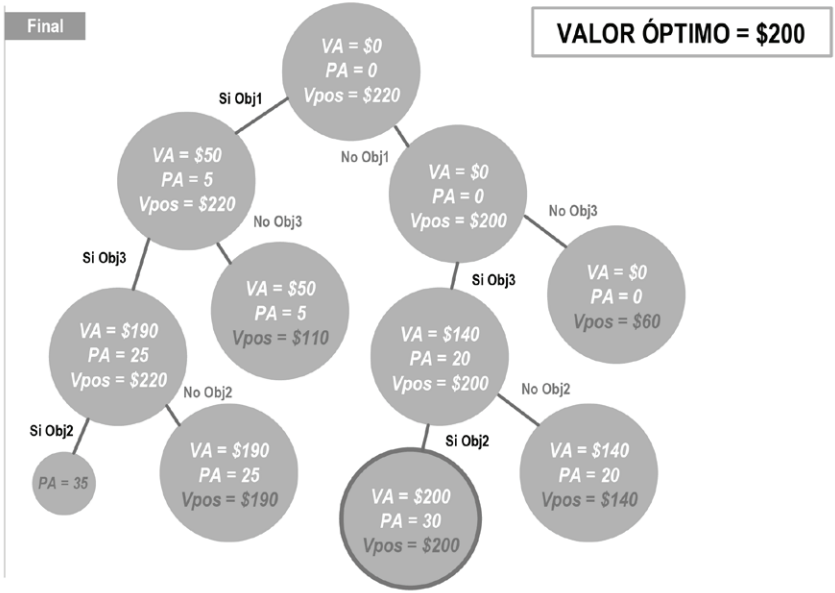


Figura 5.7 Ejemplo paso a paso el problema de la mochila con backtracking

El algoritmo de la ***mochila con backtracking*** es:

Algoritmo 5.7 Algoritmo de mochila con *Backtracking*

Entrada: vector con los valores y pesos de cada objeto ya ordenados descendentemente por su valor proporcional (val_i/$peso_i$), así como el peso que soporta la mochila, el nivel en que va, así como el nodo (valor acumulado, peso acumulado y valor posible).

Salida: el valor máximo que puede guardar la mochila donde la suma de los pesos de los objetos no rompa la mochila.

1. ¿El peso acumulado es menor o igual al peso de la mochila y el valor posible es mejor que el valor óptimo hasta el momento?

1.1 Sí,

1.1.1 ¿El valor acumulado es mejor que el óptimo?

1.1.1.1 Sí,

1.1.1.1.1 Actualizar el valor óptimo con el valor acumulado.

1.1.2 Calcular el valor acumulado, peso acumulado y valor posible incluyendo al siguiente objeto.

1.1.3 Llamar recursivamente con estos datos.

1.1.4 Calcular el valor acumulado, peso acumulado y valor posible sin incluir al siguiente objeto.

1.1.5 Llamar recursivamente con estos datos.

La implementación del algoritmo de la **mochila con *backtracking*** quedaría de la siguiente forma:

```
// n = Cant Objetos
// PESO = Peso de la Mochila
// VOptimo = Valor Optimo del proceso
int PESO;
int vOptimo;
int n;
// Registro de Objetos
struct obj{
    int val;
    int peso;
    float valpeso;
};

// datos = Los objetos de entrada
vector <obj> datos;

// Comparador para ordenar por valpos en forma descendente
bool my_cmp(const obj &a, const obj & b){
    return a.valpeso > b.valpeso;
}
// Método de BackTracking para la Mochila
// i = objeto o nivel en el que voy
// va = El valor Acumulado
// pa = El Peso Acumulado
// vp = El Valor Posible
void bt (int i, int va, int pa, int vp){
    if (i < n && pa <= PESO && vp > vOptimo){
        if (va > vOptimo){
            vOptimo = va;
        }
        // Si al sig Obj (i+1)
        bt(i+1, va+datos[i+1].val, pa+datos[i+1].peso, vp);
        // No al sig Obj (i+1)
        int vpAux = va;
        int pesoAux = pa;
        int k=i+1;
        while (k<n && pesoAux+datos[k].peso <= PESO){
            vpAux += datos[k].val;
            pesoAux += datos[k].peso;
            k++;
        }
        if (k < n){
            vpAux += ((PESO-pesoAux)*datos[k].valpeso);
        }
        bt(i+1, va, pa, vpAux);
        }
}
```

Complejidad del algoritmo de la mochila con *backtracking*	
Clasificación del problema	NP-Complete
Tiempo de ejecución (peor caso)	$O(2^n)$
Memoria adicional	Ninguna

Si se quisiera saber los objetos que conforman ese valor óptimo hay que ir almacenando cuáles se incluyen y cuáles no cada vez que se actualiza el valor óptimo.

5.4.4 Algoritmo de ramificación y poda (Branch & Bound)

El problema de la mochila al ser un problema de selección con optimización cumple con los requisitos para poder ser resuelto con la técnica de *Branch & Bound*. El árbol de búsqueda de soluciones será idéntico al de *backtracking*, donde se tendrán n+1 niveles y cada nivel se trabajará con un objeto, el cual será ordenado descendentemente por valor proporcional ($valor_i$ / $peso_i$) y cada nodo tendrá 2 hijos: el de la derecha, donde si incluye al siguiente objeto y el de la derecha donde no lo incluye. Los nodos almacenarán el valor y peso acumulados hasta el momento, así como su valor posible (el cual se calcula igual que *backtracking*). La gran diferencia es que en *backtracking* se va procediendo a profundidad y en *Branch & Bound* se trabaja en anchura con *Best-First*, esto es se maneja una fila priorizada con prioridad mayor valor posible y se saca al mejor y se generan sus dos hijos.

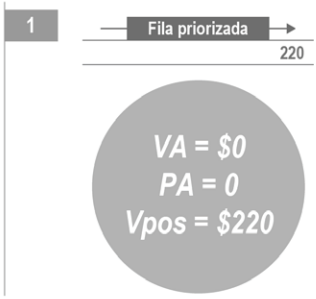
Tomando el ejemplo de la Figura 5.6 en donde se tiene una mochila capaz de soportar 30 unidades de peso y tres objetos:

- Objeto1 con valor de $50 y peso de 5
- Objeto2 con valor de $60 y peso de 10
- Objeto3 con valor de $140 y peso de 20

Se tendrá un árbol de máximo 4 niveles, después de calcular su valor por unidad de peso y ordenarlos descendentemente se puede calcular la solución con *Branch & Bound*. La Figura 5.8 muestra paso a paso el procedimiento que se realiza con *Branch & Bound*.

PESO MOCHILA = 30
Objeto$_1$, valor$_1$: $50, peso$_1$: 5, valor$_1$/peso$_1$ = $10
Objeto$_3$, valor$_3$: $140, peso$_3$: 20, valor$_3$/peso$_3$ = $7
Objeto$_2$, valor$_2$: $60, peso$_2$: 5, valor$_2$/peso$_2$ = $6

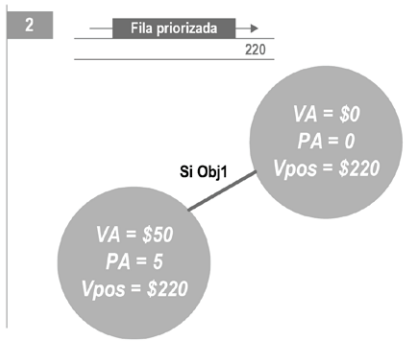
El paso 1 será generar en nodo de arranque.



VALOR ÓPTIMO = $0

Valor posible a acumular:
$0+190+(30-5-20)*$6 = $220
PA < 30 && Vpos > Valor óptimo

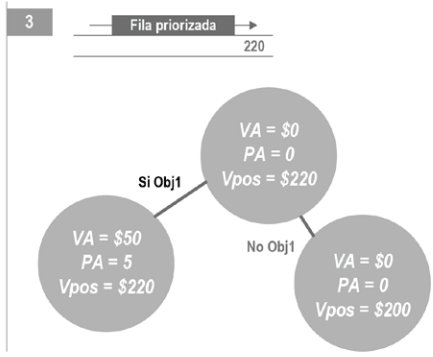
El paso 2 saca de la pila priorizada el valor del frente y genera el nodo donde se incluye al Objeto1 y actualiza el valor óptimo metiendo su valor posible a la fila priorizada.



VALOR ÓPTIMO = $50

Valor posible a acumular:
$50+140+(30-5-20)*$6 = $220
PA < 30 && Vpos > Valor óptimo

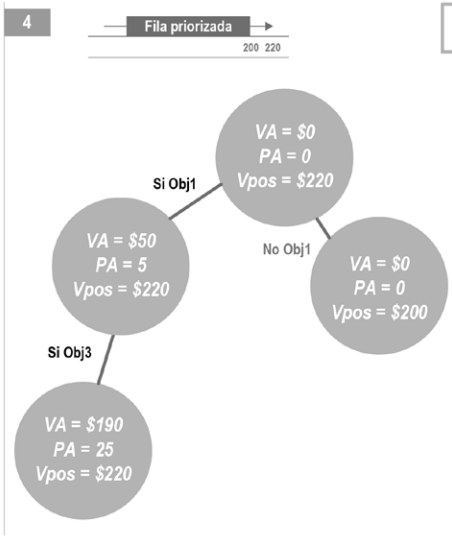
El paso 3 es no incluir al nodo que no incluye al Objeto1 y meter el valor posible a la fila priorizada.



VALOR ÓPTIMO = $50

Valor posible a acumular:
$0+200+(0) = $200
PA < 30 && Vpos > Valor óptimo

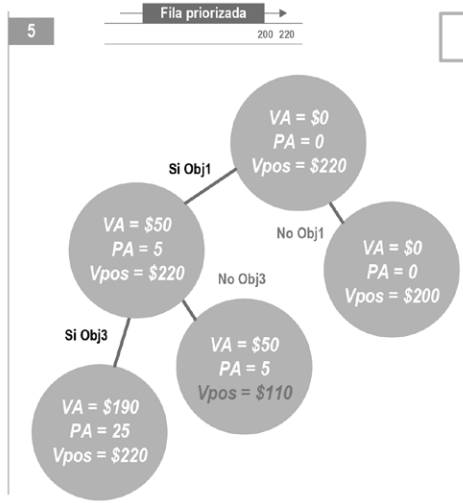
El paso 4 saca de la fila priorizada el valor del frente y genera el nodo donde incluye al Objeto3 dado que se incluyó al Objeto1 y actualiza el valor óptimo con el valor acumulado de este nodo y mete su valor posible a la fila priorizada.



VALOR ÓPTIMO = $190

Valor posible a acumular:
$190+(30-5-20)*$6 = $220
PA < 30 && Vpos > Valor óptimo

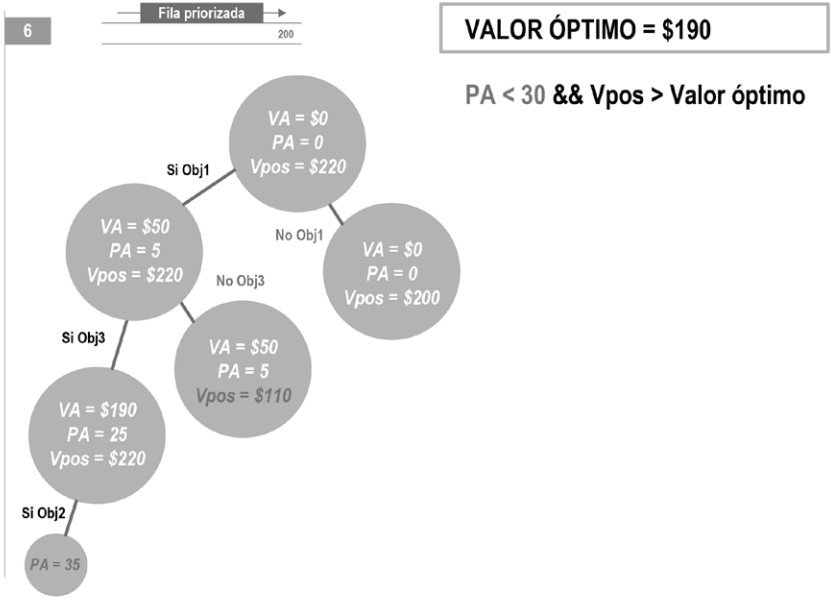
El paso 5 genera el nodo donde no incluye al Objeto3 dado que, sí incluyó al Objeto1, como el valor posible no mejor el valor óptimo ahí poda esa rama.



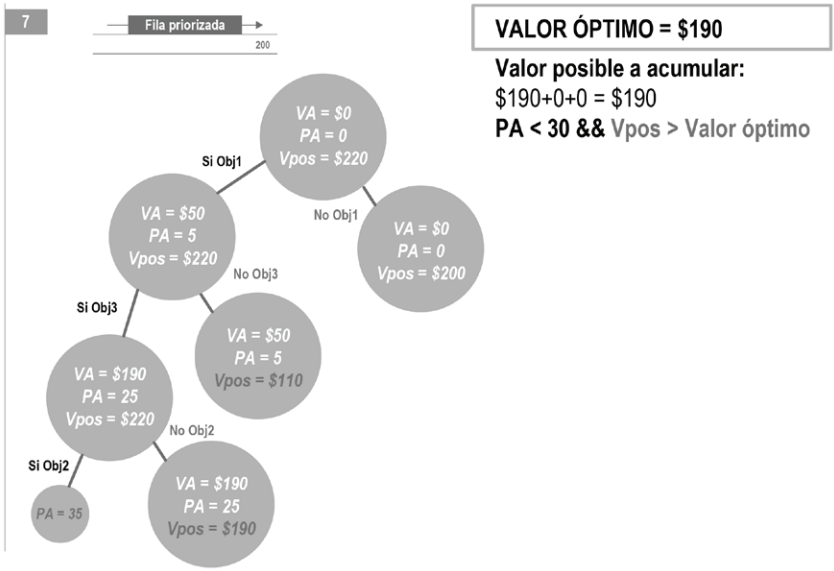
VALOR ÓPTIMO = $190

Valor posible a acumular:
$50+60+0 = $110
PA < 30 && Vpos > Valor óptimo

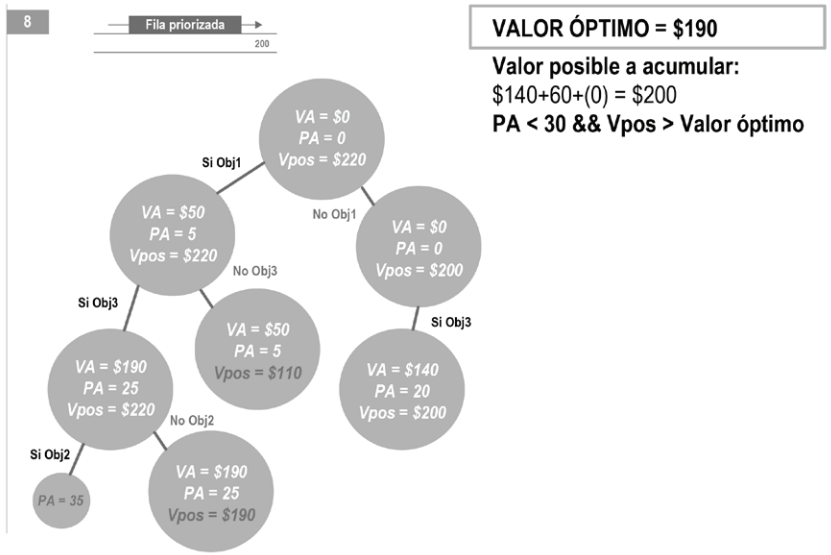
El paso 6 saca de la fila priorizada el valor del frente y genera el nodo donde se incluye al Objeto2, dado que se incluyó al Objeto1 y Objeto3, pero este nodo sobrepasa el peso de la mochila (5+10+20 > 30).



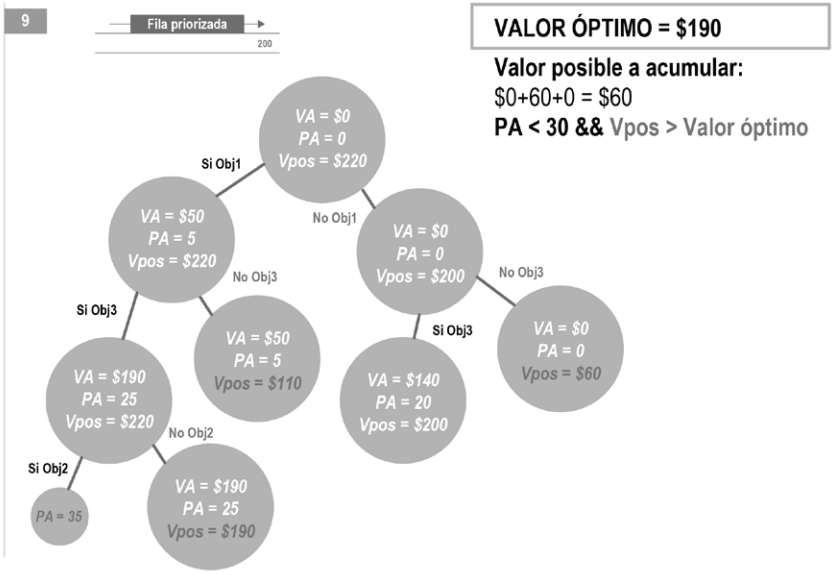
El paso 7 genera el nodo donde no incluye al Objeto2 dado que incluyó a los Objetos1 y Objeto3.



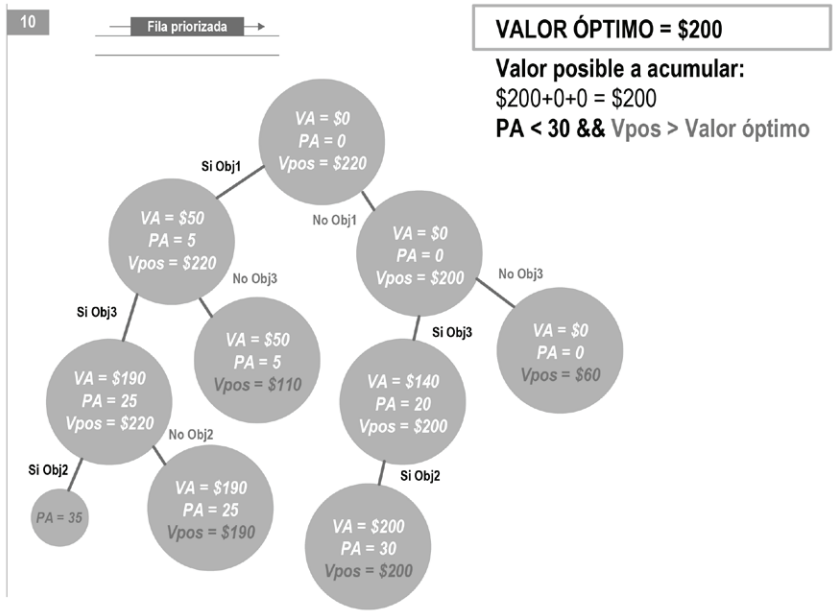
El paso 8 saca de la fila priorizada el valor de enfrente y genera el nodo donde incluye al Objeto3 dado que no incluyó al Objeto1, como el valor posible es mejor que el valor óptimo lo mete a la fila priorizada.



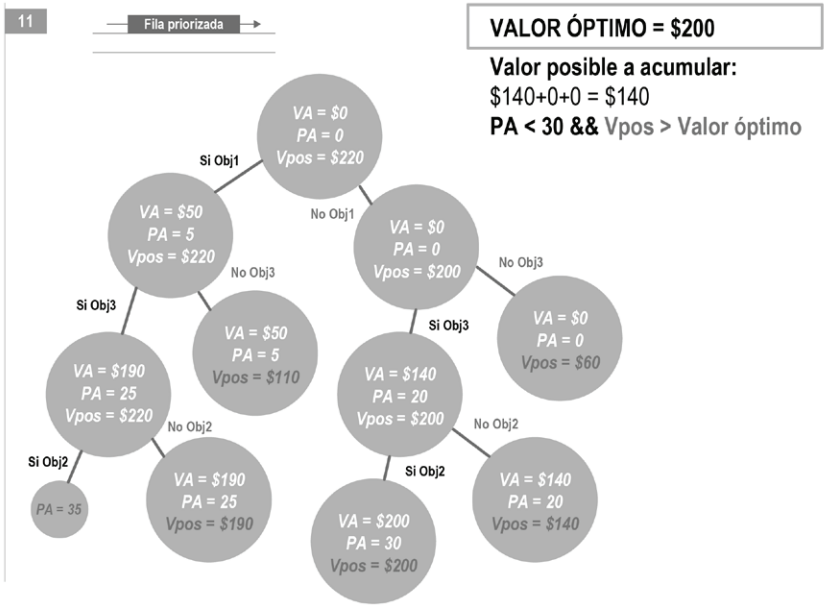
El paso 9 genera el nodo donde no incluye el Objeto 3 dado que no se incluyó el Objeto1, como el valor posible es menor que el valor óptimo ahí poda esa rama.



El paso 10 saca de la fila priorizada el valor del frente y genera el nodo donde incluye al Objeto2 dado que incluyó al Objeto3 y no incluyó al Objeto1, como el valor acumulado mejora el valor óptimo actualiza el valor óptimo con este valor acumulado.



El paso 11 es no incluir el Objeto2 dado que se incluyó al Objeto3 y no se incluyó al Objeto1.



Como ya la fila priorizada está vacía, ahí termina la solución.

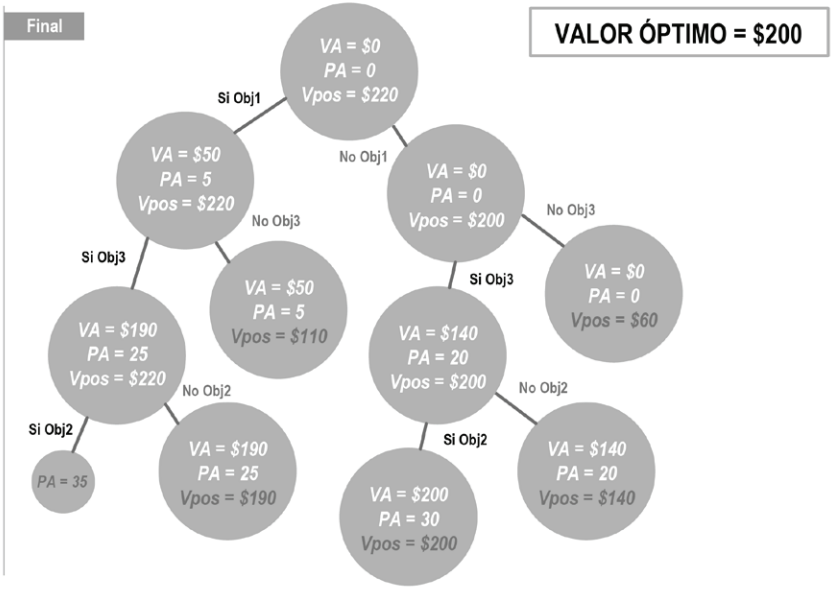


Figura 5.8 Paso a paso el problema de la mochila con Branch & Bound

El algoritmo de la **mochila con *Branch & Bound*** es:

Algoritmo 5.8 Algoritmo de mochila con *Branch & Bound*.

Entrada: vector con los valores y pesos de cada objeto ya ordenados descendentemente por su valor proporcional (val_i/$peso_i$), así como el peso que soporta la mochila.

Salida: el valor máximo que puede guardar la mochila donde la suma de los pesos de los objetos no rompa la mochila.

1. Generar una fila priorizada de nodos (nivel, valor acumulado, peso acumulado, valor posible), donde la prioridad es mayor valor posible.

2. Calcular el valor posible de arranque y meterlo a la fila priorizada.

3. Mientras la fila priorizada no esté vacía.

3.1 Sacar el nodo del frente de la fila priorizada

3.2 ¿El valor acumulado de este objeto es mayor que el valor óptimo?

3.2.1 Sí,

3.2.1.1 Actualizar el valor óptimo con el valor acumulado.

3.3 ¿El valor posible es mayor que el valor óptimo?

3.3.1 Sí,

3.3.1.1 Calcular el nodo que incluye al siguiente objeto y meterlo a la fila priorizada.

3.3.1.2 Calcular el nodo que no incluye al siguiente objeto y meterlo a la fila priorizada.

La implementación del algoritmo de la **mochila con *Branch & Bound*** quedaría de la siguiente forma:

```
int PESO;      // Peso de la Mochila
int vOptimo;   // Valor Óptimo
int n;         // Cantidad de Objetos
struct node{
    int niv;   // Nivel del nodo
    int va;    // Valor acumulado
    int pa;    // Peso acumulado
    float vp; // Valor por unidad de peso
    bool operator<(const node &otro) const{
        return vp < otro.vp;
    }
};
// Registro de Objetos
struct obj{
    int valor, peso;
    float valpeso;
};
// datos = Los objetos de entrada
vector<obj> datos;
// Comparador para ordenar por valpos en forma descendente
bool my_cmp(const obj& a, const obj& b){
    return a.valpeso > b.valpeso;
}
// Función para calcular el valor pósible
float calculaVP(int i, float vpAux, int pesoAux){
    int k = i+1;
    while (k < n && pesoAux+datos[k].peso <= PESO){
        vpAux += datos[k].valor;
        pesoAux += datos[k].peso;
        k++;
    }
    if (k < n){
        vpAux += ((PESO-pesoAux)*datos[k].valpeso);
    }
    return vpAux;
}
void BB(){
    priority_queue<node> pq;
    node arranque;
    arranque.niv = -1;
    arranque.va = 0;
    arranque.pa = 0;
    arranque.vp = calculaVP(-1, 0, 0);
    pq.push(arranque);
    while (!pq.empty()){
        arranque = pq.top();
        pq.pop();
        if (arranque.va > vOptimo){
            vOptimo = arranque.va;
            res.clear();
```

```
51              for (int j=0; j<n; j++){
52                  res.push_back(arranque.inc[j]);
53              }
54          }
55          if (arranque.vp > vOptimo){
56              arranque.niv++;
57              // No incluyo al sig obj
58              arranque.vp = calculaVP(arranque.niv, arranque.va,
    arranque.pa);
59              if (arranque.vp > vOptimo && arranque.pa <= PESO){
60                  pq.push(arranque);
61              }
62              // Si incluyo al sig obj
63              arranque.va += datos[arranque.niv].valor;
64              arranque.pa += datos[arranque.niv].peso;
65              arranque.vp = calculaVP(arranque.niv, arranque.va,
    arranque.pa);
66              if (arranque.vp > vOptimo && arranque.pa <= PESO){
67                  pq.push(arranque);
68              }
69          }
70      }
71 }
```

Complejidad del algoritmo de la mochila con *Branch & Bound*	
Clasificación del problema	NP-Complete
Tiempo de ejecución (peor caso)	$O(2^n)$
Memoria adicional	Ninguna

Si se quisiera saber los objetos que conforman ese valor óptimo hay que ir almacenando cuáles se incluyen y cuáles no cada vez que se actualiza el valor óptimo.

5.5 Problema del viajero

El problema del viajero o TSP por sus siglas en inglés (*Travelling Saleman Problem*), es que, dado un grafo ponderado y no dirigido se deberá encontrar el ciclo hamiltoniano con el menor costo, es decir, salir de un vértice, visitar todos los vértices restantes una sola vez y regresar al mismo punto de partida con el menor costo. En la Figura 5.9 se puede ver un ejemplo de un grafo dirigido y ponderado en donde se tienen 6 ciclos hamiltonianos.

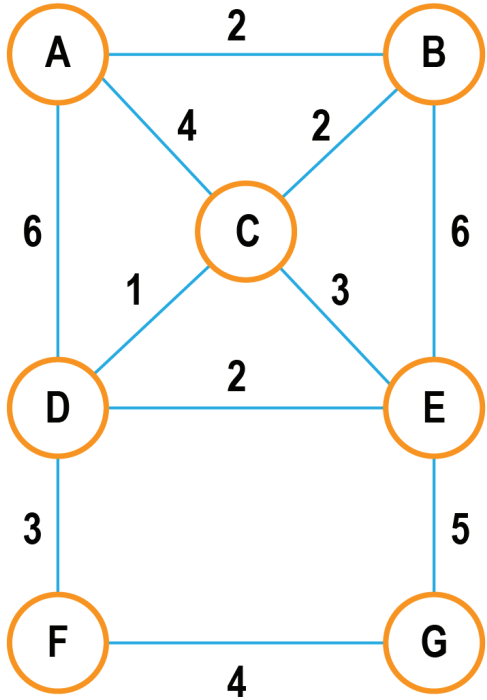


Figura 5.9 Ejemplo del grafo no dirigido y ponderado

Los 6 ciclos hamiltonianos son:

- A-B-C-E-G-F-D-A con un costo de 25.
- A-B-E-G-F-D-C-A con un costo de 25.
- A-C-B-E-G-F-D-A con un costo de 30.
- A-C-D-F-G-E-B-A con un costo de 25.
- A-D-F-E-G-B-C-A con un costo de 30.
- A-D-F-E-G-C-B-A con un costo de 25.

La solución del problema del viajero pude ser cualquiera de los ciclos hamiltonianos que tengan un costo de 25.

5.5.1 Algoritmo con ramificación y poda (Branch & Bound)

El problema del Viajero, al ser un problema de selección con optimización cumple con los requisitos para poder ser resuelto con la técnica de *Branch & Bound*. El árbol de búsqueda de soluciones tendrá **n** niveles, y cada nivel se trabajará un nodo a conectar y cada nodo tendrá ***n-1-i*** hijos donde ***i*** es el nivel en que va diciendo los posibles nodos que falten en esa ruta. Los nodos almacenarán el costo acumulado hasta el momento, así como su costo posible, el último nodo visitado y el nodo actual. La idea es trabajar en anchura con *Best-First*, esto es: maneja una fila priorizada con prioridad menor costo posible, se saca al mejor y se generan sus hijos.

El cálculo del costo posible se realiza de la siguiente forma:

- Costo acumulado, más…

- Del último vértice alcanzado: escoger el menor costo con los vértices faltantes, más…

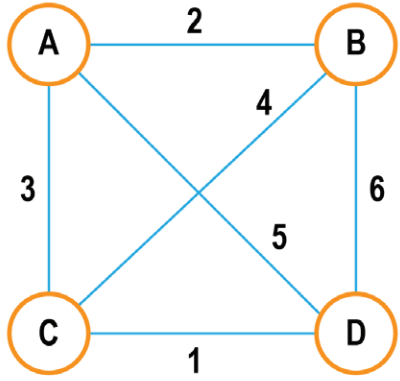
- Por cada vértice faltante: escoger el menor costo entre ellos y llegar al vértice inicial

El criterio de selección del nodo será:

- Si el vértice en el nivel *i* del árbol es **adyacente** al vértice en el nivel *i-1* del camino correspondiente en el árbol.

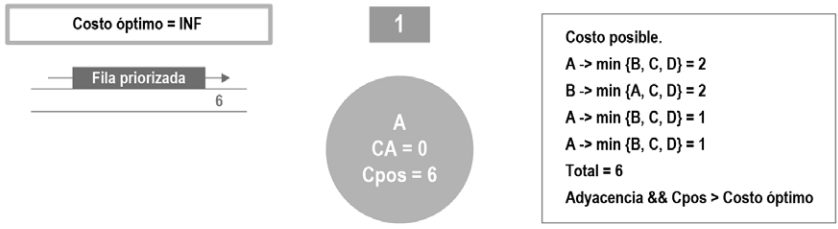
- Si el costo posible a acumular al expandir el nodo *i* es **menor** al mejor costo acumulado hasta ese momento.

Se debe llevar una variable global de costo mínimo, el cual se inicializa con infinito y se modifica cuando se obtenga un ciclo hamiltoniano que mejore ese costo. La Figura 5.10 muestra un grafo y su matriz de adyacencias, así como la solución para el problema del viajero con *Branch & Bound* paso a paso.

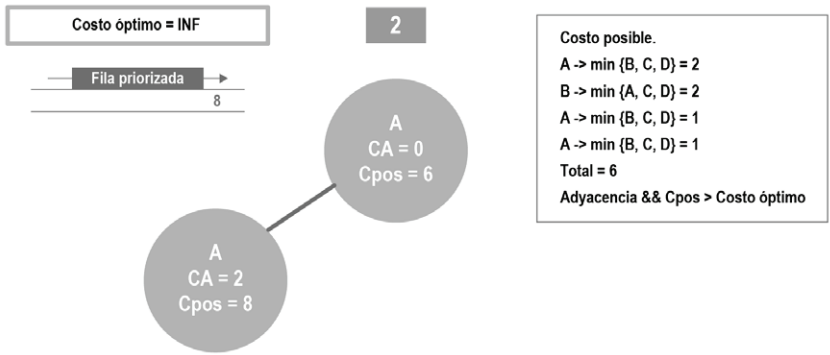


	A	B	C	D
A	0	2	3	5
B	2	0	4	6
C	3	4	0	1
D	5	6	1	0

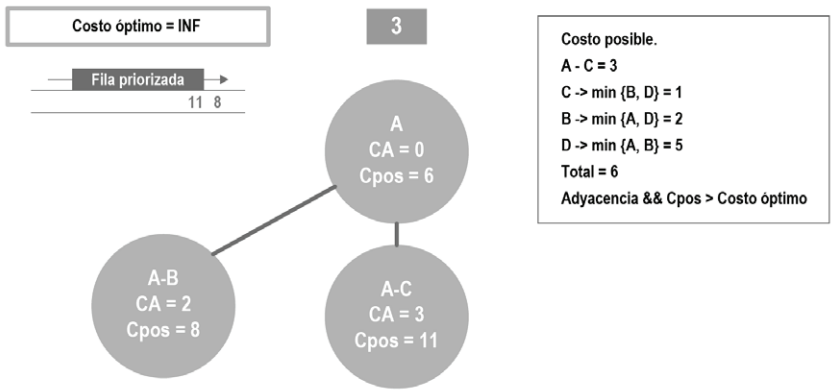
El paso 1 genera el nodo inicial, cuyo costo posible lo mete a la fila priorizada.



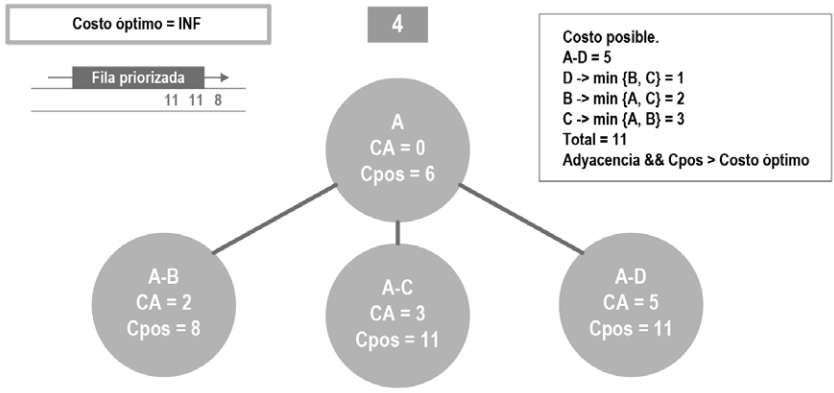
El paso 2 saca de la fila priorizada el valor del frente, genera el nodo de la conexión A-B y mete el costo posible en la fila priorizada.



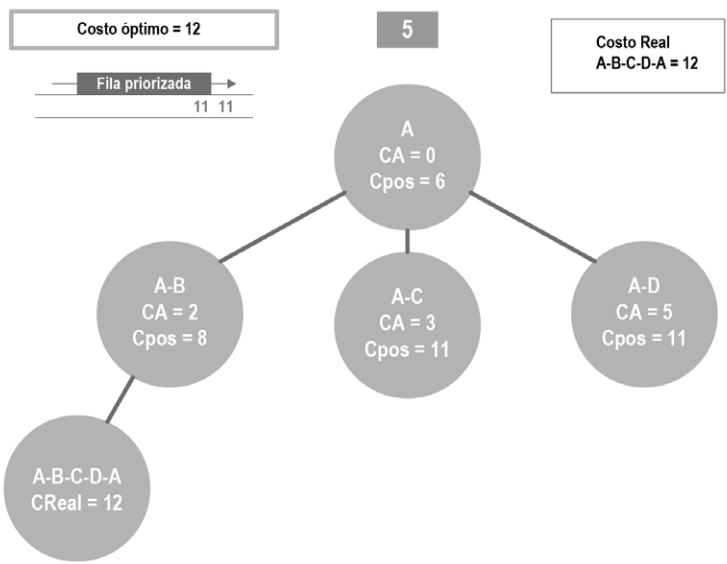
El paso 3 genera el nodo de la conexión A-C y mete el costo posible en la fila priorizada.



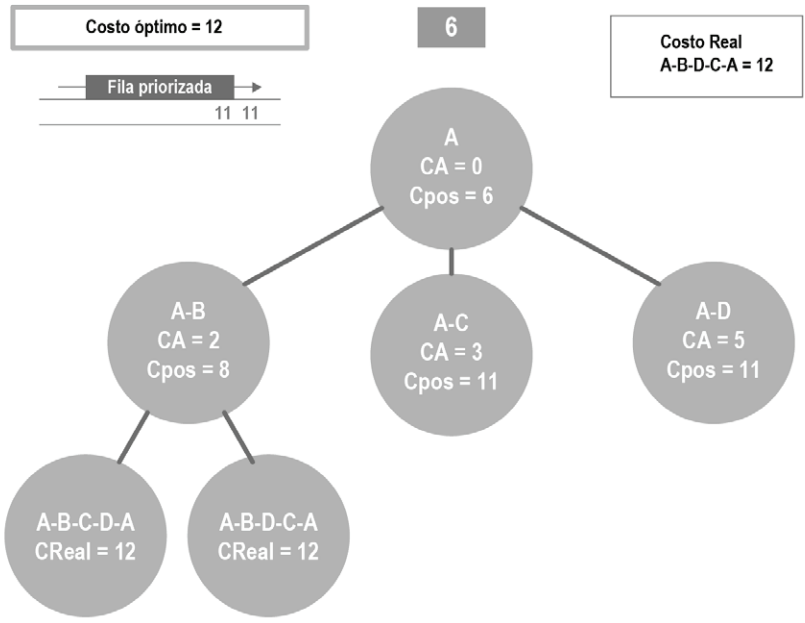
El paso 4 genera el nodo de la conexión A-D y mete el costo posible en la fila priorizada.



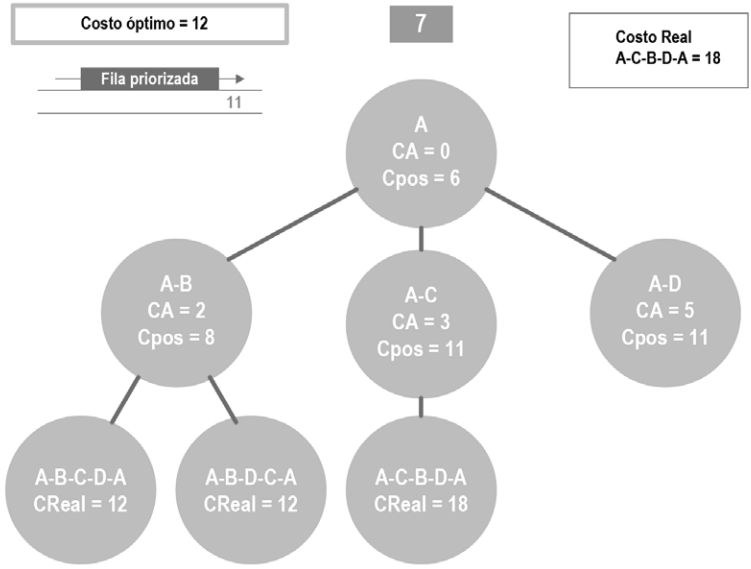
El paso 5 saca de la fila priorizada el valor del frente y genera el nodo de la conexión A-B-C, como este solo tendría 1 hijo, lo absorbe generando el nodo A-B-C-D-A que ya es un ciclo hamiltoniano completo y mejora el costo óptimo actualizándolo con su costo acumulado.



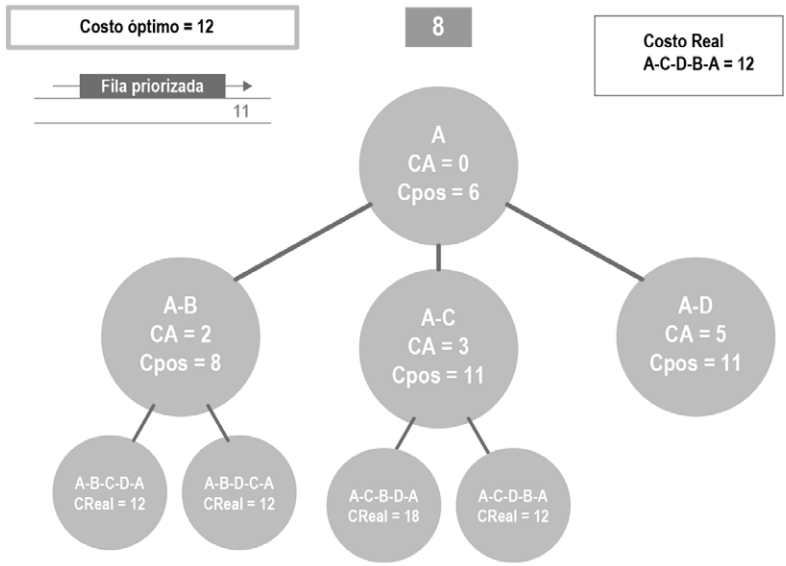
El paso 6 genera el nodo de la conexión A-B-D, como este solo tendría 1 hijo, lo absorbe generando el nodo A-B-D-C-A que ya es un ciclo hamiltoniano completo y no mejora el costo óptimo.



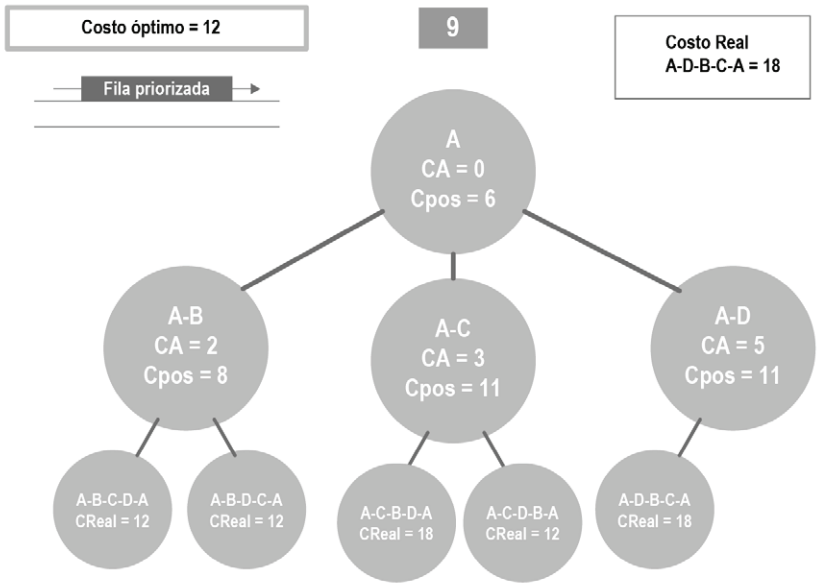
El paso 7 saca de la fila priorizada el valor del frente y genera el nodo de la conexión A-C-B, como este solo tendría 1 hijo, lo absorbe generando el nodo A-C-B-D-A que ya es un ciclo hamiltoniano completo y no mejora el costo óptimo.



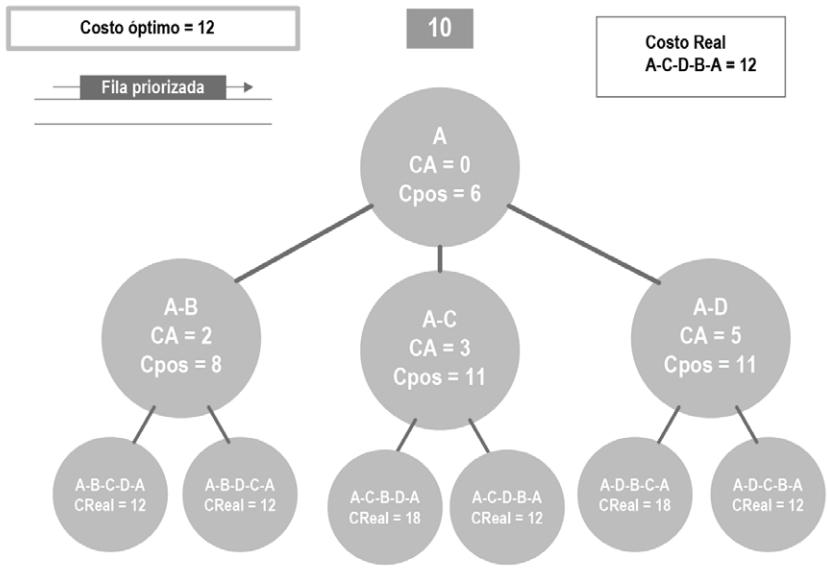
El paso 8 genera el nodo de la conexión A-C-D, como este solo tendría 1 hijo, lo absorbe generando el nodo A-C-D-B-A que ya es un ciclo hamiltoniano completo y no mejora el costo óptimo.



El paso 9 saca de la fila priorizada el valor del frente y genera el nodo de la conexión A-D-B, como este solo tendría 1 hijo, lo absorbe generando el nodo A-D-B-C-A que ya es un ciclo hamiltoniano completo y no mejora el costo óptimo.



El paso 10 genera el nodo de la conexión A-D-C, como este solo tendría 1 hijo, lo absorbe generando el nodo A-D-C-B-A que ya es un ciclo hamiltoniano completo y no mejora el costo óptimo.



Aunque existen varias soluciones con el mismo costo de 12, el que se encontró primero será el que sea la solución.

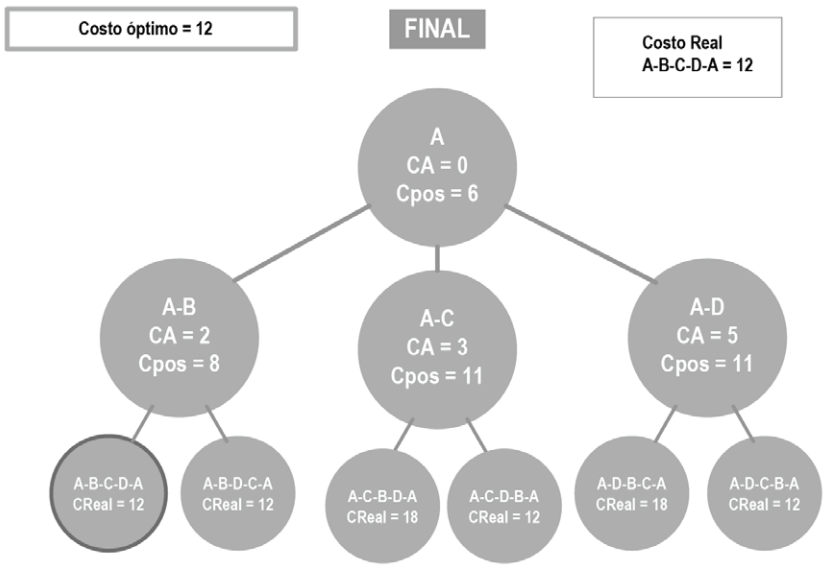


Figura 5.10 Ejemplo paso a paso del viajero con Branch & Bound

El algoritmo del **problema del viajero con *Branch & Bound*** es:

Algoritmo 5.9 Algoritmo del problema del viajero con *Branch & Bound*.

Entrada: matriz de adyacencia del grafo.

Salida: el costo mínimo del ciclo hamiltoniano del grafo.

1. Generar una fila priorizada de nodos (nivel, costo acumulado, costo posible, último nodo visitado, nodo actual), donde la prioridad es menor costo posible.

2. Calcular el costo posible de arranque y meterlo a la fila priorizada.

3. Mientras la fila priorizada no esté vacía.

3.1 Sacar el nodo del frente de la fila priorizada

3.2 ¿El costo posible es menor que el costo óptimo?

3.2.1 Sí,

3.2.1.1 Para cada conexión del nodo.

3.2.1.1.1 ¿No ha sido visitado?

3.2.1.1.1.1 Sí,

3.2.1.1.1.1.1 Calcula el nodo.

3.2.1.1.1.1.2 ¿Es la última conexión?

3.2.1.1.1.2.1 Sí,

3.2.1.1.1.2.1.1 Calcular el valor acumulado real.

3.2.1.1.1.1.2.1.2 Si mejora es mejor el costo real nuevo que el costo óptimo, se actualiza.

3.2.1.1.1.1.2.2 No,

3.2.1.1.1.1.2.2.1 Meter el nodo a la fila priorizada.

4. Regresar el costo óptimo.

La implementación del algoritmo del **viajero con *Branch & Bound*** quedaría de la siguiente forma:

```
struct Nodo{
    int iNivel;         // Nivel
    int iCostoAcum;     // Costo Real Acumulado
    int iCostoPos;      // Costo Posible
    int iVisitedAnt;    // Último vértice visitado
    int iVisitedNow;    // Vértice actual
    bool bVisited[21]; // Arreglo de ciudades visitadas
    bool operator < (const Nodo& aux) const{
        return iCostoPos >= aux.iCostoPos;
    }
};

int iN;                         // Cantidad de ciudades
int iMatDist[21][21];           // Matriz de Adyacencia
int iMinCost = 1000000;         // Costo óptimo
priority_queue<Nodo> qNodos;  // Fila priorizada de Nodos

// Función para calcular el valor acumulado
void calcularValorAcum (Nodo &nodoActual){
    nodoActual.iCostoAcum +=
  iMatDist[nodoActual.iVisitedAnt][nodoActual.iVisitedNow];
}

// Función para calcular el valor pósible
void calcularValorPosible (Nodo &nodoActual){
    nodoActual.iCostoPos = nodoActual.iCostoAcum;
    int iNivel = nodoActual.iNivel, iNodoAct =
  nodoActual.iVisitedNow, iTemp = 1000000;
    for (int iI = 1, iK = iN - iNivel; iI <= iN && iK >= 0; iI++) {
        if (!nodoActual.bVisited[iI] || iI == iNodoAct){
            if (!nodoActual.bVisited[iI]) {
                for (int iJ = 1; iJ <= iN; iJ++){
                    if (iI != iJ && (!nodoActual.bVisited[iJ] || iJ
  == 1)) {
                        iTemp = min(iTemp, iMatDist[iI][iJ]);
                    }
```

```
34                  }
35              }
36              else if (iI == iNodoAct) {
37                  for (int iJ = 1; iJ <= iN; iJ++){
38                      if (!nodoActual.bVisited[iJ]){
39                          iTemp = min(iTemp, iMatDist[iI][iJ]);
40                      }
41                  }
42              }
43              nodoActual.iCostoPos += iTemp;
44              iTemp = DISTMIN;
45              iK--;
46          }
47      }
48 }
49
50 // Función para calcular el valor final
51 void calcularValorFinal (Nodo &nodoActual){
52      int iLastNode = -1;
53      calcularValorAcum(nodoActual);
54      for (int iI = 1; iI <= iN && iLastNode == -1; iI++)
55          if (!nodoActual.bVisited[iI])
56              iLastNode = iI;
57      nodoActual.iCostoAcum +=
   iMatDist[nodoActual.iVisitedNow][iLastNode];
58      nodoActual.iCostoAcum += iMatDist[iLastNode][1];
59 }
60
61 // Función del Viajero con B&B
62 void branchAndBound (Nodo raiz){
63      Nodo nodoAux, nodoHijo;
64      qNodos.push(raiz);
65      while (!qNodos.empty()){
66          nodoAux = qNodos.top();
67          qNodos.pop();
68          if (nodoAux.iCostoPos <= iMinCost) {
69              for (int iI = 1; iI <= iN; iI++){
70                  if (!nodoAux.bVisited[iI]) {
71                      nodoHijo.iNivel = nodoAux.iNivel + 1;
72                      memcpy(nodoHijo.bVisited, nodoAux.bVisited,
   (iN+1) * sizeof(bool));
73                      nodoHijo.bVisited[iI] = true;
74                      nodoHijo.iVisitedAnt = nodoAux.iVisitedNow;
75                      nodoHijo.iVisitedNow = iI;
76                      nodoHijo.iCostoAcum = nodoAux.iCostoAcum;
77                      if (nodoHijo.iNivel == iN-2) {
78                          calcularValorFinal(nodoHijo);
79                          iMinCost = min(iMinCost,
   nodoHijo.iCostoAcum);
80                      }
81                      else {
82                          calcularValorAcum(nodoHijo);
83                          calcularValorPosible(nodoHijo);
84                          qNodos.push(nodoHijo);
85                      }
86                  }
87              }
88          }
89      }
90 }
```

Complejidad del algoritmo del viajero con *Branch & Bound*	
Clasificación del problema	NP-Complete
Tiempo de ejecución (peor caso)	$O(2^n)$
Memoria adicional	Ninguna

Si se quisiera saber el ciclo hamiltoniano que conforman ese costo óptimo hay que guardar la ruta cada vez que se actualiza el costo óptimo.

5.6 Árbol de mínima expansión

El problema del árbol de mínima expansión (MST por sus siglas en inglés, *Minimum Spanning Tree*) es un conjunto de arcos (la cantidad de vértices menos 1 arco) de un grafo no dirigido, ponderado y conectado, que conecta a todos los vértices juntos, sin ciclos y con la mínima suma de pesos de los arcos. La Figura 5.11 muestra un grafo no dirigido, ponderado y conectado, con las líneas naranjas se puede ver la solución del árbol de mínima expansión.

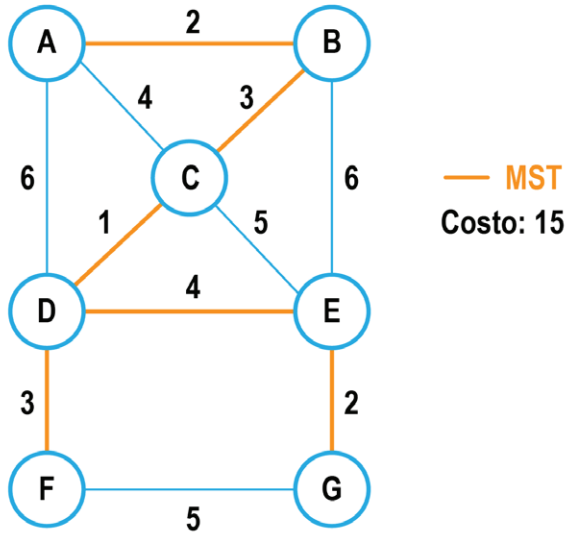


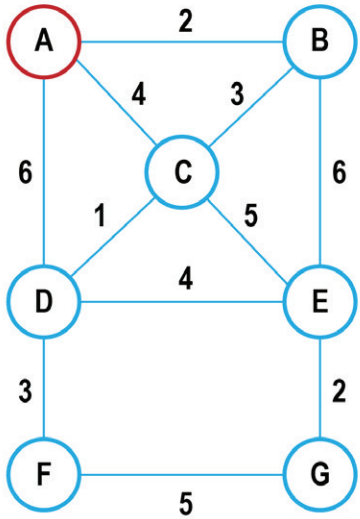
Figura 5.11 Ejemplo del árbol de mínima expansión

5.6.1 Algoritmo de Prim

Este es un algoritmo voraz que fue diseñado en 1930 por Vojtech Jarnik, luego de forma independiente fue diseñado en 1957 por Robert Prim y redescubierto por Dijkstra en 1959, es por esta razón que también es conocido como algoritmo DJP o algoritmo de Jarnik.

El algoritmo empieza por un vértice como parte de un conjunto solución y va seleccionado el arco con menos costo que vaya de uno de los vértices que están en el conjunto solución a algún vértice faltante por conectar almacenando este arco en un conjunto de arcos de la solución. Cuando en el conjunto solución se encuentran los n vértices, se encuentra el árbol de mínima expansión (MST). La Figura 5.12 tiene un ejemplo paso a paro del algoritmo de **Prim**.

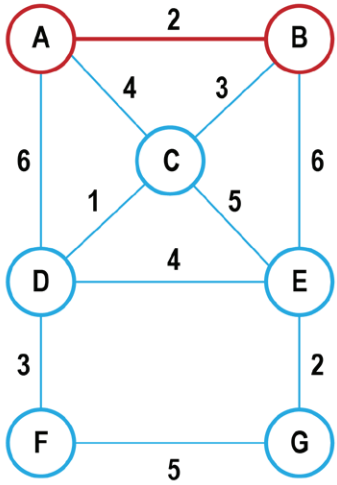
El paso 1 arranca de un vértice, para el ejemplo se toma como arranque el vértice A.



1

S = { }
Y = {A}
V-Y = {B, C, D, E, F, G}

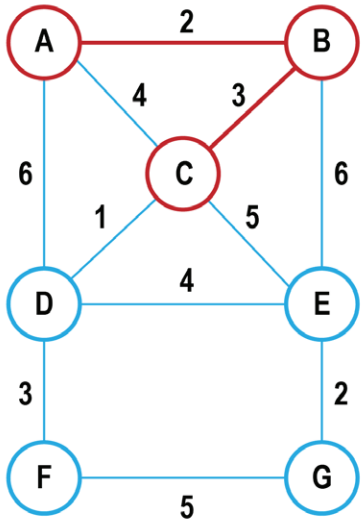
El paso 2 selecciona el arco de menor costo saliendo de A y es el A-B con un costo de 2.



2

S = {(A-B)}
Y = {A, B}
V-Y = {C, D, E, F, G}

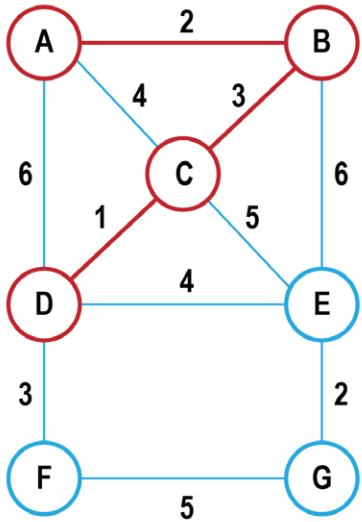
El paso 3 selecciona el arco de menor costo saliendo de A o B hacia el resto de los vértices y es el arco A-C con un costo de 3.



3

S = {(A-B), (B-C)}
Y = {A, B, C}
V-Y = {D, E, F, G}

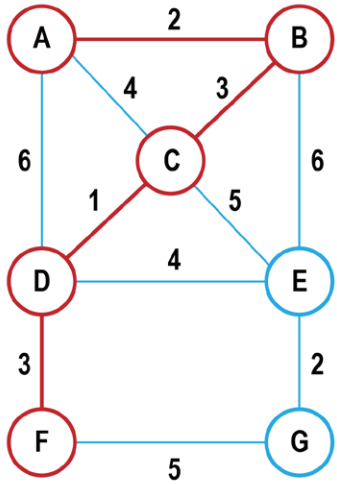
El paso 4 selecciona el arco de menor costo saliendo de A, B o C hacia el resto de los vértices y es el arco A-D con un costo de 1.



4

S = {(A-B), (B-C), (C-D)}
Y = {A, B, C, D}
V-Y = {E, F, G}

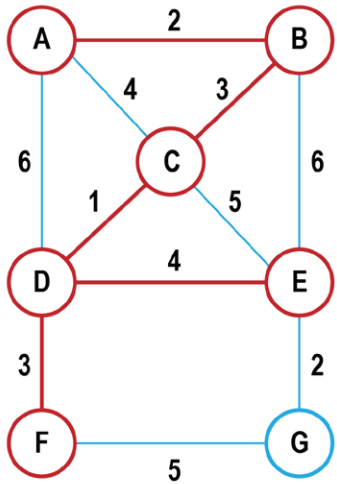
El paso 5 selecciona el arco de menor costo saliendo de A, B, C o D hacia el resto de los vértices y es el arco D-F con un costo de 3.



5

S = {(A-B), (B-C), (C-D), (D-F)}
Y = {A, B, C, D, F}
V-Y = {E, G}

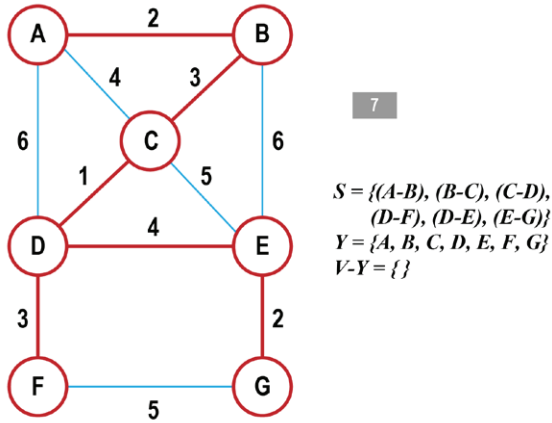
El paso 6 selecciona el arco de menor costo saliendo de A, B, C, D o F hacia el resto de los vértices y es el arco D-E con un costo de 4.



6

S = {(A-B), (B-C), (C-D), (D-F), (D-E)}
Y = {A, B, C, D, E, F}
V-Y = {G}

El paso 7 selecciona el arco de menor costo saliendo de A, B, C, D, E o F hacia el último de los vértices y es el arco E-G con un costo de 2.



La solución del árbol de mínima expansión con el algoritmo de **Prim** queda con un costo de 15 teniendo el siguiente árbol como solución.

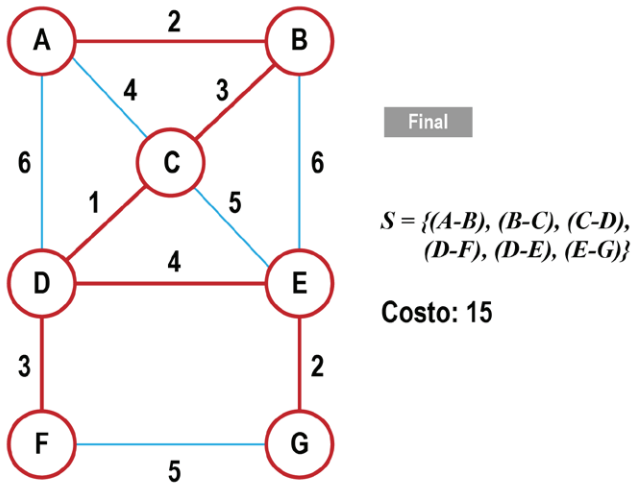


Figura 5.12 Ejemplo paso a paso del algoritmo de Prim

El algoritmo de **Prim** es:

Algoritmo 5.10 Algoritmo de Prim.

Entrada: lista de adyacencia del grafo.

Salida: el costo del árbol de mínima expansión.

1. Inicializar el costo con 0.

2. Generar un conjunto de seleccionados y meter al vértice de arranque.

3. Generar un conjunto de faltantes y meter a todos los vértices excepto al de arranque.

4. Inicializar una variable de conectados con la cantidad de vértices menos 1.

5. Mientras conectados no llegue a 0.

5.1 Inicializar la variable de costo mínimo con infinito.

5.2 Pada cada vértice dentro del conjunto de seleccionados.

5.2.1 Para cada vecino del vértice del conjunto seleccionados

5.2.1.1 ¿Falta por seleccionar ese vecino y su costo es menor a la variable de costo mínimo?

5.2.1.1.1 Sí,

5.2.1.1.1.1 La variable de costo mínimo y asignar el valor de la conexión de ese arco.

5.2.1.1.1.2 La variable vértice seleccionado asignarle el vecino.

5.2.2 Actualizar el costo sumándole el costo del arco seleccionado como mínimo

5.2.3 Agregar la variable vértice seleccionado al conjunto de seleccionados y borrarla del conjunto de faltantes.

6. Regresar el costo del MST.

La implementación del algoritmo del algoritmo **Prim** quedaría de la siguiente forma:

```
// adjList es la lista de adjacencia, donde almacena un par
// * el first del par es el vértice vecino
// * el second del par es el costo del arco.
int primMST(vector<vector<pair<int, int>>> &adjList){
        int costMSTPrim = 0;
        unordered_set<int> selected;
        unordered_set<int> missing;
        selected.insert(0);
        for (int i=1; i<V; i++){
                missing.insert(i);
        }
        int conected = V-1, minCost, selVertex;
        while (conected){
                minCost = INT_MAX;
                for (auto it1=selected.begin(); it1 !=
  selected.end(); ++it1){
                        for (auto it2=adjList[*it1].begin(); it2 !=
  adjList[*it1].end(); ++it2){
                                if (missing.find((*it2).first) !=
  missing.end() && (*it2).second < minCost){
                                        minCost = (*it2).second;
                                        selVertex = (*it2).first;
                                }
                        }
                }
                costMSTPrim += minCost;
                selected.insert(selVertex);
                missing.erase(selVertex);
                conected--;
        }
        return costMSTPrim;
}
```

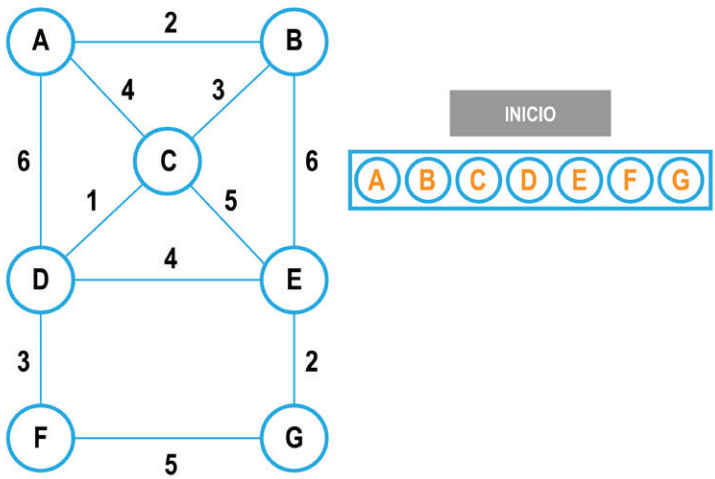
Complejidad del Algoritmo de Prim	
Clasificación del problema	P
Tiempo de ejecución (peor caso)	$O(V^2)$
Memoria adicional	O(V)

Se requiere memoria adicional para ir almacenando los vértices seleccionados y faltantes. Si se quisiera saber los arcos que logran el árbol de mínima expansión se requiere ir almacenándolos cada vez que son seleccionados.

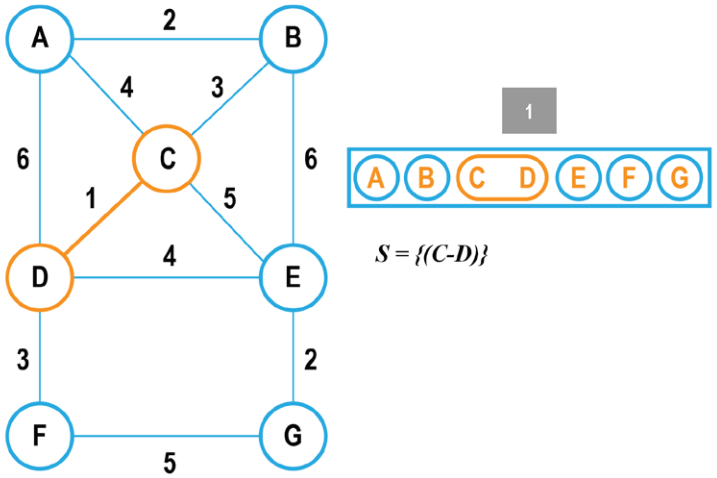
5.6.2 Algoritmo de Kruskal

Este es un algoritmo voraz que fue publicado en 1956 por Joseph Kruskal. El algoritmo empieza ordenando los arcos en forma ascendente según su costo, va seleccionando de menor a mayor verificando que al conectarlo no forme ciclo, en este caso lo integra como parte de la solución, en caso contrario lo desecha, cuando logra llegar a todos los vértices termina su ejecución. Al utilizar un conjunto disjunto (Union-*Find*) comprueba si hay o no ciclo. La Figura 5.13 tiene un ejemplo paso a paso del algoritmo de **Kruskal**.

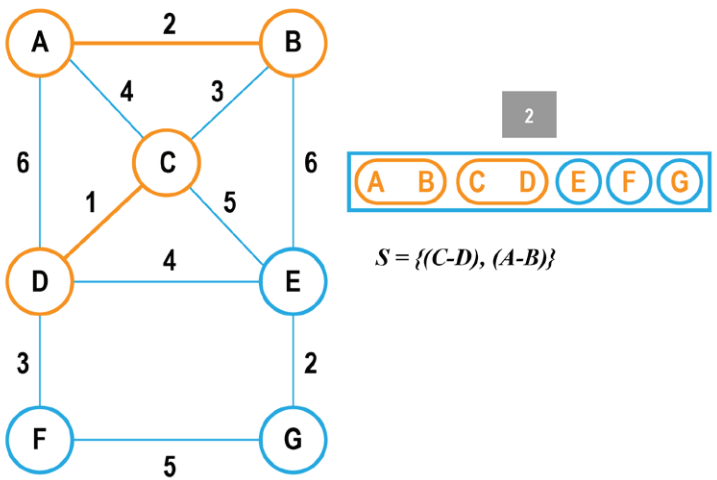
El paso inicial es realizar el conjunto disjunto de todos los vértices.



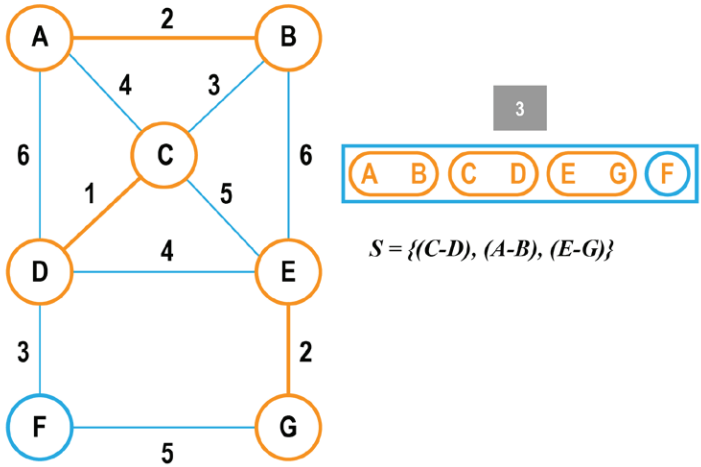
El paso 1 selecciona el arco (C-D) y une los vértices C y D en el conjunto disjunto.



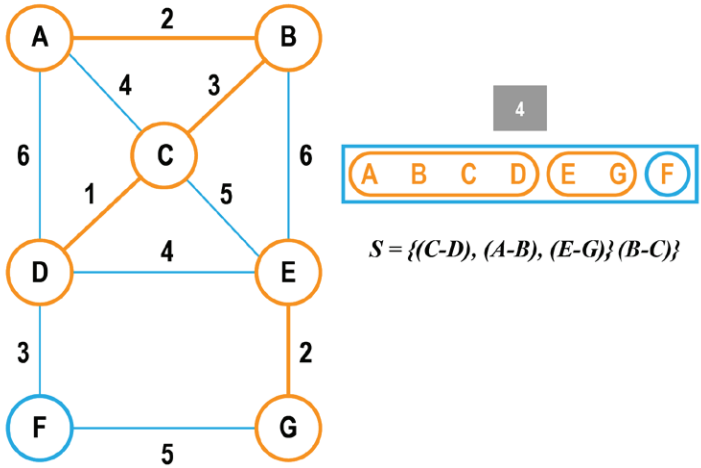
El paso 2 selecciona el arco (A-B) y une los vértices A y B en el conjunto disjunto.



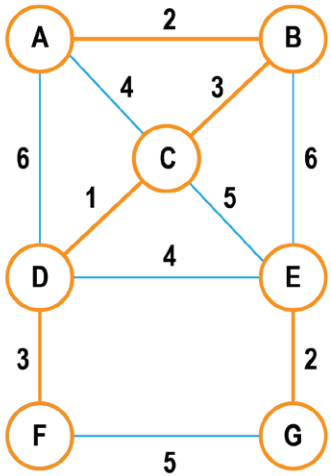
El paso 3 selecciona el arco (E-G) y une los vértices E y G en el conjunto disjunto.



El paso 4 selecciona el arco (B-C) y une los vértices A, B, C y D en el conjunto disjunto.

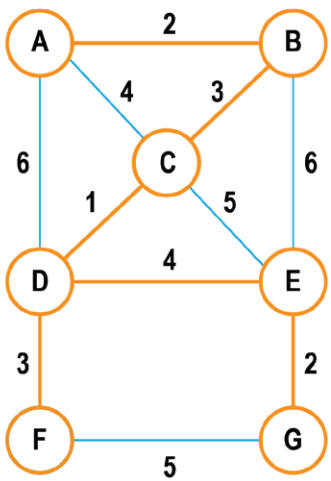


El paso 5 selecciona el arco (D-F) y une los vértices A, B, C, D y F en el conjunto disjunto.



S = {(C-D), (A-B), (E-G)} (B-C), (D-F)}

El paso 6 selecciona el arco (D-E) y une los vértices A, B, C, D, E, F y G en el conjunto disjunto.



S = {(C-D), (A-B), (E-G)} (B-C), (D-F), (D-E)}

La solución del árbol de mínima expansión con el algoritmo de **Kruskal** queda con un costo de 15 teniendo el siguiente árbol como solución.

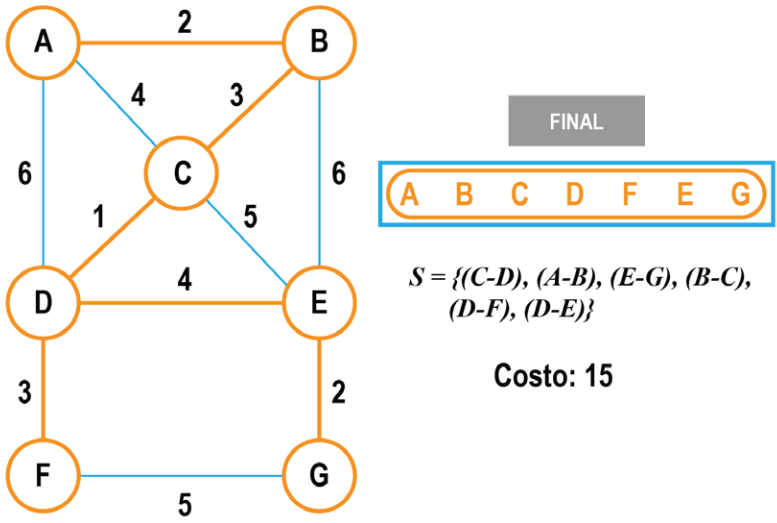


Figura 5.13 Ejemplo paso a paso del algoritmo de Kruskal

El algoritmo de Kruskal es:

Algoritmo 5.11 Algoritmo de Kruskal.

Entrada: lista de arcos del grafo.

Salida: el costo del árbol de mínima expansión.

1. Ordenar los arcos ascendentemente por peso del arco.

2. Generar un conjunto disjunto y agregar a todos los vértices.

3. Inicializar el costo del MST con 0.

4. Para cada arco previamente.

    4.1.1 Buscar el conjunto de cada vértice en conjunto disjunto.

    4.1.2 ¿Son diferentes conjuntos?

        4.1.2.1 Sí,

            4.1.2.1.1 Aumentar el costo del MST con el costo del arco.

            4.1.2.1.2 Juntar los dos conjuntos de los vértices.

5. Regresar el costo del MST.

La implementación del algoritmo del algoritmo **Kruskal** quedaría de la siguiente forma:

```
// edges es un vector de pares.
// * el first del par es el costo del arco,
//   esto sirve para que el sort ordene con este criterio.
// * el second es un par de los vértices que unen ese arco.
int kruskalMST(vector< pair<int, pair<int, int>> > &edges){
    int costMST = 0;
    sort (edges.begin(), edges.end());
    DisjointSets ds(V);
    for (auto it=edges.begin(); it!=edges.end(); it++){
        int u = it->second.first;
        int v = it->second.second;
        int set_u = ds.find(u);
        int set_v = ds.find(v);
        if (set_u != set_v){
            costMST += it->first;
            ds.merge(u, v);
        }
    }
    return costMST;
}
```

Complejidad del Algoritmo de Kruskal.	
Clasificación del problema	P
Tiempo de ejecución (peor caso)	$O(E\log_2 E)$
Memoria adicional	O(V)

Se requiere memoria adicional para el manejo del conjunto disjunto de vértices. Si se quisiera saber los arcos que logran el árbol de mínima expansión se requiere ir almacenándolos cada vez que son seleccionados.

5.7 Problema de la multiplicación encadenada de matrices

Para poder multiplicar dos matrices se tiene que cumplir la regla de que la cantidad de columnas de la primera matriz coincida con la cantidad de renglones de la segunda matriz, esto es, si se quisiera multiplicar **A x B** y las dimensiones de **A** fueran de 4 x 5 y las de **B** fueran de 4 x 5 no se podrían multiplicar, en cambio si las de **A** fueran de 4 x 3 y las de **B** fueran de 3 x 5 sí se podrían multiplicar. Generalizando, cuando se desee multiplicar dos matrices **A x B**, las dimensiones de **A** deben de ser ***m x n*** y las de **B** deben de ser ***n x p***, las dimensiones de la matriz resultante de la multiplicación serían de los dos extremos, o sea de los renglones de **A** por las columnas de **B**, por lo tanto, **C** sería la matriz resultante con dimensiones de ***m x p***, debido a esta regla, la multiplicación de matrices no tiene la propiedad conmutativa, esto es, no se puede intercambiar el orden de la multiplicación. Para poder obtener esta matriz resultante se requieren n multiplicaciones escalares por cada celda, por lo tanto, se requieren **m x n x p** multiplicaciones escalares para la obtener la matriz resultante.

El problema es que teniendo una serie matrices que se desean multiplicar en forma encadenada, por ejemplo, si se quisiera multiplicar:

M = A x B x C x D

Con las siguientes dimensiones:

- A de 8x5 (8 renglones y 5 columnas)
- B de 5x4 (5 renglones y 4 columnas)
- C de 4x10 (4 renglones y 10 columnas)
- D de 10x6 (10 columnas y 6 columnas)

La matriz resultante M será de dimensiones de 8x6 (8 renglones y 6 columnas). Si se multiplicara de izquierda derecha, esto es:

(((A x B) x C) x D)

Por lo tanto, se tendrían:

- **AxB** serían *8x5x4* multiplicaciones escalares = **160 +**
- La matriz resultante anterior **xC** serían *8x4x10* = **320 +**
- La matriz resultante anterior **xD** serían *8x10x6* = **480**

Dando un total de 160+320+480 = **960** multiplicaciones escalares.

Como se conoce que la multiplicación de matrices si tiene la propiedad asociativa podríamos asociarla de diferente forma, por ejemplo, podría ser:

((A x B) x (C x D))

Con la cual se tendrían:

- **AxB** serían *8x5x4* multiplicaciones escalares = **160 +**
- **CxD** serían *4x10x6* multiplicaciones escalares = **240 +**
- Al multiplicar las dos matrices resultantes serían *8x4x6* = **192**

Dando un total de 160+240+192 = **592** multiplicaciones escalares, lo cual sería mucho más eficiente que la forma tradicional.

El problema de la multiplicación encadenada de matrices es encontrar la forma de asociar las multiplicaciones de forma tal que sea el mínimo de multiplicaciones escalares, por lo tanto, la entrada del problema son las dimensiones de las matrices y no los datos de ellas, ya que no se realizará la multiplicación de matrices.

5.7.1 Algoritmo de Godbole

En 1973, Godbole publicó el algoritmo para la multiplicación encadenada de matrices utilizando la técnica de programación dinámica, donde se dice que el problema es tamaño n porque se tienen n matrices que multiplicar en forma encadenada:

$$\mathbf{M_1 \times M_2 \times \ldots \times M_{n-1} \times M_n}$$

Cuyas dimensiones son almacenadas en un vector ***d*** de tamaño n+1, donde las dimensiones de cada matriz son almacenadas en las posiciones $\mathbf{d_{i-1} \times d_i}$ por lo tanto las dimensiones de las matrices quedarían almacenadas:

- Las dimensiones de $\mathbf{M_1}$ estarán en $d_o \times d_1$.
- Las dimensiones de $\mathbf{M_2}$ estarán en $d_1 \times d_2$.
- Las dimensiones de $\mathbf{M_3}$ estarán en $d_2 \times d_3$.

…

- Las dimensiones de $\mathbf{M_{n-1}}$ estarán en $d_{n-2} \times d_{n-1}$.
- Las dimensiones de $\mathbf{M_1}$ estarán en $d_{n-1} \times d_n$.

Para resolver el problema se van almacenando en una matriz **D** en la posición **[i][j]** las multiplicaciones escalares óptimas de la multiplicación encadena de matrices desde la matriz **i** hasta **j**, por lo que se trabajará con problemas de matrices continuas, esto es, si se está trabajando con problemas de 3 matrices, estas tendrán que estar continuas: $\mathbf{M_1xM_2xM_3}$, o $\mathbf{M_3xM_4xM_5}$ o $\mathbf{M_7xM_8xM_9}$.

Se comienza con problemas de 1 matriz, así que se sabe que serán 0 multiplicaciones escalares, por lo tanto:

$$D[i][j] = 0 \text{ cuando } i = j$$

Continuando con problemas de 2 matrices sería directamente la multiplicación de sus dimensiones, ya que no hay nada que minimizar, quedando:

$$D[i][j] = di\text{-}1 \times d_i \times d_{i+1} \text{ cuando } i+1 = j$$

Para problemas de 3 hasta n matrices hay que buscar el mínimo, suponiendo que k es última matriz de la parte izquierda de la última multiplicación de la asociación, esto es:

$$(M_i \times M_{i+1} \times \ldots \times M_{k-1} \times M_k) \times (M_{k+1} \times M_{k+2} \times \ldots \times M_{j-1} \times M_j)$$

En la parte izquierda se tuvo que calcular previamente y almacenar en **D[i][k]** y la parte derecha se tuvo que calcular y almacenar en **D[k+1][j]**, solo faltaría la cantidad de multiplicaciones escalares que se calculan con sus dimensiones que son: $d_{i-1} \times d_k \times d_j$ y sabiendo que la **k** puede ser un valor desde **i** hasta **j-1**, entonces el cálculo de D[i][j] para problemas de 3 o más matrices continuas será:

$$D[i][j] = \min_{i \le k < j} \left(D[i][k] + D[k+1][j] + d_{i-1} * d_k * d_j \right)$$

Se va resolviendo desde el caso base, problemas de 1 matriz, luego problemas de 2 matrices y así sucesivamente hasta llegar a la solución de n matrices. La Figura 5.14 muestra este paso a paso para un problema de **n** matrices.

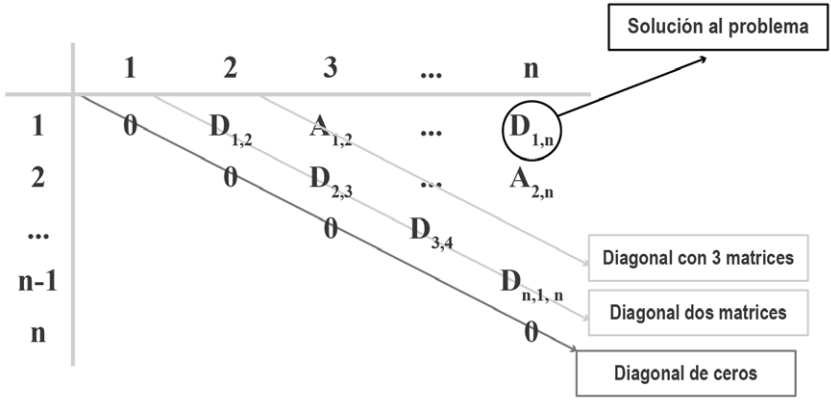


Figura 5.14 Llenado de la matriz para la solución de la multiplicación encadenada de matrices

El algoritmo de multiplicación encadenada de matrices es:

Algoritmo 5.12 Algoritmo de Godbole.

Entrada: la cantidad de matrices y sus dimensiones.

Salida: la cantidad de multiplicaciones escalares para la asociatividad óptima.

1. Generar una matriz llamada D cuadrada de n x n.
2. Para i desde 1 hasta n.
   2.1 Asignar 0 a la matriz D en la posición[i][i].
3. Para diag desde 1 hasta n-1.
   3.1 Para i desde 1 hasta n-diag.
      3.1.1 Asignar i+diag a j.
      3.1.2 Asignar infinito a la variable mínimo
      3.1.3 Para k desde i hasta j-1

3.1.3.1 Es D[i][k]+D[k+1][j]+d[i-1]*d[k]*d[j] menor que mínimo.

3.1.3.1.1 Sí,

3.1.3.1.1.1 Asignar D[i][k]+D[k+1][j]+d[i-1]*d[k]*d[j] a mínimo.

3.1.4 Asignar mínimo a la matriz D en la posición [i][j].

4. Regresar el contenido de D[1][n].

La implementación del algoritmo del algoritmo **multiplicación encadenada de matrices** quedaría de la siguiente forma:

```
1 // d = vector de las dimensiones de las matrices
2 // n = Cantidad de matrices.
3 void calcula(vector<int> d, int n){
4     int D[i+1][i+1];
5     for (int i=1; i<=n; i++){
6         D[i][i] = P[i][i] = 0;
7     }
8     int j, aux, men, menk;
9     for (int diag=1; diag<=n-1; diag++){
10        for (int i=1; i<=n-diag; i++){
11            j = i+diag;
12            men = INT_MAX;
13            for (int k=i; k<j; k++){
14                aux = D[i][k] + D[k+1][j] + d[i-1]*d[k]*d[j];
15                if (aux < men){
16                    men = aux;
17                }
18            }
19            D[i][j] = men;
20        }
21    }
22 }
```

Complejidad del Algoritmo de Kruskal.	
Clasificación del problema	P
Tiempo de ejecución (peor caso)	$O(n^3)$
Memoria adicional	$O(n^2)$

Si se desea reconstruir la asociatividad se requiere una matriz adicional del mismo tamaño que D para ir almacenando la k que minimiza cada uno de los problemas y posteriormente con un recorrido de *inorden* se puede reconstruir la asociatividad para conseguir el óptimo de multiplicaciones escalares.

5.8 Problema del BST óptimo

Un árbol binario de búsqueda (BST por sus siglas en inglés, *Binary Search Tree*), es una estructura de datos de organización jerárquica donde cada nodo puede tener un máximo de dos hijos, no permite datos repetidos, para cada nodo todos los nodos de su subárbol izquierdo son menores a este y todos los nodos de su subárbol derecho son mayores a él. Este tipo de estructuras son muy útiles para realización de búsquedas. La altura del árbol es la cantidad máxima de comparaciones que se tienen que hacer para buscar un dato.

Para unos mismos datos se pueden tener diferentes formas de árbol, esto va a depender de la forma como se insertan o borran los datos, por ejemplo: si se tuvieran los datos 1, 2 y 3 se pueden tener 5 formas diferentes de BST, la Figura 5.15 muestra estas 5 diferentes formas.

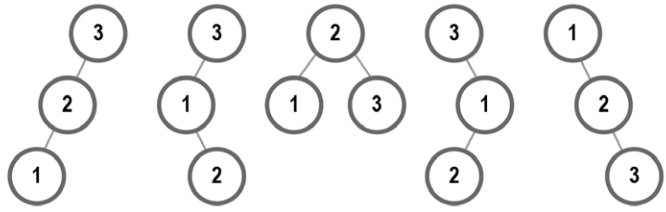


Figura 5.15 Formas que puede tener el BST con los datos 1, 2 y 3

El problema del BST óptimo es: dadas n llaves, cada llave con su probabilidad de ser buscada, encontrar la forma del árbol que minimice el tiempo promedio de búsqueda. El tiempo promedio de búsqueda de un BST se calcula sumado la cantidad de comparaciones que se requieren para llegar a cada nodo multiplicado por su probabilidad. Por ejemplo, si se tuvieran 3 datos con las siguientes probabilidades: el 1 con probabilidad de 0.6, el 2 con probabilidad de 0.1 y el 3 con probabilidad de 0.3 (la suma de las probabilidades debe dar 1.0 o 100%). El cálculo del promedio de búsqueda para cada forma de árbol se puede ver en la Figura 5.16.

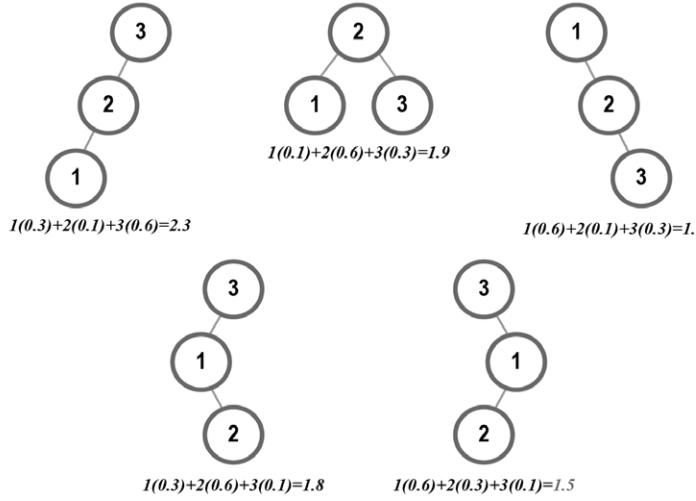


Figura 5.16 Ejemplo del cálculo del promedio de búsqueda de un BST

Como se puede visualizar en la Figura 5.16, el mínimo promedio de búsqueda de los datos 1, 2 y 3 con esas probabilidades es de 1.5 comparaciones con árbol donde 1 es la raíz, el 3 a la derecha del 1 y el 2 a la izquierda de 3.

Realizar todas las posibles formas de un BST para poder obtener el mínimo promedio de búsqueda tendría un comportamiento exponencial.

5.8.1 Algoritmo de Gilbert and Moore

El algoritmo de Gilbert and Moore (1959), utiliza la técnica de programación dinámica y almacena el resultado en una matriz **A** de dimisiones $(n+1)$x$(n+1)$ donde **n** es la cantidad de datos a almacenar, teniendo los reglones **1** hasta **n+1** y las columnas **0** hasta **n**. En donde **A[i][j]** almacena el mínimo promedio de búsqueda de los nodos **i** hasta **j**. Lo que se desea encontrar el tiempo mínimo promedio de búsquedas de todo el árbol, el cual estará almacenado en la celda **A[i][n]**.

El punto de arranque es tener un árbol de 1 solo nodo, por lo que esto estará en la celda **A[i][i]**, la cual será igual a $\mathbf{p_i}$ (la probabilidad del nodo **i**), ya que sería una comparación por su probabilidad. Partiendo de esto hay que encontrar la solución general. Asumiendo que **k** es el nodo raíz del árbol óptimo desde **i** hasta **j**, esto es, k tiene que ser un valor entre **i** y **j** inclusive. Los subárboles del nodo raíz k serán BST óptimos que tienen un tiempo mínimo promedio previamente calculado y almacenados en:

- El subárbol izquierdo que tiene las llaves desde i hasta k-1 y su valor del tiempo mínimo promedio estará almacenado en A[i][k-1].

- El subárbol derecho que tiene las llaves desde k+1 hasta j y su valor del tiempo mínimo promedios estará almacenado en **A[k+1][j]**.

Ahora bien, el cálculo del subárbol desde i hasta j, cuando la k es la raíz, será:

- Una comparación por la probabilidad de p_k (la raíz).
- A[i][k-1] más una comparación por la probabilidad de los nodos desde i hasta k-1, para bajarlos de 1 nivel, ya que ahora serán el subárbol izquierdo de k.
- A[k+1][j] más una comparación por la probabilidad de los nodos desde k+1 hasta j, para bajarlos 1 nivel, ya que ahora serán el subárbol derecho de **k**.

Se tendrá que calcular para la k desde i hasta j y obtener el mínimo, la fórmula general queda de la siguiente forma:

$$A[i][j] = \min_{i \le k \le j} (A[i][k-1] + A[k+1][j]) + \sum_{a=i}^{j} p_a$$

Problemas de un nodo se van resolviendo desde el caso base, luego problemas de dos nodos continuos, luego problemas de 3 nodos continuos hasta llegar a el problema final de n nodos. El autor utiliza una diagonal (A[0][1], A[1][2], … , A[n+1][n]) para poner ceros, apoyándose cuando no haya nodos en el subárbol izquierdo o derecho. La Figura 5.17 muestra este paso a paso para un BST de n nodos.

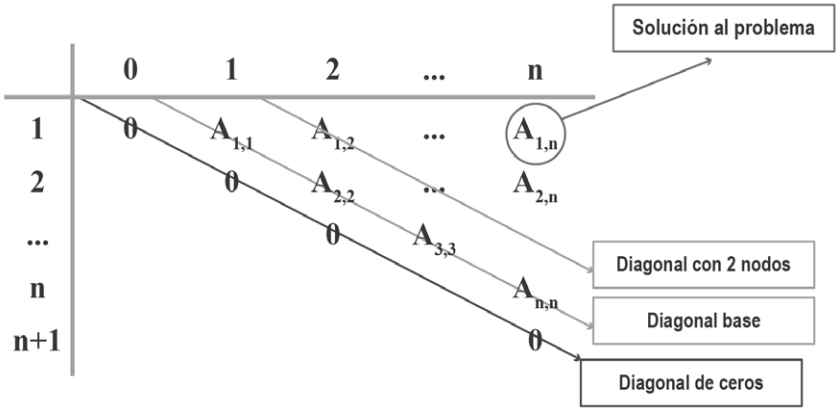


Figura 5.17 Llenado de la matriz para la solución del BST óptimo

El algoritmo del **BST óptimo** es:

Algoritmo 5.13 Algoritmo del Gilbert and Moore.

Entrada: la cantidad de llaves y sus probabilidades.

Salida: el tiempo mínimo promedio de búsqueda.

1. Generar una matriz llamada A cuadrada de n+1 x n+1.
2. Para i desde 1 hasta n.
   2.1 Asignar 0 a la matriz A en la posición[i][i-1].
   2.2 Asignar pi a la matriz A en la posición [i][i].
3. Para diag desde 1 hasta n-1.
   3.1 Para i desde 1 hasta n-diag.
      3.1.1 Asignar i+diag a j.
      3.1.2 Asignar infinito a la variable mínimo
      3.1.3 Para k desde i hasta j

3.1.3.1 Es A[i][k-1]+A[k+1][j] menor que mínimo.

3.1.3.1.1 Sí,

3.1.3.1.1.1 Asignar A[i][k-1]+A[k+1][j] a mínimo.

3.1.4 Asignar mínimo + la suma de las probabilidades desde i hasta j a la matriz A en la posición [i][j].

4. Regresar el contenido de A[1][n].

La implementación del algoritmo del algoritmo **BST óptimo** quedaría de la siguiente forma:

```
1 // p = vector de probabilidades indexado desde 0..(n-1)
2 // pacum = vector de probabilidades acumuladas
3 //         indexado desde 0..n.
4 float BSTOp(vector<float> p, vector<float> pacum){
5         int n = p.size();
6         float A[n+1][n+1];
7         for (int i=1; i<=n; i++){
8                 A[i][i-1] = 0;
9                 A[i][i] = p[i-1];
10        }
11        for (int diag=1; diag<n; diag++){
12                for (int i=1; i<=n-diag; i++){
13                        int j=i+diag;
14                        float min = A[i][i-1]+A[i+1][j];
15                        for (int k=i+1; k<=j; k++){
16                                if (A[i][k-1]+A[k+1][j] < min){
17                                        min = A[i][k-1]+A[k+1][j];
18                                }
19                        }
20                        A[i][j] = min + pacum[j]-pacum[i-1];
21                }
22        }
23        return A[1][n];
24 }
```

Complejidad del algoritmo de Gilbert and Moore.	
Clasificación del problema	P
Tiempo de ejecución (peor caso)	$O(n^3)$
Memoria adicional	$O(n^2)$

Si se desea reconstruir el BST óptimo se requiere una matriz adicional del mismo tamaño que A para ir almacenando la k que minimiza ese subárbol y posteriormente, con un recorrido de *inorden* se puede reconstruir el árbol para conseguir esos costos mínimos.

5.9 Problema de coloreo de grafos

El coloreo de grafos, es un problema en donde se va coloreando (etiquetando) a cada vértice de un grafo no dirigido con la restricción de los vértices adyacentes no tengan el mismo color o etiqueta, esto es que cada vértice tenga un color diferente a los de sus vecinos. Esta idea puede servir para solucionar problemas de selección para grupos, equipos, etc., donde los vértices son las personas u objetos y las restricciones se vean como arcos que los une. La Figura 5.18 muestra un ejemplo de un posible coloreo de un grafo con 7 vértices.

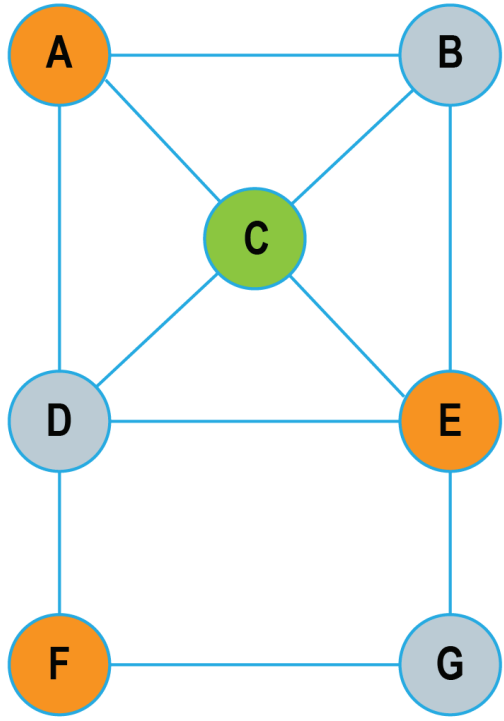


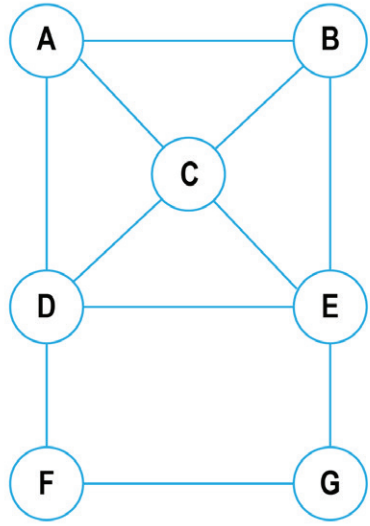
Figura 5.18 Ejemplo de coloreado de grafos

En este ejemplo, los vértices A, E y F pueden tener el mismo color o etiqueta dado que no son adyacentes, los vértices B, D y G tienen el mismo color o etiqueta al no ser vecinos y tienen que ser un color diferente al de los vértices A, B, D y E por ser sus vecinos. Otra posible solución puede ser que el vértice F o G tengan el mismo color del vértice C, pero solo uno de los dos, ya que entre ellos son vecinos.

5.9.1 Algoritmo de Welsh Powell

El algoritmo de coloreo de grafos de Welsh y Powell de 1967, consiste en obtener el grado de cada vértice (cantidad de arcos que llegan a él) y ordenarlos en forma descendente. Se empieza colorear el de mayor grado y se va checando en forma ordenada si se puede colorear con el mismo color a algún otro vértice. Una vez terminando esta pasada se continua con el siguiente vértice de mayor grado siguiendo la misma idea, hasta llegar al último. La Figura 5.19 muestra un ejemplo de este algoritmo paso a paso.

En primer lugar al tener el grafo, se calcula el grado de cada vértice y se ordena en forma descendente.

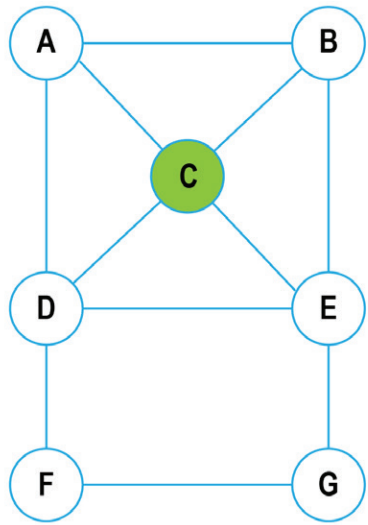


El ordenamiento queda de la siguiente forma:

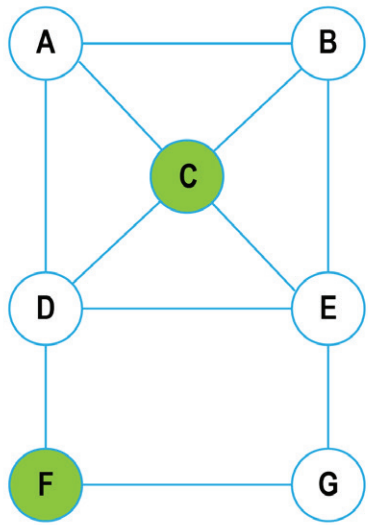
- C con grado 4
- A con grado 3
- B con grado 3

- D con grado 3
- E con grado 3
- F con grado 2
- G con grado 2

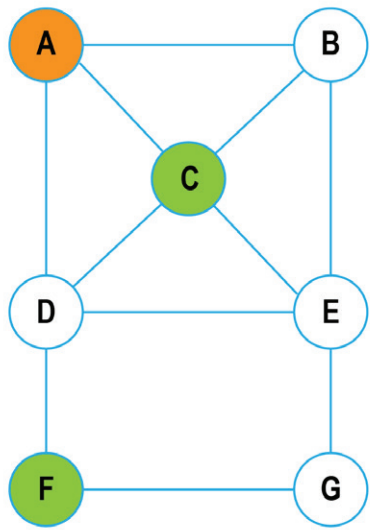
Se selecciona el vértice C y se le asigna el primer color.



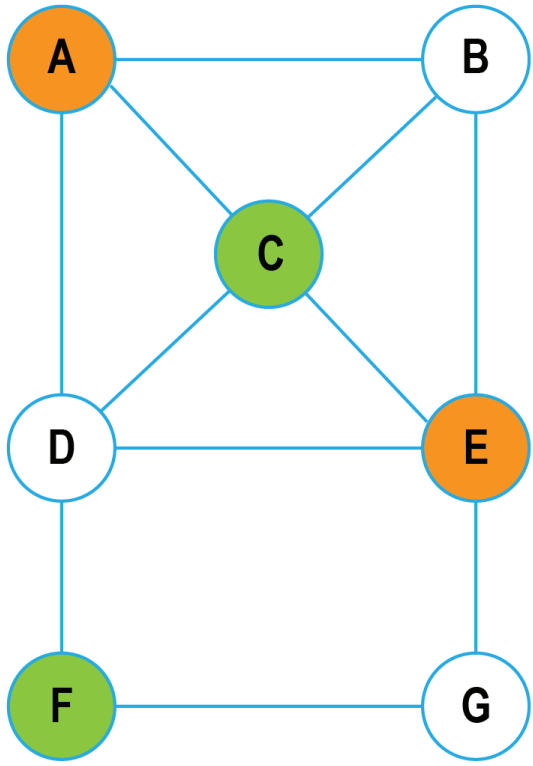
Posteriormente, se sigue analizando la lista para ver si a algún otro vértice se le puede asignar el mismo color. No pueden ser los vértices A, B, D ni E por ser vecinos de C, la F sí puede ser, la G no puede ser por ser vecino de F que previamente se le asignó ese color, con esto termina la primera pasada con el primer color.



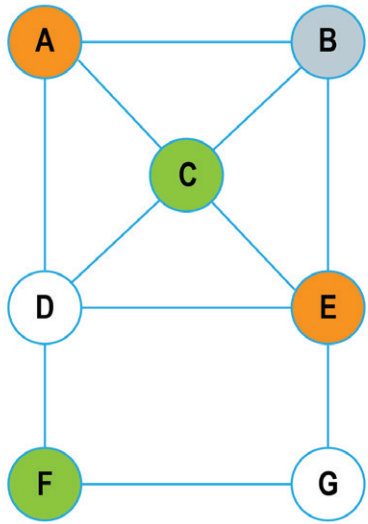
Luego se selecciona el siguiente vértice con mayor grado, al cual no se le asignado color y le asigna el color 2, en este caso es el vértice A.



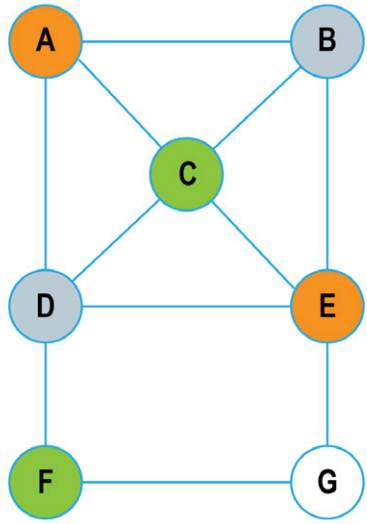
Se sigue con la lista analizando si a los siguientes vértices se les puede asignar el color 2, a B y D no se puede por ser adyacentes del vértice A, pero al vértice E sí puede y se le asigna el color 2.



El siguiente vértice sin colorear se analiza, es el G y no puede asignarle ese color porque es vecino de E. Vuelve a empezar la lista con el color 3 verificando el vértice con mayor grado que no ha sido coloreado, en ese caso es el vértice B y le asigna el color 3.



Sigue analizándose la lista para verificar los vértices no asignados de color y le asigna el color 3 al vértice D.



Siguiendo la lista, se topa con el último vértice que no ha sido coloreado y al ver que no tiene adyacencia con B ni con D, lo colorea del color 3.

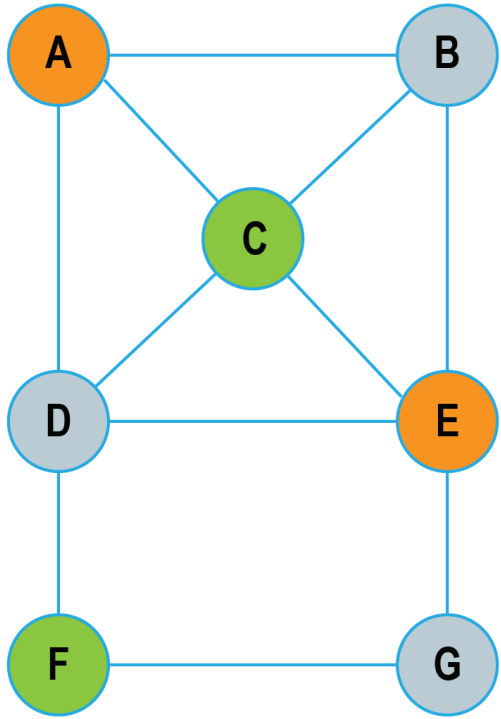


Figura 5.19 Ejemplo paso a paso de coloreado de grafos

El algoritmo de **Welsh Powel de coloreado de grafos** es:

Algoritmo 5.14 Algoritmo de Welsh Powel de coloreado de grafos.

Entrada: matriz de adyacencias.

Salida: vector con vértices y colores asignados.

1. Generar un vector con los grados de cada vértice

2. Ordenar descendentemente por grado de vértice.

3. Inicializar la variable colNumber con 0

4. Para cada vértice dentro del vector

    4.1 No ha sido coloreado ese vértice.

        4.1.1 Sí,

            4.1.1.1 Sumarle 1 a la variable colNumber

            4.1.1.2 Asignarle colNumber al vértice.

            4.1.1.3 Para vértice en la lista a partir del que se acaba de colorear.

                4.1.1.3.1 No ha sido coloreado ni es adyacente a algún vértice con este color

                    4.1.1.3.1.1 Sí,

                        4.1.1.3.1.1.1 Asignarle el colNumber a ese vértice

5. Regresar el vector de colores.

La implementación del algoritmo del **algoritmo de Welsh Powel de coloreado de grafos** quedaría de la siguiente forma:

```
// degree = grado del vértice
// vtx = vértice
struct color{
        int conex;
        int vtx;
};
// matAdj = Matriz de Adyacencias del Grafo
// vtxColor = vector de vértices ordenado descendentemente por grado
 vector<int> colorGraph(bool matAdj[MAX][MAX], vector<color>
&vtxColor){
        int colNum = 0;
        int n = vtxColor.size();
        vector<int> colored(n, 0);
        for (auto it=vtxColor.begin(); it!=vtxColor.end(); ++it){
                int i = (*it).vtx;
                if (!colored[i]){
                        colored[i] = ++colNum;
                        unordered_set<int> conj;
                        conj.insert(i);
                        for (auto it2=it+1; it!=vtxColor.end();
++it2){
                                int j= (*it2).vtx;
                                if (!colored[j] && !matAdj[i][j] &&
 conj.find(j)==conj.end()){
                                        colored[j] = colNum;
                                        conj.insert(j);
                                }
                        }
                }
        }
        return colored;
}
```

Complejidad del algoritmo de Gilbert and Moore.	
Clasificación del problema	P
Tiempo de ejecución (peor caso)	$O(n^2)$
Memoria adicional	$O(n)$

Existen otras variantes del algoritmo de Welsh Powell para el coloreo de grafos, como el de ordenar los vértices en forma ascendente en lugar de descendentemente por su grado.

5.10 Problema de flujo máximo

El problema del flujo máximo, es que dado un grado dirigido, quieres ver el máximo flujo que puede ser movido de un punto de partida a un punto de llegada. Este problema puede ser que lo que quieras mover, es flujo de agua, transmisión de datos, vehículos. Por ejemplo: para el grafo de la Figura 5.20 el flujo máximo de A a G es 8.

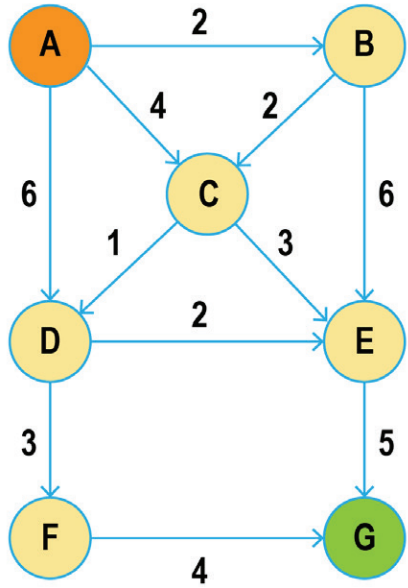


Figura 5.20 Ejemplo de grafo dirigido para calcular el flujo máximo de A a G

La solución de este ejemplo se muestra en la Figura 5.21, en donde se puede apreciar en rojo el flujo utilizado en cada uno de los arcos.

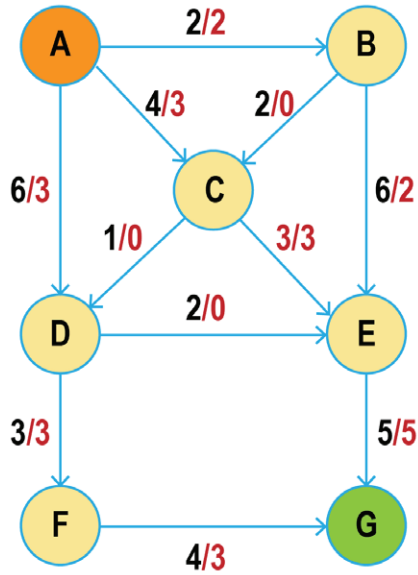


Figura 5.21 Solución de flujo máximo de A a G

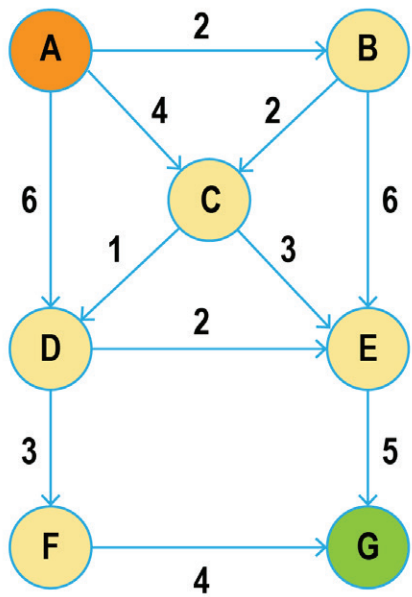
La solución nos indica que:

- Del arco A-B se utilizan las 2 unidades que tiene el arco.
- Del arco A-C solo se utilizan 3 unidades de las 4 que tiene el arco.
- Del arco A-D solo se utilizan 3 unidades de las 6 que tiene el arco.
- Del arco B-C no se utiliza nada de las unidades del arco.
- Del arco B-E solo se utilizan 2 unidades de las 6 que tiene el arco.
- Del arco C-D no se utiliza nada de las unidades del arco.
- Del arco C-E se utilizan las 3 unidades que tiene el arco.

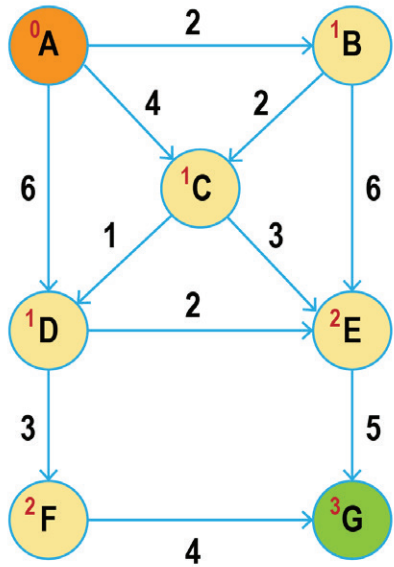
- Del arco D-E no se utiliza nada de las unidades del arco.
- Del arco D-F se utilizan las 3 unidades que tiene el arco.
- Del arco E-G se utilizan las 5 unidades que tiene el arco.
- Del arco F-G solo se utilizan 3 de las 4 unidades del arco.

5.10.1 Algoritmo de Dinic

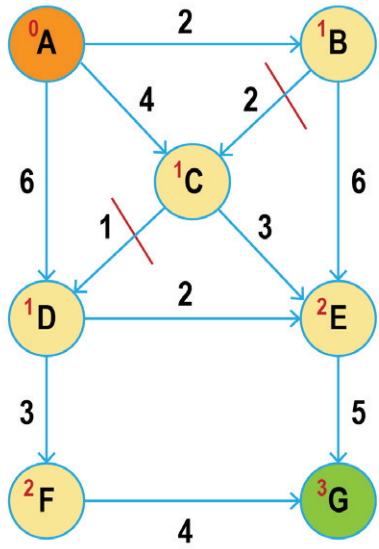
El algoritmo de Dinic, nombrado así por su autor Yefim A. Dinitz (1970) va iterando en posibles trayectorias del punto de origen al punto destino, etiquetando a los nodos intermedios por su nivel dentro de la trayectoria y obteniendo un costo residual en cada iteración. La Figura 5.22 muestra paso a paso cómo se va realizando este algoritmo.



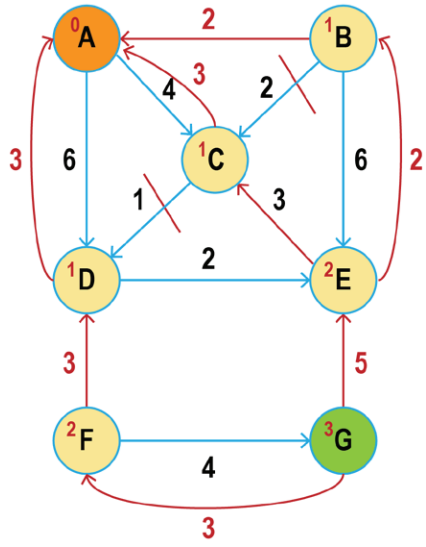
En este ejemplo se desea obtener el flujo máximo para ir del vértice A al vértice G, en el primer paso desarrolla una búsqueda en anchura (BFS) y va etiquetando a cada vértice por su nivel en la trayectoria.



Como se puede apreciar, el vértice de arranque A se etiqueta con 0, B, C y D, que son vecinos directos de A, se etiquetan con nivel 1, con nivel 2 se etiqueta a E y F; y, por último, se etiqueta a la G con nivel 3. Posterior a esto se bloquean los arcos que van lleva a dos vértices del mismo nivel.



En este caso se cancelaron los arcos B-C y C-D. Como punto siguiente se van sacando flujos desde el vértice de llegada poniendo los costos residuales, esto es, se va poniendo en lado contrario el costo utilizado.



En este caso se utilizaron:

- Las 5 unidades del arco E-G.
- 3 de las 4 unidades del arco F-G.
- Las 3 unidades del arco D-F.
- Las 3 unidades del arco C-E.
- 3 de las 6 unidades del arco A-D.
- 3 de las 4 unidades del arco A-C.
- 2 de las 6 unidades del arco B-E.
- Las 2 unidades del arco A-B.

Dando las siguientes trayectorias de A a G.

- A -> B -> E -> G con 2 unidades.
- A -> C -> E -> G con 3 unidades.
- A -> D -> F -> G con 3 unidades.

Con un flujo máximo hasta este momento de 8 unidades.

En la siguiente iteración vuelve a etiquetar los niveles de los vértices en la trayectoria de A a G contando ahora con los costos residuales hasta el momento, quedando:

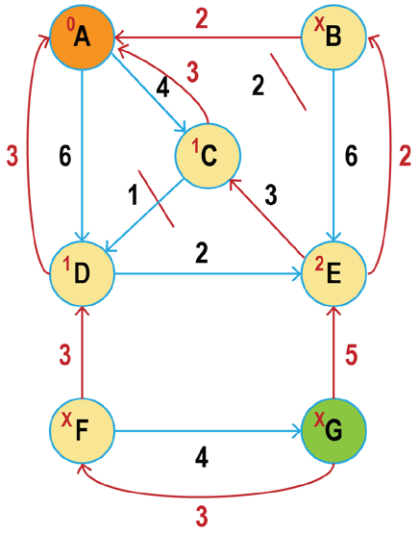


Figura 5.22 Paso a paso el ejemplo con el algoritmo de Dinic

Como ya no hay forma de llegar al vértice final, en este momento el algoritmo para y obtiene un flujo máximo de 8.

El algoritmo de **Dinic de flujo máximo** es:

Algoritmo 5.15 Algoritmo de Dinic de flujo máximo.

Entrada: lista de adyacencias, vértice origen y vértice de llegada.

Salida: flujo máximo.

1. Inicializa la variable maxflow con 0.

2. Mientras se pueda alcanzar desde el vértice inicial al vértice final mediante la búsqueda en anchura (BFS). El BFS etiqueta el nivel de cada vértice.

    2.1 Actualiza la variable maxflow con la suma de flujo

    2.2 Actualizando además el residual de cada arco utilizado en cada trayectoria.

3. Regresar la variable maxflow.

La implementación del **algoritmo de Dinic de flujo máximo** quedaría de la siguiente forma:

```
// n = cantidad de vértices

struct Edge{
    int v ;
    int flow;
    int C;
    int rev ;
};

bool BFS(vector<Edge> &adj, int s, int t)
{
    for (int I = 0 ; i < n ; i++){
        level[i] = -1;
    }
    level[s] = 0;
    queue< int > q;
    q.push(s);
    vector<Edge>::iterator i ;
    while (!q.empty()){
        int u = q.front();
        q.pop();
        for (i = adj[u].begin(); i != adj[u].end(); i++){
            Edge &e = *i;
            if (level[e.v] < 0  && e.flow < e.C){

                level[e.v] = level[u] + 1;

                q.push(e.v);
            }
        }
    }
    return level[t] < 0 ? false : true ;
}

int sendFlow(vector<Edge> &adj, int u, int flow, int t, int start[])
{
    if (u == t)
        return flow;
    while (start[u] < adj[u].size()){
        Edge &e = adj[u][start[u]];
        if (level[e.v] == level[u]+1 && e.flow < e.C){
            int curr_flow = min(flow, e.C-- e.flow);
            int temp_flow = sendFlow(e.v, curr_flow, t, start);
            if (temp_flow > 0){
                e.flow += temp_flow;
                adj[e.v][e.rev].flow -= temp_flow;
                return temp_flow;
            }
        }
        start[u]++;
    }
```

```
52      return 0;
53  }
54
55  int maxflowD(vector<Edge> &adj, int s, int t){
56      if (s == t)
57          return -1;
58      int maxflow = 0;
59      while (BFS(adj, s, t) == true){
60          vector<int> start(n+1,0);
61          while (int flow = sendFlow(adj, s, INT_MAX, t, start))
62              maxflow +=218loww;
63      }
64      return total;
65  }
```

Complejidad del Algoritmo de Dinic de flujo máximo.	
Clasificación del problema	P
Tiempo de ejecución (peor caso)	$O(\mid V \mid^2 \mid E \mid)$
Memoria adicional	O(n)

La memoria adicional es para ir almacenando el nivel de cada vértice. Existen otros algoritmos para el cálculo del flujo máximo como el Edmonds-Karps, que corre con una complejidad de $O(|E|^2|V|)$, o el de Ford–Fulkerson, que corre con una complejidad de O(Ef) donde f es el flujo máximo.

5.11 Algoritmos aleatorios

En un algoritmo aleatorio su comportamiento depende, en gran medida, del resultado de elecciones hechas al azar. Existen dos ventajas principales que suelen tener los algoritmos aleatorios. Primero, a menudo el tiempo de ejecución o el espacio

utilizado en memoria es menor que el del mejor algoritmo determinista que se conozca para el mismo problema. En segundo lugar, si observamos los diversos algoritmos aleatorios que se han inventado hasta ahora, descubrimos que, invariablemente, son extremadamente sencillos de comprender e interpretar.

Además, existen algunos algoritmos deterministas, especialmente aquellos que exhiben un buen tiempo promedio de ejecución, que con la simple introducción de elecciones al azar es suficiente para convertir un algoritmo simple e ingenuo, con un mal comportamiento en el peor de los casos, en un algoritmo aleatorizado que funciona bien con alta probabilidad para todas las posibles entradas. Esto será evidente cuando estudiemos el algoritmo de ordenamiento conocido como *Quicksort*.

5.11.1 Algoritmos aleatorizados y "Divide y vencerás"

Anteriormente ya revisamos la técnica **divide y vencerás** para diseñar algunos algoritmos. Esta técnica a menudo funciona bien junto con la aleatorización. Para ilustrarlo revisaremos dos algoritmos básicos que emplean la técnica **divide y vencerás:** encontrar la mediana de n números y el algoritmo *Quicksort*. En cada caso, el paso de "dividir" se realiza mediante aleatorización.

5.11.1.1 Encontrar la mediana

Supongamos que nos dan un conjunto S de n números. La mediana es el número que estaría en la posición media si tuviéramos que ordenarlos. Aunque, existe una "dificultad técnica" si n es par, ya que no existe una posición media; así que tendremos que ser más específicos: la mediana de S es igual al k-ésimo elemento más grande S, en donde $k = (n + 1) / 2$ si n es impar y

$k = n / 2$ si n es par. Por simplicidad de la explicación consideraremos que todos los números son distintos. Resulta evidente que si ordenamos los elementos es muy fácil encontrar la mediana en $O(n \log n)$. Pero, ¿sería posible hacerlo sin ordenar y logrando el mismo tiempo o incluso mejor? Bueno, pues lo intentaremos usando aleatorización con la técnica de **divide y vencerás**.

El primer paso clave es pasar del problema de la búsqueda de la mediana al problema más general de selección. Da un conjunto S de n números y un número k entre 1 y n, considera la función SELECT (S, k) que devuelve el k-ésimo elemento más grande en S. Como casos especiales, SELECT resuelve el problema de encontrar la mediana de S mediante SELECT (S, $n / 2$) o SELECT (S, $(n + 1) / 2$). Además, también permite resolver los problemas de encontrar el mínimo, SELECT (S, 1), y el máximo, SELECT (S, n).

La estructura básica de este algoritmo es la siguiente: Vamos a elegir un elemento a_i del conjunto S. A partir de este "pivote" formaremos dos conjuntos $\mathbf{S_{lower}}$ con los elementos menores al pivote, y $\mathbf{S_{greater}}$ con los elementos mayores al pivote. Con esto, ya podemos determinar cuál de los dos conjuntos: S_{lower} y $S_{greater}$ contiene el k-ésimo elemento más grande y recorrer solo ese. Sin especificar todavía cómo planeamos elegir al pivote, enseguida encontrarás la descripción del algoritmo.

Algoritmo 5.16. Encontrar la mediana (SELECT)

Entrada: un conjunto S de n números, k-ésimo elemento a buscar.

Salida: a_i, el k-ésimo elemento más grande en S.

1. Elegir un pivote, a_i.
2. Para cada elemento a_j en S hacer
3. Colocar a_j in S_{lower} si $a_j < a_i$.
4. Colocar a_j in $S_{greater}$ si $a_j > a_i$.
5. i = | S_{lower} |
6. Si i = k entonces
7. Regresar a_i.
8. Si no
9. Si i > k entonces
10. El k-ésimo elemento está en S_{lower}, llamar SELECT(S_{lower}, k)
11. Si no
12. Si i > k entonces
13. El k-ésimo elemento está en $S_{greater}$, llamar SELECT($S_{greater}$, $k - i - 1$)

Ahora, consideremos cómo el tiempo de ejecución del algoritmo se verá afectado por la selección del pivote. Supongamos que podemos elegir un pivote en tiempo lineal, el resto del algoritmo requiere de tiempo lineal (líneas 2-4) más el tiempo de la llamada recursiva. Esencialmente es muy importante que el

pivote reduzca significativamente el tamaño del conjunto que se está considerando, de tal forma que no sigamos trabajando con grandes conjuntos de números. Se puede demostrar que un elemento "bien centrado" puede servir como un buen pivote. Si tuviéramos una forma de elegir un pivote ai tal que hubiera al menos εn elementos más grandes y pequeños que a_i para cualquier constante $\varepsilon > 0$, entonces el tamaño de los conjuntos en la llamada recursiva se reduciría en un factor $(1 - \varepsilon)$ cada vez. Por lo tanto, el tiempo de ejecución T(n) estaría limitado por la recurrencia $T(n) \leq T((1 - \varepsilon)n + cn)$. Si desarrollamos esta recurrencia para cualquier $\varepsilon > 0$, obtenemos:

$$T(n) \leq cn + (1 - \varepsilon)cn + (1 - \varepsilon)^2 cn + \cdots = [1 + (1 - \varepsilon) + (1 - \varepsilon)^2 + \cdots$$

$$cn < \frac{1}{\varepsilon} * cn$$

Del desarrollo anterior, podemos obtener que $T(n) = O(n)$.

La elección de un pivote "bien centrado", en el sentido que acabamos de definir, se definie en encontrar un pivote que cumpla con la siguiente regla: elegir un pivote, a_i<, uniformemente al azar.

5.11.1.2 Quicksort

La técnica aleatoria de **divide y vencerás** que usamos para encontrar la mediana será nuestra base del algoritmo *Quicksort*. Como antes, vamos a elegir un pivote para el conjunto de entrada S y separaremos S en los elementos menores y mayores al pivote. La diferencia es que, en lugar de buscar la mediana en un solo lado, tendremos que hacerlo para ambos lados de la recursión. Además, necesitamos incluir un caso base cuando el tamaño del conjunto es menor a 4. He aquí el algoritmo.

Algoritmo 5.17. *Quicksort* aleatorizado

Entrada: un conjunto S de n números.

Salida: el conjunto S ordenado.

1. Si $|S| < 4$ entonces
2. Ordena y regresa S.
3. Si no
4. Elegir un pivote, a_i, uniformemente al azar.
5. Para cada elemento a_j en S hacer
6. Colocar a_j in S_{lower} si $a_j < a_i$.
7. Colocar a_j in $S_{greater}$ si $a_j > a_i$.
8. QUICKSORT(S_{lower})
9. QUICKSORT ($S_{greater}$)

Pero, ¿qué sucede si siempre elegimos el elemento más pequeño como pivote? Entonces, tendríamos un tiempo de ejecución de $O(n^2)$. De hecho, este el peor tiempo de ejecución para un *Quicksort*. Aunque, en el lado positivo, si los pivotes seleccionados resultan ser las medianas de los conjuntos en cada iteración obtendríamos un tiempo de ejecución $O(n \log n)$.

Entonces, lo que debemos hacer es acotar nuestro tiempo a $O(n \log n)$, logrando que los pivotes estén perfectamente centrados. De manera similar a lo que hicimos con el pivote en SELECT, la definición crucial es la de un pivote central. En este caso, lo que queremos es que este pivote central divida el conjunto de tal forma que cada lado contenga al menos una cuarta parte de los elementos.

Ahora, lo que haremos será modificar ligeramente el algoritmo anterior para que haga las llamadas recursivas cuando encuentre un buen pivote central. Esencialmente, descartaremos aquellos pivotes que estén "descentrados" e intentaremos uno nuevo.

Algoritmo 5.17. *Quicksort* aleatorizado modificado

Entrada: un conjunto S de n números.

Salida: el conjunto S ordenado.

1. Si $|S| < 4$ entonces
2. Ordena y regresa S.
3. Si no
4. Mientras no se haya encontrado un pivote central hacer
5. Elegir un pivote, a_i, uniformemente al azar.
6. Para cada elemento a_j en S hacer
7. Colocar a_j in S_{lower} si $a_j < a_i$.
8. Colocar a_j in $S_{greater}$ si $a_j > a_i$.
9. Si $| S_{lower} | \geq |S|/4$ y $| S_{greater} | \geq |S|/4$ entonces
10. a_i es el pivote central, terminamos mientras
11. QUICKSORT(S_{lower})
12. QUICKSORT ($S_{greater}$)

El tiempo de ejecución esperado de este algoritmo modificado se puede analizar de manera muy simple por analogía directa con el análisis realizado para el algoritmo de la mediana. Sin embargo, no describiremos este análisis aquí. Solo diremos que el tiempo esperado de ejecución para este algoritmo es *O (n log n)*.

5.12 Ejercicios del capítulo 5

1. Modifica el algoritmo de ***Longest Common Substring*** de forma tal que se obtenga el *substring* más largo dados dos *string* y no solo el tamaño.

2. Modifica el algoritmo de ***Longest Common Substring*** para que lo obtenga dado 3 *strings* y no solo 2.

3. Modifica el algoritmo de ***Longest Common Subsequence*** de forma tal que se obtenga la subsecuencia más larga dados dos *string* y no solo el tamaño.

4. Modifica el algoritmo de ***Longest Common Subsequence*** para que lo obtengas dado 3 *strings* y no solo 2.

5. Modifica el algoritmo de Dijkstra para que se pueda consultar la trayectoria y su costo del punto de partida a todos los demás puntos.

6. Modifica el algoritmo de Dijkstra para que pare cuando se desee consultar solo la trayectoria y costo de un punto de salida a un punto de llegada.

7. Modifica el algoritmo de Floyd para que se pueda consultar la trayectoria y su costo de cualquier punto a otro punto.

8. Genera una aplicación para que ejecute con el algorimto que sea más eficiente entre el problema de la mochila con programación dinámica y **divide y vencerás**.

9. Modifica el algoritmo de la mochila con *backtracking* para que genere los objetos que maximizan el valor de la mochila.

10. Modifica el algoritmo de la mochila con *Branch & Bound* para que genere los objetos que maximizan el valor de la mochila.

11. Genera una aplicación que realice la comparación entre los algoritmos de *backtracking* y *Branch & Bound* para un caso de entrada, diciendo cuál es mejor y por qué de los dos.

12. Modifica el algoritmo del problema del viajero con *Branch & Bound* para que genere el ciclo hamiltoniano de la solución.

13. Modifica el algoritmo de **Prim** para que genere el árbol de solución.

14. Modifica el algoritmo de **Kruskal** para que genere el árbol de solución.

15. Genera una aplicación que realice la comparación entre los algoritmos de **Prim** y **Kruskal** para la obtención del árbol de mínima expansión diciendo para un caso de entrada cuál fue mejor y por qué.

16. Modifica el algoritmo de Godbole para la multiplicación encadena de matrices para que genere la asociatividad de las matrices que minimiza con esa cantidad mínima de multiplicaciones escalares.

17. Modifica la versión del algoritmo de Welsh Powell para el problema de coloreo de grafos para que se ordene en forma ascendente por grado de los vértices.

18. Genera una aplicación para que realice la comparación de los algoritmos de coloreo de grafos cuando se ordenan en forma ascendente o descendente por grado de los vértices señalando cuál es mejor para un caso de entrada.

19. Modifica el algoritmo de Dinic para flujo máximo para que de los arcos y el fujo por ese arco para que se obtenga el flujo máximo.

20. Investiga el algoritmo de Edmonds-Karps y realiza una comparación con el algoritmo de Dinic para decidir cuál es mejor para un caso específico.

Capítulo 6. Geometría computacional.

La geometría computacional es el área de las ciencias computacionales que se ocupa de calcular las propiedades geométricas de conjuntos de objetos geométricos en el espacio, como la sencilla operación de determinar si un punto está encima o debajo de una línea determinada. En un sentido más general, la geometría computacional se ocupa del diseño y análisis de algoritmos para resolver problemas geométricos.

Si bien, la geometría computacional es un área de las ciencias computacionales que suena interesante, ¿en qué tipo de problemas podemos aplicarlas?

Imagina que estás caminando por el campus de tu universidad, de repente recuerdas que tienes que hacer una llamada telefónica urgente y, sucede algunas veces, tu celular se ha quedado sin batería. Lo bueno, es que todavía hay muchos teléfonos públicos en el campus , por supuesto, deseas ir al más cercano. ¿Pero cuál es el más cercano? Sería muy útil poder contar con un mapa que nos muestre, por regiones, el teléfono público más cercano, ¿no es cierto? Pero, ¿cómo definimos estas regiones? Quizás, esta situación no sea un tema importante. Sin embargo, describe una aplicación básica de un concepto geométrico fundamental que juega un papel muy importante en muchas aplicaciones. Podemos ver la subdivisión del campus en un diagrama de Voronoi (Figura 6.1), que estudiaremos en este capítulo más adelante. Este concepto se puede utilizar para modelar áreas comerciales de diferentes ciudades, para guiar a robots e incluso describir y simular el crecimiento de cristales.

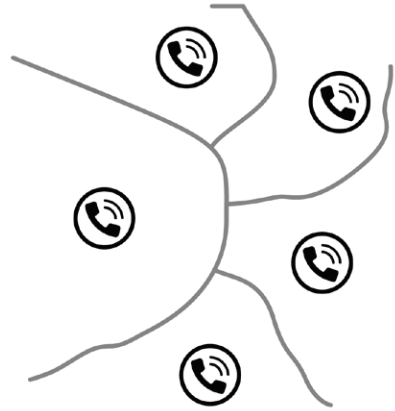


Figura 6.1 Diagrama de Voronoi

Bueno, supongamos que ya hemos encontrado el ansiado teléfono público más cercano. Ahora, con un mapa del campus en mano, probablemente tendremos pocos problemas para llegar a él por el camino más corto sin golpear paredes u otros obstáculos. Sin embargo, programar a un robot para que realice la misma tarea es mucho más difícil. Lo anterior implicaría que tendríamos que calcular el camino más corto entre dos puntos evitando colisionar con una colección de obstáculos geométricos.

Estos y otros tipos de problemas son resueltos usando la geometría computacional. A lo largo capítulo analizaremos algoritmos de esta área, pero antes revisaremos la implementación de primitivas geométricas básicas necesarias para estos algoritmos.

6.1 Proximidad e intersección

6.1.1 Punto

Empecemos con la primitiva geométrica más simple: el punto. El punto es el bloque de construcción básico de los objetos

de geometría de dimensiones superiores. En el espacio euclidiano 2D, un punto es, generalmente, representado como estructura o clase. Por ejemplo:

```
1   class Point {
2   private:
3     double x, y;
4
5   public:
6     Point();
7     Point(double, double);
8     Point(const Point&);
9
10    double getX() const;
11    double getY() const;
12  };
```

En algunos problemas necesitaremos determinar la distancia euclidiana entre dos puntos.

```
1   double dist(const Point &p1, const Point &p2) {
2     double diffx = p1.getX() - p2.getX();
3     double diffy = p1.getY() - p2.getY();
4     return sqrt((diffx * diffx) + (diffy * diffy));
5   }
```

Si necesitamos rotar un punto un ángulo θ en sentido antihorario alrededor del origen (0,0) usaremos una matriz de rotación como se muestra en la Figura 6.2.

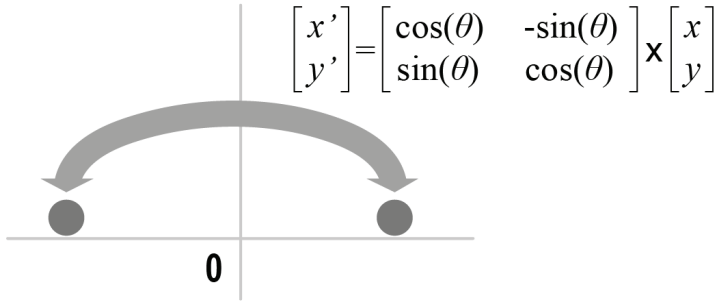


Figura 6.2 Rotación de un punto en 180 grados en sentido antihorario alrededor del origen (0,0)

```
1   Point rotate(const Point &p, double theta) {
2     double rad = degToRad(theta);
3     return Point(p.getX() * cos(rad) - p.getY() * sin(rad),
4                  p.getX() * sin(rad) + p.getY() * cos(rad));
5   }
```

6.1.2 Línea

Las líneas son de longitud infinita en ambas direcciones. Si son infinitas, ¿cómo representamos una línea en un plano? Las líneas pueden ser representadas de dos formas diferentes: ya sea usando puntos o la ecuación de la línea. En el primer caso, cada línea l está completamente representada por cualquier par de puntos (x_1, y_1) y (x_2, y_2) que se encuentren en la línea. Si usamos la ecuación de la línea, una línea se describe mediante la ecuación $y = mx + b$, donde m es la pendiente de la línea y b es la intersección con el eje, es decir, el punto único $(0, b)$ donde cruza el eje X. La línea l tiene una pendiente $m = \Delta y / \Delta x = (y_1 - y^2) / (x_1 - x_2)$ e intercepta en $b = y_1 - mx_1$.

Sin embargo, las líneas verticales no se pueden describir con estas ecuaciones, porque dividir por Δx significaría dividir por cero. La ecuación $x = c$ denota una línea vertical que cruza el

eje *X* en el punto *(c, 0)*. Este caso especial requiere una atención adicional al realizar la implementación. Por eso, y para tener una implementación más robusta, usaremos la fórmula más general $ax + by + c = 0$ como base, ya que cubre todas las líneas posibles en el plano.

```
1   class Line {
2   private:
3     double a, b, c;
4
5   public:
6     Line();
7     Line(double, double, double);
8     Line(const Line&);
9
10    double getA() const;
11    double getB() const;
12    double getC() const;
13  };
```

Ahora, ¿cómo podemos definir una línea que pasa por dos puntos dados? Fácil, con la función **`pointsToLine`** que recibe dos puntos y nos regresa la línea que pasa por ambos puntos.

```
1   Line pointsToLine(const Point &p1, const Point &p2) {
2     double aux;
3
4     if (fabs(p1.getX() - p2.getX()) < EPSILON) {
5       return Line(1.0, 0.0, -p1.getX());
6     } else {
7       aux = (p1.getY() - p2.getY()) / (p1.getX() - p2.getX());
8       return Line (-aux,
9                    1.0,
10                   -(aux * p1.getX()) - p1.getY());
11    }
12  }
```

En la línea 4 del código verificamos si los puntos forman parte de una línea vertical. Dado que la resta se realiza sobre números flotantes, con el fin de evitar error de manejo de precisión, en vez

de verificar si el resultado es cero optamos por determinar si el valor resultado se encuentra dentro de un rango de error llamado **EPSILON**. Este valor es un número muy pequeño, comúnmente es 1e-9. Si resulta que ambos puntos están sobre una línea vertical **(fabs(p1.getX() - p2.getX()) < EPSILON)** definimos valores por omisión para representar esa línea recta. Si no es el caso realizamos los cálculos para obtener los valores de *a*, *b*, y *c*.

También podemos construir una línea a partir de un punto y la pendiente de la línea. La función **pointAndSlopeToLine** recibe ambos elementos y nos regresa la línea que cumple con los datos de entrada.

```
1   Line pointAndSlopeToLine(const Point &p, double m) {
2     return Line(-m, 1, -(-m * p.getX()) + p.getY());
3   }
```

Decimos que dos líneas siempre tienen un punto de intersección a menos que sean paralelas; en cuyo caso, no lo tienen. Para determinar si dos rectas son paralelas solo debemos verificar si sus coeficientes *a* y *b* son iguales.

```
1   bool areParallel(const Line &l1, const Line &l2) {
2     return ( (fabs(l1.getA() - l2.getA()) <= EPSILON) &&
3              (fabs(l1.getB() - l2.getB()) <= EPSILON) );
4   }
```

Basándonos en la función previa también podemos comprobar si esas dos líneas son la misma. Decimos que las líneas *a* y *b* son la misma si son paralelas y, además, sus coeficientes *c* son los mismos. En otras palabras, los tres coeficientes: *a*, *b* y *c* son los mismos.

```
1   bool areSameLine(const Line &l1, const Line &l2) {
2     return ( areParallel(l1, l2) &&
3              (fabs(l1.getC() - l2.getC()) <= EPSILON) );
4   }
```

Ahora, si dos líneas no son paralelas, y por lo tanto no pueden ser la misma, ambas líneas se intersecarán en un cierto punto. El punto de intersección (x, y) se puede encontrar resolviendo el sistema de ecuaciones que se genera de las fórmulas: $a_1x + b_1y + c_1 = 0$ y $a_2x + b_2y + c_2 = 0$.

```
1   Point* intersectsAt(const Line &l1, const Line &l2) {
2     double x, y;
3
4     if (areParallel(l1, l2)) {
5       return NULL;
6     }
7     x = ((l1.getB() * l1.getC()) - (l1.getB() * l2.getC())) /
8         ((l2.getA() * l1.getB()) - (l1.getA() * l2.getB()));
9     if (fabs(l1.getB()) < EPSILON) {
10      y = -((l1.getA() * x) + l1.getC());
11    } else {
12      y = -((l1.getA() * x) + l2.getC());
13    }
14    return new Point(x, y);
15  }
```

La función **`intersectAt`** regresa el punto de intersección de ambas líneas: l_1 **y** l_2, **o** **`NULL`**, si ambas líneas son paralelas. Fíjate que en la línea 9 del código revisamos el caso especial de una línea vertical. Para esto, verificamos que el valor de b sea diferente de cero **`(fabs(l1.getB()) < EPSILON)`**.

Además, podemos determinar el ángulo en que intersecan. Dadas dos líneas, $l1 : a_1x + b_1y + c_1 = 0$ y $l_2 : a_2x + b_2y + c_2 = 0$, el ángulo de intersección está determinado por:

$$tan(\theta) = \frac{a_1 b_2 - a_2 b_1}{a_1 a_2 + b_1 b_2}$$

```
1   double intersectionAngle(const Line &l1, const Line &l2) {
2     return atan ( (l1.getA() * l2.getB()) - (l2.getA() * l1.getB())
3                 / (l1.getA() * l2.getA()) - (l1.getB() * l2.getB()));
4   }
```

Usando la función anterior podemos determinar si dos líneas son perpendiculares. Decimos que dos líneas son perpendiculares si se cruzan en ángulo recto entre sí.

Por último, un problema muy útil cuando manejamos líneas y puntos es identificar el punto de una línea que está más cerca de un punto ***p*** determinado. Este punto más cercano se encuentra en la línea que pasa por ***p***, que es perpendicular a ***l***, por lo tanto, se puede encontrar usando las funciones que desarrollamos anteriormente.

```
1   Point* closestPoint(const Point &p, const Line &l) {
2     if (fabs(l.getB()) <= EPSILON) {
3       return new Point(-l.getC(), p.getY());
4     }
5
6     if (fabs(l.getA()) <= EPSILON) {
7       return new Point(p.getX(), -l.getC());
8     }
9
10    Line aux = pointAndSlopeToLine(p, 1 / l.getA());
11    return intersectsAt(l, aux);
12  }
```

6.1.3 Segmento de línea

Un segmento de línea ***s*** es la porción de una línea ***l*** que se encuentra entre dos puntos datos inclusive. Por lo tanto, los segmentos de línea se representan fácilmente por un par de puntos:

```
1   class Segment {
2   private:
3     Point p1, p2;
4
5   public:
6       Segment();
7       Segment(const Point&, const Point&);
8       Segment(const Segment&);
9
10      Point getP1() const;
11      Point getP2() const;
12  };
```

La primitiva geométrica más importante de los segmentos, que determina si un par determinado de ellos se intersecan, resulta ser sorprendentemente complicada debido a los casos especiales que surgen. Dos segmentos pueden estar en línea paralelas, lo que significa que no se cruzan en absoluto. Un segmento puede intersecarse en el punto final de otro o los dos segmentos pueden estar uno encima del otro así que se intersecan en todo el segmento y no solo en un punto.

Que este problema tenga muchos casos especiales geométricos o degeneración complica bastante la tarea de construir implementaciones robustas de algoritmos de geometría computacional. La forma correcta de lidiar con la degeneración es basar todos los cálculos en una pequeña cantidad de primitivas geométricas cuidadosamente elaboradas.

Para la implementación del problema de intersección de segmentos usaremos las funciones sobre líneas que hemos definido previamente.

```
1   bool pointInBox(const Point &p, const Point &b1, const Point &b2) {
2     return ( (p.getX() >= std::min(b1.getX(), b2.getX())) &&
3              (p.getX() <= std::max(b1.getX(), b2.getX())) &&
4              (p.getY() >= std::min(b1.getY(), b2.getY())) &&
5              (p.getY() <= std::max(b1.getY(), b2.getY())) );
6   }
7
8   bool intersectAt(const Segment &s1, const Segment &s2) {
9     Line l1 = pointsToLine(s1.getP1(), s1.getP2());
10    Line l2 = pointsToLine(s2.getP2(), s2.getP2());
11    bool result;
12
13    if (areParallel(l1, l2)) {
14      return false;
15    }
16
17    if (areSameLine(l1, l2)) {
18      return ( pointInBox(s1.getP1(), s2.getP1(), s2.getP2()) ||
19               pointInBox(s1.getP2(), s2.getP1(), s2.getP2()) ||
20               pointInBox(s2.getP1(), s1.getP1(), s1.getP2()) ||
21               pointInBox(s2.getP2(), s1.getP1(), s1.getP2()) );
22    }
23
24    Point *p = intersectAt(l1, l2);
25    result = (pointInBox(*p, s1.getP1(), s1.getP2()) &&
26              pointInBox(*p, s2.getP1(), s2.getP2()));
27    delete p;
28    return result;
29  }
```

Como podrás observar en las líneas 9 y 10 empleamos la función **`pointsToLine`** para convertir los segmentos a líneas. A partir de estas líneas podemos determinar si son paralelas (línea 11) o la misma línea (línea 17). Después, los que nos resta es determinar si un punto *p* se encuentra dentro de una región definida por alguno de nuestros segmentos de línea. Esto se prueba más fácilmente estableciendo si el punto de intersección se encuentra en un cuadro delimitador definido alrededor de cada segmento de línea (**`pointInBox`**). Este cuadro está definido por los puntos finales de cada segmento.

La intersección de segmentos también se puede determinar fácilmente usando una primitiva para verificar si tres puntos ordenados giran en sentido antihorario. Tal primitiva la revisaremos en la siguiente sección.

6.1.4 Polígonos

Los polígonos son la estructura básica para describir formas en un plano. Un polígono es solo una colección de segmentos de línea que forman un ciclo y no se cruzan entre sí. Cuando decimos que forman un ciclo nos referimos a que el primer vértice de la secuencia es el mismo que el último.

En lugar de representar un polígono como una colección de segmentos, la forma estándar es enumerar los vértices de un polígono en sentido de las agujas del reloj o en el sentido contrario, siendo el primer vértice igual al último. De esta manera, existe un segmento entre el *i*-ésimo y *(i+1)*-ésimo punto de la cadena para $0 \leq i \leq n$. Así podemos representar implícitamente los segmentos:

```
1   class Polygon {
2   private:
3     std::vector<Point> points;
4
5   public:
6     Polygon(int n, const Point &startPoint);
7     Polygon(int n, int xs[], int ys[]);
8     Polygon(const Polygon&);
9
10    std::vector<Point> getPoints() const;
11  };
```

Los polígonos pueden ser convexos o cóncavos. Decimos que es un polígono es convexo si cualquier segmento de línea definido por dos puntos dentro P se encuentra complemente dentro de P; es decir, no hay muescas ni protuberancias de modo que el segmento, al alargarlo, no interseca un borde de este. Esto implica que todos los ángulos internos de un polígono convexo deben ser agudos; es decir, miden cómo máximo 180 grados o π radianes (Figura 6.3). Todo polígono que no es convexo se denomina cóncavo (Figura 6.4).

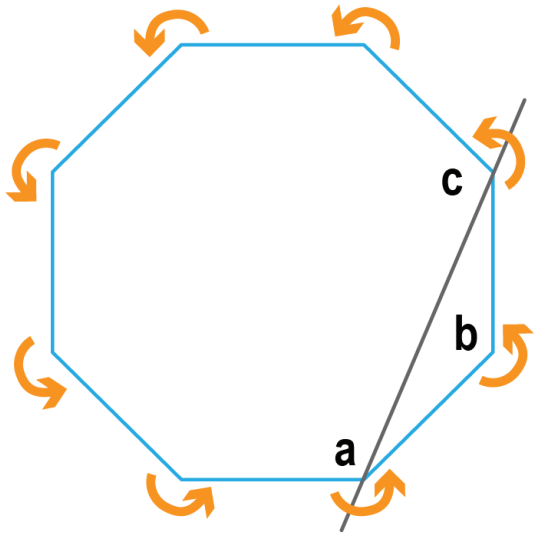


Figura 6.3 Polígono convexo

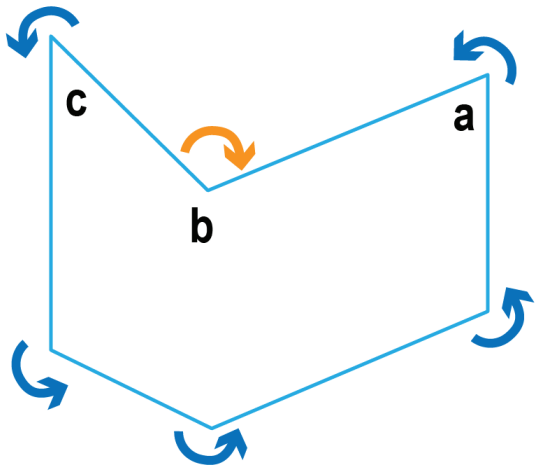


Figura 6.4 Polígono cóncavo

El perímetro de un polígono (sea convexo o cóncavo) con *n* vértices dados en algún orden (en sentido horario o antihorario) se puede calcular mediante la sencilla función que se muestra a continuación:

```
1   double perimeter(const Polygon &p) {
2     std::vector<Point> points = p.getPoints();
3     double result = 0.0;
4
5     for (int i = 0; i < points.size() - 1; i++) {
6       result += dist(points[i], points[i + 1]);
7     }
8     return result;
9   }
```

Observa que el límite superior del ciclo en la línea 5 es **`size() - 1`**, dado que el punto inicial (**`P[0]`**) es el mismo que el último (**`P[n-1]`**).

El área con signo de un polígono (convexo o cóncavo) con n vértices dados en algún orden (en sentido horario o antihorario) se puede determinar calculando el determinando de la matriz que se muestra a continuación.

$$A = \frac{1}{2} * \begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ \cdots & \cdots \\ x_{n-1} & x_{n-1} \end{bmatrix}$$

$$= \frac{1}{2} \times (x_0 \times y_1 + x_1 \times y_2 + \cdots + x_{n-1} \times y_0 - x_1 \times y_0 - x_2 \times y_1 - \cdots - x_0 \times y_{n-1})$$

Esta fórmula puede ser fácilmente escrita así:

```
1   double area(const Polygon &p) {
2     std::vector<Point> points = p.getPoints();
3     double x1, x2, y1, y2, result = 0;
4
5     for (int i = 0 ; i < points.size() - 1; i++) {
6       x1 = points[i].getX();
7       x2 = points[i + 1].getX();
8       y1 = points[i].getY();
9       y2 = points[i + 1].getY();
10      result += (x1 * y2 - x2 * y1);
11    }
12    return (fabs(result) / 2.0);
13  }
```

Como mencionamos anteriormente, un polígono es convexo si todos los ángulos internos son agudos. Es decir, cualquiera de los tres vértices del polígono *a, b*, y *c* deben formar un ángulo agudo. Para esto, debemos conocer si, al viajar del primer punto a, al segundo b, y al tercero c, giramos en sentido antihorario. Si es así, el ángulo formado entre *a* y *c* gira en sentido antihorario sobre *b* y por lo tanto es agudo (Figura 6.3). De lo contrario, el punto se encuentra a la izquierda de esa línea o los tres puntos son colineales.

Lo anterior se puede resolver calculando el área con signo del triángulo formado por los tres vértices. Un área negativa resulta si el punto *c* está a la izquierda de la línea que pasa por los puntos *a* y *b*; cero indica que los tres puntos son colineales. Y un valor positivo muestra que *c* está a la derecha.

```
1   double signedTriangleArea(const Point &a, const Point &b,
2                             const Point &c) {
3     return ((a.getX() * b.getY()) - (a.getY() * b.getX()) +
4             (a.getY() * c.getX()) - (a.getX() * c.getY()) +
5             (b.getX() * c.getY()) - (c.getX() * b.getY())) / 2.0;
6   }
7
8   bool ccw(const Point &a, const Point &b, const Point &c) {
9     return (signedTriangleArea(a, b, c) > EPSILON);
10  }
11
12  bool cw(const Point &a, const Point &b, const Point &c) {
13    return (signedTriangleArea(a, b, c) < EPSILON);
14  }
15
16  bool collinear(const Point &a, const Point &b, const Point &c) {
17    return (fabs(signedTriangleArea(a, b, c)) <= EPSILON);
18  }
```

Siendo así, la forma más sencilla saber si un polígono es convexo es comprobar si para tres vértices cualquiera y consecutivos del polígono forman giros antihorarios. Si podemos encontrar al menos un triple donde esto es falso, entonces el polígono es cóncavo.

```
1   bool isConvex(const Polygon &p) {
2     std::vector<Point> points = p.getPoints();
3     bool isLeft;
4     int p3;
5
6     if (points.size() <= 3) return false;
7
8     isLeft = ccw(points[0], points[1], points[2]);
9     for (int i = 1; i < points.size() - 1; i++) {
10      p3 = ((i + 2) == points.size())? 1 : (i + 2);
11      if (ccw(points[i], points[i + 1], points[p3]) != isLeft) {
12        return false;
13      }
14    }
15    return true;
16  }
```

Fíjate que la línea 3 se encarga de determinar si hablamos de un solo punto (**`size == 2`**) o de una sola línea (**`size == 3`**). En cuyo caso, regresamos falso.

Otra prueba común que se realiza en un polígono es verificar si un punto p está dentro o fuera del polígono. El algoritmo calcula la suma de ángulos en tres puntos: **`P[i], p, P[i + 1]`**, donde **`P[i]`** y **`P[i + 1]`** son lados consecutivos del polígono **`P`**, teniendo cuidado de verificar si los giros son antihorario (sumar el ángulo) o en el sentido de las manecillas del reloj (restar el ángulo). Si la suma final es 360 grados o 2π, entonces p está dentro del polígono.

```
1   bool inPolygon(const Point &p, const Polygon &P) {
2     std::vector<Point> points = P.getPoints();
3     double sum;
4
5     if (points.size() == 0) return false;
6
7     sum = 0;
8     for (int i = 0; i < points.size() - 1; i++) {
9       if (ccw(p, points[i], points[i + 1])) {
10        sum += angle(points[i], p, points[i + 1]);
11      } else {
12        sum -= angle(points[i], p, points[i + 1]);
13      }
14
15      return (fabs(fabs(sum) - 2*PI) < EPSILON);
16    }
```

6.2 Cascos convexos

Un casco convexo es una estructura muy importante en geometría, ya que puede ser utilizada en la construcción de muchas otras estructuras geométricas. El casco convexo de un conjunto *S* de puntos en el plano se define como el polígono convexo más pequeño que contiene todos los puntos de *S*. Los vértices del casco convexo de un conjunto *S* de puntos forman un subconjunto (no necesariamente propio) de *S*.

Un algoritmo muy simple consistiría en obtener los puntos extremos de un conjunto *S* de puntos en el plano. Para comprobar lo anterior habría que observar cada posible tripleta de puntos y ver si un *p* se encuentra en el triángulo formados por estos tres puntos. Si *p* se encuentra en cualquiera de estos triángulos, no es un punto extremo; de lo contrario lo es. La prueba para determinar si *p* se encuentra en un triángulo dado se puede hacer en un tiempo *O* (1). Dado que hay n^3 posibles triángulos, se necesitaría un tiempo O (n^3) para determinar si un punto *p* dado es extremo o no y, puesto que hay *n* puntos, este algoritmo se ejecutaría en un tiempo total de O (n^4). Sin embargo, usando la técnica de **divide y vencerás** podemos resolver el problema del casco convexo en un tiempo ***O (nh)***.

Antes de detallar cualquier algoritmo, revisemos un poco más la definición del casco convexo y algunas de sus características. Dado un conjunto *S* de puntos en el plano, el casco convexo *CH(S)* de este conjunto es el polígono convexo más pequeño *P*, para el cual cada punto del conjunto *S* está en el límite de *P* o en su interior. La intersección de conjuntos convexos es convexa, por lo que el caso convexo es un conjunto convexo. También es un politopo, una intersección acotada de medios espacios, que en dos dimensiones significa que es un polígono. Otra característica útil del casco convexo es como cierre convexo; toma todas las

combinaciones convexas posibles de los puntos del conjunto *S* y el resultado es el casco convexo.

Existen dos algoritmos para que calcular el caso convexo de un conjunto de *n* puntos. Ambos algoritmos generan los vértices del casco convexo en orden antihorario. El primero, conocido como exploración de Graham, se ejecuta en tiempo *O (n log n)*. El segundo, llamado marcha de Jarvis, se ejecuta en tiempo *O (nh)*, donde *h* es el número de vértices del casco convexo. En este libro nos enfocaremos en el algoritmo de exploración de Graham.

6.2.1 Algoritmo de exploración de Graham

El algoritmo de exploración de Graham primero ordena todos los n puntos del S basándose en los ángulos que forman con respecto a un punto llamado pivote. Un posible pivote, que es el que usaremos en este ejemplo, es el punto más inferior y más a la derecha posible (Figura 6.5).

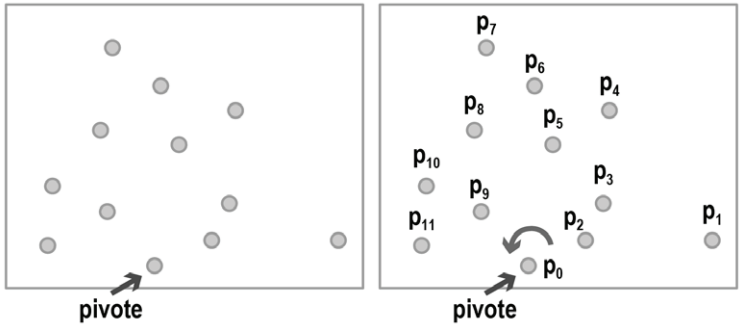


Figura 6.5 Ordenamiento de los puntos con respecto al pivote

Una vez ordenados, el algoritmo usa una pila *R* para mantener los puntos candidatos. Cada punto de *S* se colocar en el tope de la pila *R* y los puntos que no van a ser parte de *CH(S)* eventual-

mente son extraídos de *R*. El algoritmo de exploración de Graham mantiene esta invariante: los tres elementos superiores de la pila *R* siempre deben girar a la izquierda (que, como recordarás, es una propiedad básica de un polígono convexo).

Inicialmente, insertaremos estos tres puntos: p_{n-1}, p_0 y p_1, esto debido a que el ordenamiento nos asegura que estos tres puntos siempre forman un giro a la izquierda. A continuación, intentaremos insertar el punto p_2 (Figura 6.6) y, dado que p_0-p_1-p_2 forman un giro a la izquierda, lo colocamos en la pila *R*. Ahora, *R* contiene, de arriba abajo, los puntos p_{11}-p_0-p_1-p_2.

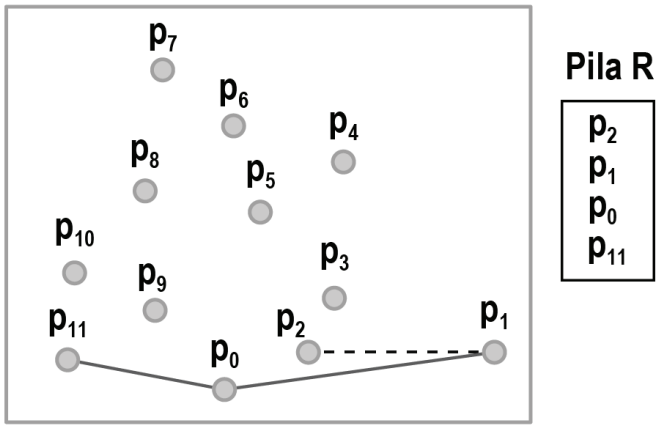


Figura 6.6 Agrandando el punto 2

A continuación, intentamos insertar el punto 3 (Figura 6.7). Sin embargo, p_1-p_2-p_3 forman un giro a la derecha, esto significa que, si aceptamos el punto antes de p_3, que es p_2, no tendremos un polígono convexo. Así que tenemos que sacar p_2 de la pila. La pila *R* ahora contiene, de arriba abajo p_{11}-p_o-p_1. Una vez hecho lo anterior volveremos a intentar colocar p_3. Ahora p_o-p_1-p_3 forman un giro a la izquierda, por lo que colocamos p_3 en la pila. La pila *R* es ahora, de arriba abajo p_{11}-p_0-p_1-p_3.

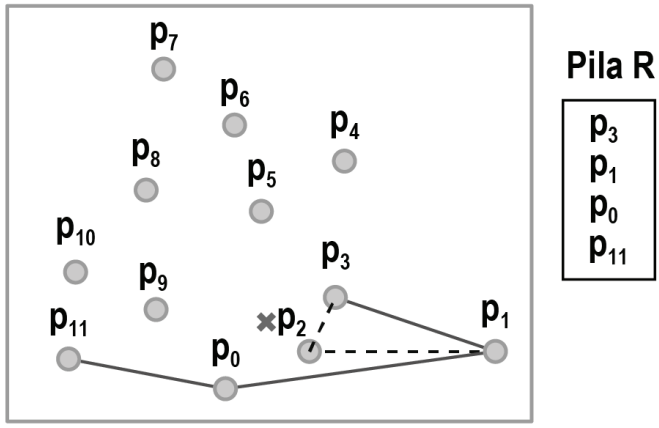


Figura 6.7 Agrandando el punto 3

Repetiremos este proceso hasta que hayamos procesado todos los vértices. Cuando terminamos el algoritmo de exploración de Graham (Figura 6.8) lo que queda en la pila *R* son los puntos de *CH(S)*. El algoritmo elimina todos los giros a la derecha, por lo que tenemos un polígono convexo.

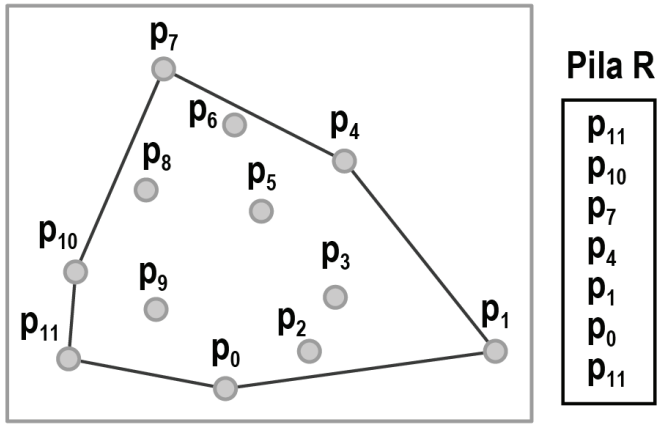


Figura 6.8 Resultado final del algoritmo de exploración de Graham

```
bool compare(const Point &a, const Point &b) {
  double d1x, d2x, d1y, d2y;

  if (collinear(pivot, a, b)) {
    return dist(pivot, a) < dist(pivot, b);
  }

  d1x = a.getX() - pivot.getX();
  d2x = b.getX() - pivot.getX();
  d1y = a.getY() - pivot.getY();
  d2y = b.getY() - pivot.getY();
  return ((atan2(d1y, d1x) - atan2(d2y - d2x)) < 0);
}

vector<Point> CH(vector<Point> S) {
  int i, j, n;

  n = S.size();
  if (n <= 3) {
    if (S[0] != S[n - 1]) {
      S.push_back(S[0]);
    }
    return S;
  }

  // encuentra el punto más inferior y más a la derecha posible
  int p0 = 0;
  for (i = 0; i < n; i++) {
    if ((S[i].getY() < S[p0].getY()) ||
        (S[i].getY() == S[p0].getY() &&
         S[p0].getX() > S[p0].getX()) {
      p0 = i;
    }
  }

  // intercambia S[p0] con S[0]
  Point aux = S[0];
  S[0] = S[p0];
  S[p0] = aux;

  // ordenamos los puntos con respecto al pivote
  pivot = S[0];
  // no necesitamos ordenadar el punto 0, ya está en su lugar
  sort(++S.begin(), S.end(); compare);

  vector<Point> result;
  result.push_back(S[n - 1]);
  result.push_back(S[0]);
  result.push_back(S[1]);
  i = 2;
  while (i < n) {
    j = result.size() - 1;
    if (ccw(result[j - 1], result[j], S[i])) {
      // se acepta el punto i
      result.push_back(S[i]);
      i++;
    } else {
      // se elimina el tope hasta que tengamos un giro a la
      // izquierda
      result.pop_back();
    }
  }
  return result;
}
```

6.3 Diagramas de Voronoi y triangulación de Delaunay

6.3.1 Diagramas de Voronoi

El diagrama de Voronoi (en honor al matemático Georgy Voronoi) de un conjunto de puntos en un plano es la división de dicho plano en regiones, de tal forma, que a cada punto se le asigna una región del plano formada por aquellos puntos que son más cercanos a él que a ningún otro.

Retomemos el problema del teléfono presentado al inicio de este capítulo, aunque lo modernizaremos un poco. Supón que sí contamos con nuestro teléfono móvil y queremos realizar nuestra llamada. Nuestro proveedor de servicios de red tiene una serie de torres celulares. Cuando hagamos la llamada, ¿a cuál torre debe conectarse nuestro teléfono móvil? En teoría, debería conectarse a la torre más cercana, pero, ¿cómo sabemos cuál es la torre más cercana? Pues es ahí donde entra el diagrama de Voronoi.

Si solo hubiera una torre celular en toda la ciudad, la región de Voronoi de dicha torre sería toda la ciudad porque todos están más cerca de dicha torre que de ninguna otra puesto que no hay más. Fácil, ¿verdad? Sin embargo, qué pasaría si hubiera dos torres celulares: A y B, la ciudad quedaría divida en dos, los que están más cerca de la torre A que de la torre B (llamaremos a esta zona Vor(A)) y los que son más cercanos a la B que la A (a esta región la llamaremos Vor(B)). Bueno, ¿y los que están a la misma distancia de las dos torres? En este caso, los puntos que están a la misma distancia de ambas torres son los que están sobre una recta: la mediatriz entre los dos puntos que definen las torres en el plano y que no es más que la recta perpendicular al segmento que une A y B por el punto medio de este (Figura 6.9).

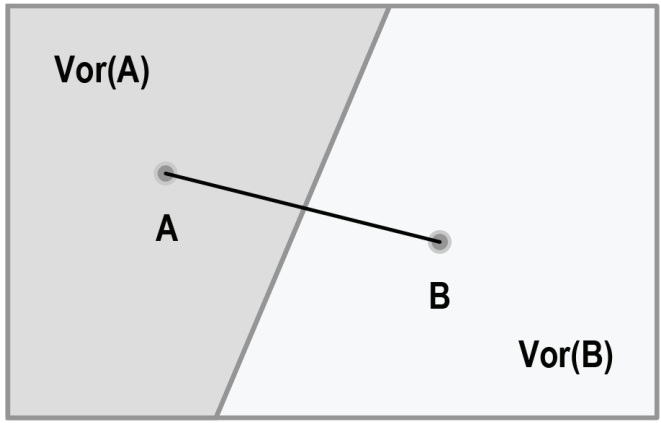


Figura 6.9 Diagrama de Vornoi de dos puntos

En el caso de 3 torres: A, B y C, razonando de manera simular y teniendo en cuenta que las mediatrices son las fronteras que delimitan las regiones de influencia de 2 a 2 como acabamos de ver. En este caso, nos quedaría una división de la ciudad en tres regiones como las que se muestran en la Figura 6.10; cada una de ellas representa la región de Voronoi de cada torre celular, es decir, la zona de la ciudad que le "corresponde" por ser la torre más cercana.

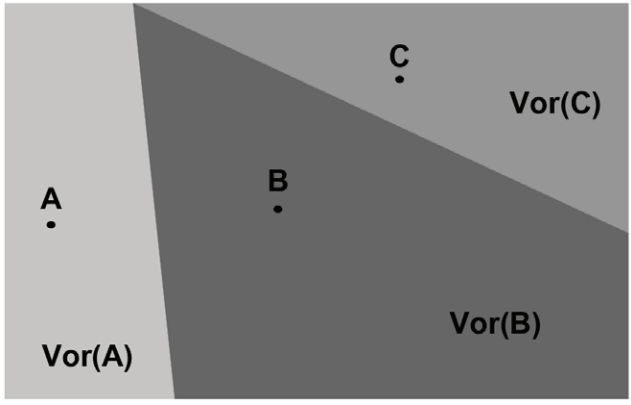


Figura 6.10 Diagrama de Vornoi de 3 puntos

Como podrás ver, el diagrama de Voronoi es una estructura inherente al concepto de proximidad o influencia. Sus aplicaciones son numerosas, no solo en las matemáticas también en varias otras disciplinas. Por ejemplo, en la epidemia de cólera de Londres de 1854, el médico John Snow utilizó un diagrama de Voronoi creado a partir de la ubicación de las bombas de agua, para contar las muertes en cada polígono e identificar una bomba en particular como la fuente de infección.

Pero, ¿cómo construimos un diagrama de Voronoi? Una solución sencilla sería la construcción de cada de región de una en una. Dado que cada región es la intersección de *n -1* semiplanos, la construcción de cada región nos tomaría un tiempo de O (n^2), lo cual significa que el algoritmo final sería $O\ (n^3)$. Sin embargo, existe un algoritmo que nos permite construir el diagrama de Voronio en *O (n log n)*. El algoritmo emplea la técnica de divide y conquista para lograr el tiempo antes mencionado. A continuación, haremos la descripción a alto nivel de este.

Algoritmo 6.1. Diagrama de Voronoi

Entrada: un conjunto *S* de *n* puntos en un plano.

Salida: el diagrama de Voronio, *V(S)*, de *S*.

1. Ordenar *S* en orden no decreciente con respecto *X*.

2. $V(S) = vd(S, 1, n)$

PROCEDURE *vd(S, low, high)*

3. Si $(high - low + 1) \leq 3$, calcula V(S) a fuerza bruta y regresa el resultado; de otra forma, continua.

4. $mid = (low + high) / 2$

5. $S_L = vd(S, low, mid)$

6. $S_R = vd(S, mid + 1, high)$

7. Construye la mediatriz *C*.

8. Descartar todas las aristas de $V(S_L)$ a la derecha de *C* y aquellas de $V(S_R)$ a la izquierda de *C*.

Sea S un conjunto de n puntos en el plano. Si n = 2 entonces en diagrama de Voronoi es la mediatriz de los puntos (Figura 6.5). De lo contrario, se divide en dos subconjuntos: S_L y S_R, que son aproximadamente del mismo tamaño (n/2). El diagrama de Voronoi $V(S_L)$ y $V(S_R)$ se calculan y combinan para obtener *V(S)*.

Se puede demostrar que el paso de combinación, que esencialmente consiste en encontrar la mediatriz, toma *O (n)*. Dado el algoritmo de ordenamiento es *O (n log n)*, el tiempo total del algoritmo sería *O (n log n)*.

6.3.2 Triangulación de Delaunay

Una triangulación de un conjunto de puntos P es un plano es una partición del casco convexo (el conjunto convexo más pequeño que contiene P) en simples triángulos de modo que:

1. La unión de todos estos triángulos es igual al casco convexo.

2. Cualquier par de estos se cruzan en una cara común (posiblemente vacía).

La triangulación de Delaunay es una de las triangulaciones más interesante por su aplicación en la solución de varios problemas aparentemente sin relación entre sí, como generación de mallas para representar terrenos en 2D o planificación de rutas para robots.

En una triangulación de Delaunay:

- Todos los puntos están conectados entre sí y forman el mayor número de triángulos posibles sin que se crucen sus aristas.

- Los triángulos se definen de forma que los puntos más próximos están conectados entre sí por una arista.

- Lo anterior, implica que los triángulos formados son los más regulares posibles, es decir, que se maximicen sus ángulos menores y se minimicen la longitud de sus lados.

La triangulación de Delaunay puede caracterizarse de varias formas que servirán para definirla y encontrar métodos que sirvan para calcular. En este libro nos enfocaremos en calcular la triangulación de Delaunay a partir del diagrama de Voronoi, que una vez conocido pueden definirse una serie de propiedades que

nos permiten calcular una triangulación de Delaunay a partir de dicho diagrama de forma directa e inequívoca. Es por eso por lo que se dice que la triangulación de Delaunay y el diagrama de Voronoi son grafos duales (Figura 6.11):

1. Cada triángulo de la triangulación se corresponde con un vértice del diagrama.

2. La triangulación y el diagrama tiene el mismo número de aristas y se corresponden entre sí.

3. Cada nodo de la triangulación se corresponde con una región del diagrama.

4. El contorno de la triangulación es equivalente al cierre convexo del conjunto de puntos.

5. No es posible encontrar ningún punto del conjunto en el interior de los triángulos formados por la triangulación.

A partir de estas propiedades nos podemos dar cuenta que hablar de diagramas de Voronoi o de triangulaciones de Delaunay es hablar de la misma estructura en lenguajes diferentes. Las propiedades de una se transforman en las propiedades de otra.

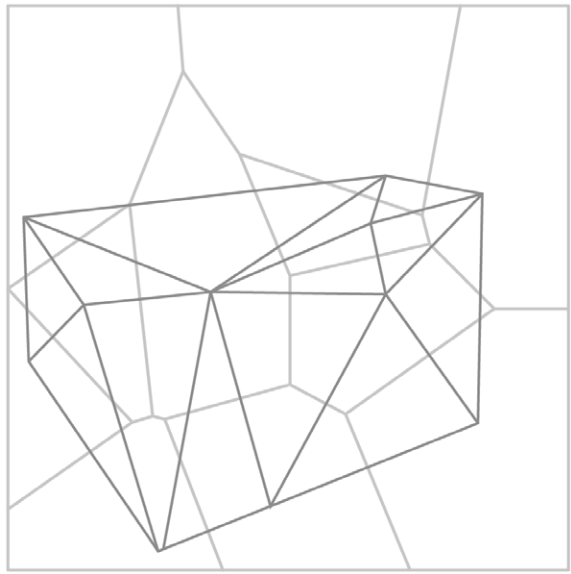


Figura 6.11 Triangulación de Dalaunay a partir de un diagrama de Voronoi

6.4 Búsqueda geométrica

Cuando hablamos de puntos en un plano es natural preguntarse cuál de esos puntos se encuentra dentro de un área específica. Por ejemplo: ¿cuáles puntos de interés hay dentro un área de un kilómetro del punto en que estoy? Si bien, el problema hace referencia a una figura geométrica (en este caso un círculo de 1 km de radio), esto se puede extender a problemas no geométricos. Por ejemplo: el problema "mostrar a todas aquellas personas entre 21 y 35 años con ingresos entre $25,000 a $100,000" hace referencia a qué "puntos" de una base de datos de personas se encuentra dentro de un área determinada de edad-ingresos.

El problema fácilmente se puede ampliar a más de dos dimensiones. Por ejemplo: enumerar todas las estrellas a 50 años luz del sol. En este caso, tenemos un problema tridimensional y, aplicado al ejemplo de la base de datos, si no solo queremos a

los jóvenes con buenos ingresos, sino que también queremos sean altos y del sexo femenino estamos hablando de un problema de cuatro dimensiones.

En general, si asumimos que tenemos un conjunto de registros con ciertos atributos que toman valores de algún conjunto ordenado, es decir, una base de datos, al encontrar todos los registros de esta base de datos que satisfagan las restricciones de rango especificadas en un conjunto específico de atributos hablamos de un problema llamado búsqueda de rango. Es una tarea difícil y un problema que, como podrás observar, tiene muchas aplicaciones prácticas.

De hecho, existen una gran cantidad de estructuras de datos y algoritmos diferentes para procesar datos multidimensionales, algunos de ellos están más allá del alcance de este capítulo. Comenzaremos revisando los árboles de rango (*range trees*) que pueden almacenar puntos multidimensionales para admitir un tipo de consulta llamada consulta de búsqueda de rango. Finalmente, analizaremos una estructura de datos llamada árboles de partición, que dividen el espacio en celdas, en particular nos enfocaremos en los árboles kd (*kd-tree*). Estas estructuras se utilizan, por ejemplo, en gráficos de computadora, donde necesitamos encontrar vecinos más cercanos a un punto de consulta o trazar la trayectoria de un rayo a través de un entorno digital.

6.4.1 Árboles de rango (Range Trees)

Como mencionamos anteriormente, una operación de consulta natural para realizar en un conjunto de puntos multidimensionales es una consulta de búsqueda de rango: **"enumerar a todas aquellas personas de sexo femenino entre 21 y 35 años con ingresos entre $25,000 a $100,000 que midan**

más de 1.70". En esta sección revisaremos la estructura de datos de árbol de rangos que se puede utilizar para responder tales consultas.

Empecemos con lo básico, consultas de búsqueda en rangos unidimensionales. Tenemos un conjunto de puntos en una sola dimensión, en decir, un conjunto de números reales. Una consulta podría ser: ¿cuáles son los puntos (o valores) que se encuentran en el intervalo $[x_1, x_2]$?

Entonces, sea $P = \{p_1, p_2, \ldots, p_n\}$ el conjunto de puntos nosotros podemos resolver el problema de búsqueda de rango unidimensional de una manera eficiente utilizando una estructura de datos conocida como árbol binario de búsqueda equilibrado *T*. Las hojas del árbol *T* almacenan los puntos de *P* y los nodos internos almacenan valores de división X_{div} para guiar la búsqueda. Consideraremos que el subárbol izquierdo de un nodo *div* contiene todos los puntos menores o iguales que X_{div} y que el subárbol derecho contiene todos los puntos estrictamente mayores que X_{div}.

Supongamos que queremos encontrar los puntos que se encuentran en el rango $[X_{low}, X_{high}]$. Sea *A* y *B* (Figura 6.8) las dos hojas donde se encuentran los límites de la búsqueda. Entonces los puntos que se encuentran en el intervalo $[X_{low}, X_{high}]$ son los que se almacenan entre las hojas *A* y *B* más, posiblemente, el punto almacenado en *A* y el punto almacenado en *B*. Por ejemplo, vamos la búsqueda [18, 78], ¿cómo podemos encontrar las hojas entre *A* y *B*? Podemos ver en la Figura 6.12 que la respuesta son las hojas de ciertos subárboles entre la búsqueda de *A* y *B*. Más específicamente, los árboles seleccionados tienen sus raíces en los nodos que están entre las dos rutas de búsqueda. Para encontrar estos dos nodos primero debemos buscar el nodo div donde las rutas hacia *A* y *B* se dividen. Esto se hace en el siguiente algoritmo. Sea *left(V)* y *right(V)* el hijo izquierdo y derecho, respectivamente, del nodo *V*.

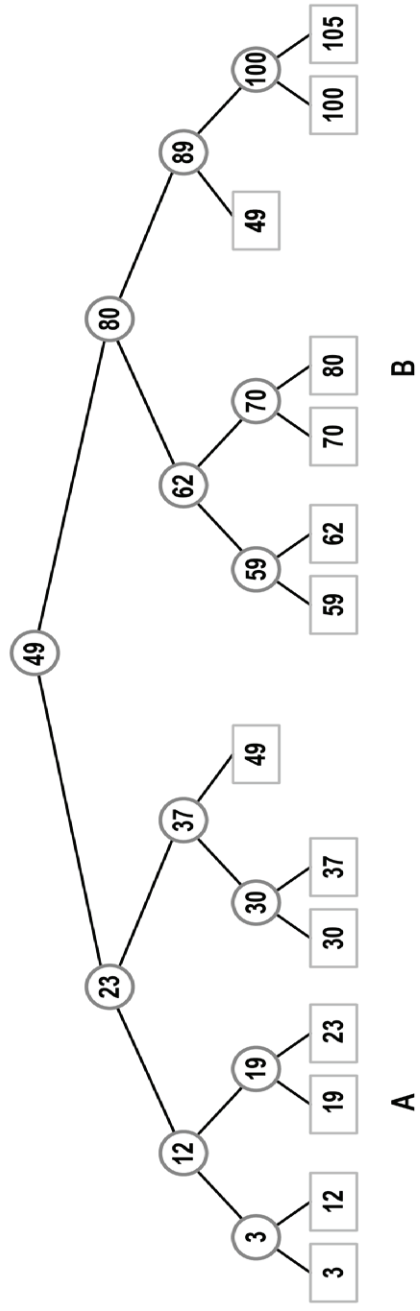


Figura 6.12 Árbol de búsqueda de rango

Algoritmo 6.2. Encontrar nodo de división (FIND_SPLIT_NODE)

Entrada: un árbol T y dos valores, *low* y *high*, siendo $low \leq high$.

Salida: el nodo V donde los caminos hacia *low* y *high* se dividen o la hoja donde ambos caminos terminan.

1. V = raíz(T)
2. Mientras V no sea una hoja y $(high \leq X_V$ o $low > X_V)$
3. Si $high \leq X_V$ entonces $V = left(V)$
4. Si no $V = right(V)$
5. Regresa V

A partir del nodo V_{split} seguimos la ruta de búsqueda de *low*; En cada nodo donde la ruta vaya hacia la derecha informamos todas las hojas en el subárbol derecho, ya que este subárbol se encuentra entre las dos rutas de búsqueda. De manera similar, seguimos el camino de *high* e informamos las hojas en el subárbol izquierdo donde el camino va a la izquierda. Por último, debemos comprobar los puntos almacenados de las hojas donde terminan los caminos con el fin de verificar si están o no en el rango [*low, high*].

En seguida encontrarás el algoritmo, aquí utilizamos la función REPORT_SUB_TREE, que recorre el subárbol y reporta los valores almacenados en sus hojas. Dado que el número de nodos internos de cualquier árbol binario es menor que su número de hojas, esta función toma una cantidad de tiempo lineal en el peor de los casos.

Algoritmo 6.3. Encontrar nodo de división (FIND_SPLIT_NODE)

Entrada: un árbol T y un rangop [*low* y *high*], siendo $low \leq high$.

Salida: el nodo V donde los caminos hacia *low* y *high* se dividen o la hoja donde ambos caminos terminan.

1. V_{split} = FIND_SPLIT_NODE (T, low, high)
2. Si V_{split} es una hoja entonces
3. Verifica si el punto almacenado en V_{split} debe ser reportado.
4. Si no
5. V = left(V_{split})
6. Mientras V no sea una hoja hacer
7. Si $low \leq X_V$ entonces
8. REPORT_SUB_TREE (*right*(V))
9. V = *left*(V)
10. Si no
11. V = *right*(V)
12. Verifica si el punto almacenado en la hoja V debe ser reportado.
13. De manera simular, sigue la ruta a *high*, informa los puntos en los subárboles a la izquierda de la ruta y verifica si se debe informar el punto almacenado en la hoja donde termina la ruta.

Ahora revisaremos el desempeño de la estructura. Debido a que hablamos de un árbol binario de búsqueda balanceado, agregar un elemento es $O\ (n \log n)$. En lo que respecta a realizar una consulta, el peor caso sería una consulta que incluyera todos los puntos almacenados. En este caso, el tiempo sería $\Theta\ (n)$, lo que no es realmente malo, sobre todo si tomamos en cuenta que comparar todos los elementos nos daría un resultado similar. Por otro lado, no se puede evitar un tiempo de consulta $\Theta(n)$ cuando tenemos que revisar todo el conjunto de puntos.

Al punto anterior, habría que agregar la llamada REPORT_SUB_TREE que es línea con los k puntos reportados. Por tanto, el tiempo total empleado en todas esas llamadas es $O\ (k)$. Dado a que el árbol está balanceado, el recorrido tiene una longitud $O\ (\log n)$. El tiempo que tomamos en revisar un nodo es $O\ (1)$, por lo que el tiempo total invertido en el recorrido de la consulta sería $O\ (\log n + k)$.

6.4.2 Árboles Kd (Kd-Trees)

Veamos ahora el problema de búsqueda de rango multidimensional. Sea P un conjunto de n puntos en el plano, para simplificar un poco el análisis, consideráremos que no hay dos puntos en P que tenga la misma coordenada en X y en Y.

En este caso, en una consulta de rango bidimensional en P solita, los puntos de P que se encuentran dentro de un rectángulo de consulta $[x_{low}, x_{high}]$ x $[y_{low}, y_{high}]$. Un punto (p_x, p_y) se encuentra dentro de este rectángulo si y solo si $p_x \in [x_{low}, x_{high}]$ y $p_y \in [y_{low}, y_{high}]$. Siendo así, podríamos decir que una consulta de rango bidimensional se compone de dos subconsultas unidimensionales: una para la coordenada x y otra para la coordenada y. ¿Podemos generalizar la estructura vista en la sección anterior

del árbol de **búsqueda binaria** unidimensional para realizar consultas bidimensionales? Bueno, revisemos nuevamente la definición del árbol de **búsqueda binaria** de la sección anterior como el conjunto de puntos unidimensionales que se divide en dos subconjuntos de aproximadamente el mismo tamaño; un subconjunto contiene los puntos menores o iguales al valor de división, en el otro subconjunto están los puntos mayores al valor de división. El valor de división se almacena de forma recursiva en los dos subárboles.

Ahora, en el caso bidimensional cada punto tiene dos valores que son su coordenada en x y su coordenada en y, por lo tanto, lo que haremos será dividir primero por la coordenada x, luego por la coordenada y, nuevamente por la coordenada x, así sucesivamente. Siendo más precisos, en el proceso es el siguiente: En la raíz dividimos el conjunto de puntos P con una línea vertical l en dos subconjuntos de aproximadamente el mismo tamaño. La línea de división se almacenará en la raíz. P_{left} es el conjunto de puntos que están a la izquierda o sobre la línea de división que estarán almacenados en el subárbol izquierdo, mientras que P_{right} será el conjunto de puntos que están a la derecha de la línea de división, estos se almacenarán en el subárbol derecho. El hijo izquierdo de la raíz, el conjunto de puntos P_{left}, lo dividimos en dos subconjuntos con una línea horizontal; los puntos por debajo o en la línea de división se almacenarán en el subárbol izquierdo del hijo izquierdo y los puntos por encima estarán el subárbol derecho. El hijo izquierdo almacenará la línea de división. De manera similar, el conjunto de punto P_{right} se divide con una línea horizontal en dos subconjuntos que se almacenan en el subárbol izquierdo y derecho del hijo derecho. En los "nietos" de la raíz se vuelve a dividir con una línea vertical. En general, dividimos con una línea vertical en los nodos con profundidad par y con una línea horizontal los de profundidad impar.

Las Figuras 6.13 y 6.14 muestran el esquema de división y cómo se ve el árbol binario correspondiente. Un árbol como este se conoce como un árbol kd. Originalmente, el nombre significaba árbol k-dimensional.

A continuación, presentamos el algoritmo para construir un árbol kd. El algoritmo recibe dos parámetros: un conjunto de puntos y un número entero. El primer parámetro es el conjunto de puntos para el que queremos construir el árbol kd. El segundo parámetro es la profundidad de la raíz del subárbol. Inicialmente este parámetro es cero en la primera llamada. Recuerda que la profundidad es necesaria, ya que nos permite determinar si debemos partir con una línea horizontal o vertical. El algoritmo regresa la raíz del árbol kd.

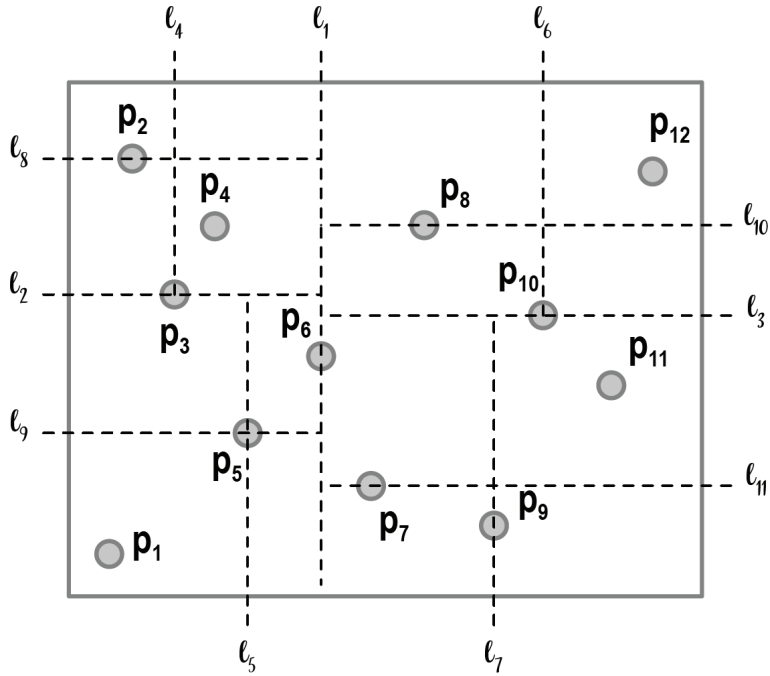


Figura 6.13 Árbol kd, la forma en que se divide el plano

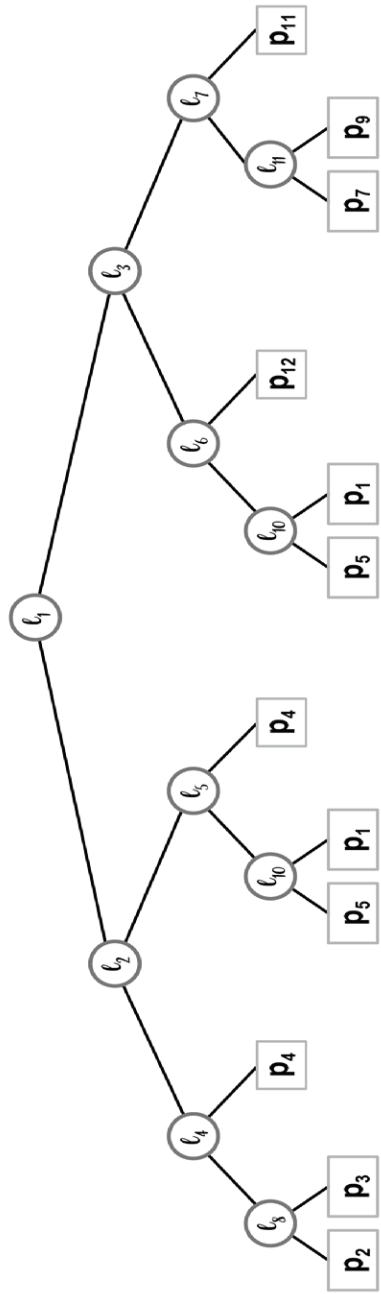


Figura 6.14 El árbol binario resultante

Algoritmo 6.4. Construir un árbol kd (BUILD_KD_TREE)

Entrada: un conjunto de puntos P y la actual profundidad, *depth*.

Salida: la raíz de un árbol kd conteniendo P.

1. Si P contiene un solo punto entonces

2. Regresa una hoja conteniendo este punto.

3. Si no

4. Si *depth* es par entonces

5. Separa P en dos subconjuntos con una línea vertical ℓ a través de la mediana de la coordenada x de los puntos en P. Sea P_1 el conjunto de la izquierda o encima de ℓ, y sea P_2 el subconjunto a la derecha de ℓ.

6. Si no

7. Separa P en dos subconjuntos con una línea vertical ℓ a través de la mediana de la coordenada y de los puntos en P. Sea P_1 el conjunto por debajo o encima de ℓ, y sea P_2 el subconjunto encima de ℓ.

8. V_{left} = BUILD_KD_TREE (P_1, *depth* + 1)

9. V_{right} = BUILD_KD_TREE (P_2, *depth* + 1)

10. Crear un nodo V conteniendo ℓ, hacer que V_{left} sea el izquierdo de V y V_{right} el hijo derecho.

Si analizamos el tiempo de construcción veremos que el paso más costoso de cada llamada recursiva sería encontrar la línea de división. Esto requiere determinar la mediana en la coordenada *x* o en la coordenada *y* depende si la profundidad es par o impar. Determinar la mediana se realiza en tiempo lineal. Un truco para lograr este tiempo es ordenar previamente los puntos tanto en la coordenada *x* como en la coordenada *y*. Por lo tanto, ahora en vez de recibir solo el conjunto *P*, se pasarán dos listas ordenadas: una en la coordenada *x* y otra en la coordenada *y*. Tomando en cuenta todo lo anterior, podemos determinar que la construcción del árbol sería *O (n log n)*. Este límite incluye el tiempo que dedicamos al ordenamiento previo de los puntos en las coordenadas *x* e *y*.

Ahora vamos a revisar cómo realizar una búsqueda en este tipo de árboles. La línea de división almacenada en la raíz divide el plano en dos semiplanos. Los puntos del semiplano izquierdo se almacenan en el subárbol izquierdo y los puntos del semiplano derecho se almacenan en el subárbol derecho. Lo anterior se aplica para cada uno de los niveles del árbol, es decir, el hijo izquierdo del hijo izquierdo de la raíz, por ejemplo, corresponde a la región delimitada a la derecha por la línea de división almacenada en la raíz y delimitada desde arriba por la línea almacenada en el hijo izquierdo de la raíz (puntos p_3 y p_4 de la Figura 6.9). En general, la región correspondiente a un nodo *V* es un rectángulo que encuentra delimitada en, al menos, dos lados por las líneas almacenadas en los ancestros de *V*. A este rectángulo se le llama *región(V)*.

Tomando en cuenta lo anterior al realizar una búsqueda tenemos que encontrar el subárbol con raíz en *V* solo si el rectángulo de consulta interseca la *región(V)*. Para ello, recorremos el árbol kd, pero visitando solo los nodos cuya región está complemente contenida en el rectángulo de consulta. Cuando una

región está completamente contenida en el rectángulo, podemos reportar todos los puntos almacenados en su subárbol. Cuando el recorrido llega a una hoja tenemos que comprobar si el punto está contenido en la región de consulta.

Algoritmo 6.5. Búsqueda en un árbol kd (SEARCH_KD_TREE)

Entrada: la raíz de un árbol (o subárbol) kd, y un rango *R*.

Salida: todos los puntos de las hojas que se encuentran en el rango *R*.

```
1. Si V es una hoja entonces
2.     Reporta el punto almacenado en V si se encuentra en la región R.
3. Si no
4.     Si region(left(V)) se encuentra contenido en R entonces
5.         REPORT_SUB_TREE(left(V))
6.     Si no
7.         Si region(left(V)) interseca en R entonces
8.             SEARCH_KD_TREE(left(V))
9.     Si region(right(V)) se encuentra contenido en R entonces
10.        REPORT_SUB_TREE(right (V))
11.    Si no
12.        Si region(right (V)) interseca  en R entonces
13.            SEARCH_KD_TREE(right (V))
```

El algoritmo de búsqueda es un procedimiento recursivo que toma como argumentos la raíz de un árbol kd y el rango de búsqueda R. Utiliza el procedimiento REPORT_SUB_TREE(V) que recorre el subárbol V e informa todos los puntos almacenados en sus hojas.

6.5 Ejercicios del capítulo 6

1. Calcula la ecuación lineal que pasa por dos puntos (2, 2) y (2, 4).

2. Dados tres puntos: a (2, 2), b (2, 4) y c (4, 3) calcula el ángulo *abc* en grados.

3. Determina si el punto *r* (35, 30) está en el lado izquierdo de, colineal con, o está en el lado derecho de una línea que pasa por dos puntos *p* (3, 7) y q (11, 13).

4. Sea *S* un conjunto de *n* puntos en el plano. Diseña un algoritmo *O (n log n)* para calcular para cada punto *p* el número de puntos en *S* dominados por *p*.

5. Considera el problema de intersección de segmentos de línea: Dados *n* segmentos de línea en el plano, determina si dos de ellos se intersecan. Proporciona un algoritmo de tiempo *O (n log n)* para resolver este problema.

6. Desarrolla un algoritmo para encontrar cuando dos polígonos intersectan.

Capítulo 7. Técnicas de búsqueda avanzada.

Los algoritmos que hemos analizado en capítulos anteriores se encuentran diseñados para explorar espacios de búsqueda de forma sistemática. No poseen información adicional sobre el problema más allá de la que se proporciona en la definición de este. Todo lo que pueden hacer es generar posibles soluciones parciales y determinar si se ha llegado, o no, al resultado final. A este tipo de búsqueda se les conoce como búsquedas ciegas.

Este tipo de búsquedas se realizan manteniendo uno o más caminos en la memoria y "recordando" las alternativas han sido exploradas en cada paso del proceso de solución. Por lo regular, cuando se encuentra una solución, el camino empleado también es parte de la solución del problema.

Sin embargo, estos algoritmos pueden ser mejorados usando técnicas de programación o estructuras de datos que nos permiten realizar una búsqueda más informada o reducir la memoria empleada en el proceso de generación de la solución. Por ejemplo, las técnicas de *backtracking* y programación dinámica se ven muy beneficias con el uso de máscaras de bits para almacenar información. En este capítulo hablaremos de algunas de ellas.

Además, conoceremos algunos algoritmos que emplean técnicas que permiten realizar una mejor selección de las alternativas generadas, lo que posibilita reducir el espacio de búsqueda y llegar más rápidamente a nuestra solución. Este tipo de búsquedas son conocidas como informadas y emplean heurísticas para este fin.

7.1 Backtracking con Bitmask

En la sección 3.4.3 conocimos el problema de la suma de subconjuntos.

Problema de la suma de subconjuntos

Dado un conjunto *S* de *N* números enteros positivos, encontrar los subconjuntos que sumen la cantidad *C*. Si no se encuentra algún subconjunto que cumple se debe responder con el conjunto vacío.

En aquella ocasión vimos que, básicamente, la solución de problema radica en dada una colección de N elementos debemos elegir un subconjunto cuya suma sea C. Para indicar si elemento X pertenece a un subconjunto Y podemos usar algunas de las siguientes opciones:

- Un mapa (o arreglo asociativo) que nos indique si el objeto ha sido seleccionado.
- Si los elementos pueden ser "indexados por números enteros" se podría usar un arreglo de booleanos.
- O, como en aquella ocasión, utilizar la estructura de datos *set*.

```
1  #include <iostream>
2  #include <set>
3  #include <vector>
4
5  using namespace std;
6
7  vector<int> S{2,3,7,9};  // conjunto de números
8  int C = 9;  // suma deseada
9
10 void imprimeConjunto(set<int> s){
11   cout << "{";
12   for (int i : s)
13     cout << i << " ";
14   cout << "}\n";
15 }
16
17 void backtracking(set<int> s, int suma, int nivel, int c){
18   if (suma > c) return; //se termina la exploración de esa
19                         // rama
20   if (suma == c){       // es una solución
21     imprimeConjunto(s);
22     return;
23   }
24   if (nivel < S.size()){  // todavía no se llega al final
25     backtracking(s, suma, nivel+1, c);
26     s.insert(S[nivel]);
27     backtracking(s, suma+S[nivel], nivel+1, c);
28   }
29 }
30
31 int main(){
32   backtracking({}, 0, 0, C);
33 }
```

En aquella ocasión no realizamos el análisis de tiempo de ejecución de la implementación. Sin embargo, con el fin de resaltar el impacto del uso de una máscara de bits en esta, lo haremos esta vez. Si recuerdas, comentamos que el tiempo de ejecución de este tipo de problemas era $O(2^n)$. No obstante, esa afirmación no es completamente cierta en este caso. Fijémonos en la línea 26 del código, como puedes observar hay una llamada al método *insert* de *set*. Si revisamos la documentación de C++ de este método encontraremos que el tiempo de ejecución es $O(n \log n)$. Si tomamos en consideración que cada ejecución implica una llamada a este método, tendríamos que el tiempo de ejecución de esta función sería $O(n \log n * 2^n)$, lo que no es óptimo que digamos.

Entonces, ¿qué podemos hacer para reducir este tiempo? La respuesta más sencilla sería substituir el método *insert* por alguna operación o función más eficiente. Quizás algo $O(1)$, para que el tiempo de ejecución no se vea tan afectado. Y es aquí donde entran las máscaras de bits. Las máscaras de bits se emplean en la solución de problemas que requieren de tiempo o memoria exponencial, que resulta ser nuestro caso.

Una máscara de bits no es más que un número binario que representa algo. Tomemos como ejemplo el problema de la suma de subconjuntos. Dado un conjunto inicial $A = \{x_1, x_2, \ldots, x_n\}$ queremos conocer qué elementos del conjunto suman la cantidad *C*. Es decir, cuáles elementos han sido seleccionados. En este caso, supongamos que el conjunto inicial es $A = \{2,3,7,9\}$. Podemos representar cualquier subconjunto de A usando una máscara de 4 bits. En esta secuencia de bits, el elemento i-ésimo se considera seleccionado si y solo si el i-ésimo bit de la máscara está prendido, es decir, es igual a 1. Supongamos que queremos representar el subconjunto $X = \{2, 3, 9\}$, si consideramos que el elemento 1 es el bit menos significativo, el elemento 2 es el bit que está a la izquierda y así sucesivamente, la máscara que debemos usar para representar el subconjunto sería b = 00001011.

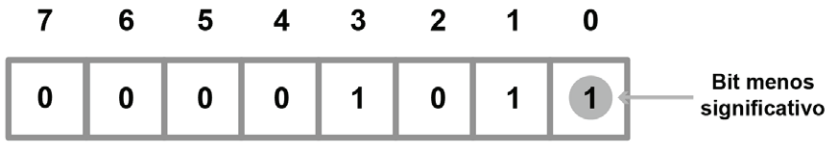


Figura 7.1

En este caso, estamos empleando un variable de tipo char para representar el conjunto, ya que no existe un tipo de dato de 4 bits. Como podemos ver, en la máscara se encuentran "pren-

didos" los bits 0, 1 y 3. Observa cómo para representar un subconjunto se requiere solo de 8 bits, lo que resulta en un uso más eficiente de la memoria.

Las operaciones básicas que podemos realizar sobre una máscara son: establecer (prender un bit), remover (apagar un bit) o verificar (determinar si un bit está prendido). Para realizar estas operaciones usaremos los operadores lógicos or (|), and (&), desplazamiento a la izquierda (<<) y desplazamiento a la derecha (>>).

Digamos que tenemos la máscara del ejemplo anterior b = 00001011:

- Si queremos establecer el i-ésimo bit de una máscara usaremos la operación: b | (1 << i). Por ejemplo si queremos establecer el tercer elemento i = 2:

```
1 << 2 = 00000100
00001011 | 00000100 = 0001111
```

Así que ahora, el subconjunto también incluye al tercer elemento X = {2, 3, 7, 9}.

- Si queremos remover el i-ésimo bit de una máscara debemos usar la siguiente operación: b & !(1 << i). Digamos que ahora queremos remover el segundo elemento i = 1 del subconjunto:

```
1 << 1 = 00000010
!00000010 = 11111101
00001111 & 11111101 = 00001101
```

De esta forma, la nueva máscara representará X = {2, 7, 9}.

- Si queremos verificar si el i-ésimo bit está prendido usaremos la siguiente operación: b & (1 << i). Si el i-ésimo bit está prendido el resultado será diferente de cero. Supongamos que queremos verificar si el cuarto elemento se encuentra en el subconjunto i = 3:

```
        1 << 3 = 00001000
00001101 & 00001000 = 00001000
```

Como el resultado es igual a 1 podemos determinar que el cuarto elemento se encuentra en el subconjunto X.

A continuación, mostramos cómo quedaría el código anterior usando máscara de bits.

```cpp
#include <iostream>
#include <vector>

using namespace std;


void imprimeConjunto(set<int> s){
  cout << "{";
  for (int i : s)
    cout << i << " ";
  cout << "}\n";
}
typedef unsigned char uchar;

uchar add(uchar mask, int pos) {
  return (mask | (1 << pos));
}

uchar remove(uchar mask, int pos) {
  return (mask & !(1 << pos));
}

bool test(uchar mask, int pos) {
  return ((mask & (1 << pos)) != 0);
}

void imprimeConjunto(vector<int> &S, uchar mask){
  cout << "{";
  for (int i = 0; i < S.size(); i++) {
    if (test(mask, i)) {
      cout << S[i] << " ";
    }
  }
  cout << "}\n";
}

void backtracking(vector<int> &S, uchar mask, int suma, int
nivel, int c){
  if (suma > c) return;  // se termina la exploración de esa
                         // rama
  if (suma == c){        // es una solución
    imprimeConjunto(S, mask);
    return;
  }
  if (nivel < S.size()){ // todavía no se llega al final
    backtracking(S, mask, suma, nivel + 1, c);
    mask = add(mask, nivel);
    backtracking(S, mask, suma + S[nivel], nivel + 1, c);
  }
}

int main(int argc, char* argv[]){
  vector<int> S{2,3,7,9};  // conjunto de números
  int C = 9;               // suma deseada
  uchar set = 0;           // la máscara inicial

  backtracking(S, set, 0, 0, C);
}
```

7.2 Encontrarse en el medio

La técnica de **encontrarse en el medio** divide el espacio de búsqueda en dos partes de aproximadamente el mismo tamaño, realiza una búsqueda separada para ambas partes y, finalmente, combina los resultados de las búsquedas. **Encontrarse en el medio** nos permite acelerar ciertos algoritmos de tiempo $O(2^n)$ para que funciones en un tiempo $O(2^{n/2})$. Pareciera que no es mucho más rápido que el tiempo original. Sin embargo, toma en cuenta que $2^{n/2} = \sqrt{2^n}$. Usando un algoritmo $O(2^n)$ podemos procesar entradas donde n ≈ 20, mientras que usando un algoritmo $O(2^n/2)$ nos permite procesar entradas de cercana a n ≈ 40.

Retomemos el problema anterior, supongamos que nos dan un conjunto de *n* números enteros y queremos determinar si el conjunto tiene un subconjunto con la suma *C*. Por ejemplo, dado el conjunto S = {2, 3, 7, 9} y C = 9 podemos elegir el subconjunto {2, 7} porque 2 + 7 = 9. Ya hemos mencionado antes que el algoritmo de *backtracking* que resuelve este problema tiene un tiempo de ejecución $O(2^n)$, pero ahora lo resolveremos usando la técnica **encontrarse en el medio**.

La idea es dividir nuestro conjunto en dos conjuntos *A* y *B* de modo que ambos conjuntos contengan aproximadamente la mitad de los números. Realizamos dos búsquedas: la primera búsqueda genera todos los subconjuntos de *A* y almacena sus sumas en una lista S_A y la segunda búsqueda crea una lista S_B similar para *B*. A continuación, basta con verificar si podemos elegir un elemento de S_A y otro de S_B tal que la suma sea *C*, lo cual es posible cuando el conjunto original contiene un subconjunto con suma *C*.

Por ejemplo, veamos cómo se procesa el conjunto $S = \{2, 3, 7, 9\}$. Primero, dividimos el conjunto en conjuntos $A = \{2, 3\}$ y $B = \{7, 9\}$. En seguida, creamos las listas $S_A = [\{\} = 0, \{2] = 2, \{3\} = 3, \{2, 3\} = 5]$ y $S_B = [\{\} = 0, \{7\} = 7, \{9\} = 9, \{7,9\} = 16]$. Dado que S_A contiene la suma 2 y S_B contiene la suma 7 concluimos que el conjunto original tiene un subconjunto con suma 2 + 7 = 9.

Con una buena implementación podemos crear las listas S_A y S_B en tiempo $O(2^{n/2})$ de forma tal que las listas estén ordenadas. Después, podemos usar la técnica de dos punteros para verificar en tiempo $O(2^{n/2})$ si la suma C puede ser creada a partir de S_A y S_B. Por tanto, la complejidad del algoritmo es $O(2^{n/2})$.

7.3 Búsqueda A*

Una de las principales operaciones que se puede hacer sobre un grafo es buscar un nodo específico, es decir, aquel que contiene la información que se desea encontrar como se muestra en la Figura 7.2. La forma *naive* de realizar esta búsqueda es moverse de un nodo a otro hasta que se logre encontrar el nodo deseado o se visiten todos y no se encuentre el nodo en cuestión, en cuyo caso, el algoritmo contestará que el nodo no existe en el grafo. Esta búsqueda *naive* parece muy simple, pero no lo es debido a que el grafo es una estructura de datos no lineal (esto es, cada nodo puede tener 0 o más antecesores y 0 o más sucesores) y no tiene un inicio o un fin. Cada nodo puede tener varios **vecinos**. Hay que recordar que un nodo es vecino de otro si ambos están conectados por una **arista**, por ejemplo, en la Figura 7.2: los nodos **{S, A, C, E}** son vecinos del nodo **B**. Estando en un nodo se tiene que decidir hacia cuál de sus vecinos se debe mover para seguir buscando, además de que se tiene que marcar cuál

nodo ya visitamos para no regresar a él y caer un ciclo. La forma de solucionarlo es establecer un **método sistemático de búsqueda** que garantice pasar por todos los nodos sin repetirlos.

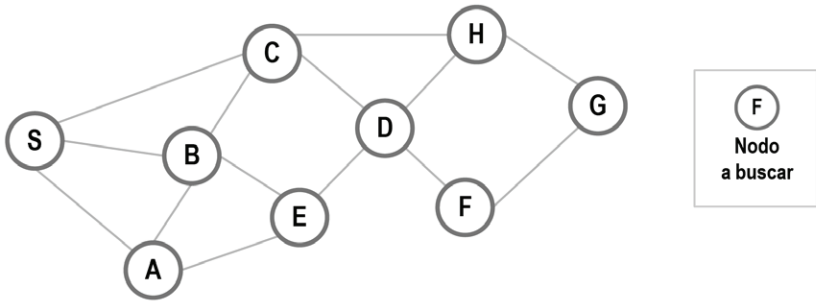


Figura 7.2 Ejemplo de un grafo y un nodo a ser buscado en el mismo

Si para hacer la búsqueda se parte de un nodo y, en caso de no ser el buscado, se procede a buscar en los vecinos de este nodo, luego en los vecinos de los vecinos y así sucesivamente, se garantizará recorrer todos en algún momento, siempre y cuando el grafo sea finito. Esto genera una estructura de búsqueda en forma de **árbol**, al que se le conoce como **árbol de búsqueda**, en el que su raíz es el nodo donde se inicia la búsqueda y su factor de ramificación depende del número de vecinos que tengan los nodos. La Figura 7.3 muestra una parte del árbol de búsqueda en el grafo mostrado en la Figura 7.2 partiendo del nodo **S**.

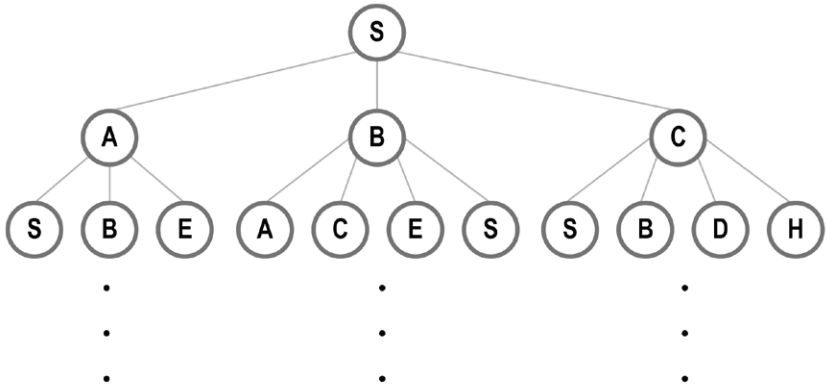


Figura 7.3 Árbol de búsqueda parcial para el grafo de la Figura 7.2

En la Figura 7.3 se puede observar que estando en un nodo sus hijos son sus vecinos. Por ejemplo: estando en el nodo **A**, sus vecinos son **{S, B, E}**, estos se convierten en sus hijos en el árbol de búsqueda.

El nodo que se está visitando en un momento dado de la búsqueda se conoce como **nodo actual**. Si el nodo actual no es la solución se procede a visitar a sus vecinos quitando los que ya fueron visitados. A esa acción se le conoce como **expandir el nodo**, es decir, generar todos sus vecinos y tenerlos listos para seguir la búsqueda. Todos los algoritmos de búsqueda sistemática realizan esa acción, la única diferencia entre ellos es la decisión de **cuál de los vecinos** se debe visitar (o explorar) a continuación. El algoritmo general de búsqueda sistemática se puede ver en el Algoritmo 7.1.

Algoritmo 7.1 Búsqueda sistemática

Entrada: el grafo donde se va a hacer la búsqueda y el valor que se quiere buscar o sus características.

Salida: el apuntador al nodo que contiene el valor buscado o **nulo** en caso de que no lo encuentre.

1. Colocar el **Nodo Inicial** en la **Estructura**.

2. Si la **Estructura** está vacía regresar nulo (no hay solución), ir a FIN.

3. Sacar un Nodo de la **Estructura** y hacerlo es **Nodo Actual.**

4. Si el **Nodo Actual** ya fue visitado ir a 2.

5. Si **Nodo Actual = Nodo Buscado**, regresar **Nodo Actual** (se encontró la solución), ir a FIN.

6. Marcar el **nodo** como visitado.

7. Expandir el **Nodo Actual** y colocar sus hijos (vecinos) en la **Estructura**.

8. Ir a 2

FIN

Existen dos algoritmos clásicos de búsqueda sistemática, los cuales no requieren más información que la definición misma del problema:

- **Búsqueda primero en anchura** (BFS, por sus siglas en inglés, *Breath First Search*) el cual visita primero el nodo menos profundo en el árbol de búsqueda. Si hay empa-

te lo resuelve con alguna regla. Esto lo lleva a explorar primero todos los nodos que están a la misma profundidad, luego los del siguiente nivel y así sucesivamente. Esta exploración por niveles de profundidad es lo que le da su nombre.

- **Búsqueda primero en profundidad** (DFS, por sus siglas en inglés, *Depth First Search*) el cual visita primero el nodo más profundo. Si hay empate lo resuelve con alguna regla. Esta exploración privilegiando a los nodos más profundos, es lo que le da su nombre.

Ambos métodos usan el mismo Algoritmo 7.1, la diferencia la hace la "Estructura" que usan para ir guardando los vecinos generados. EL BFS usa una **cola** (*queue*), lo que garantiza que todos los nodos del mismo nivel se exploran primero, ya que los primeros nodos colocados son los primeros en salir (FIFO). El DFS usa una **pila** (*stack*), lo que garantiza que el más profundo se explore primero, porque los recientemente generados son los que acaban de entrar a la pila y, por lo tanto, son los primero en salir (LIFO).

El problema fundamental de estos dos algoritmos es que no usan ninguna información extra que se pueda tener para guiar la búsqueda. Por ejemplo, si las aristas del grafo están pesadas, como el grafo de la Figura 7.4, y estos pesos representan distancias, no toma en cuenta dicha distancia y podría no encontrar una solución con la distancia mínima recorrida.

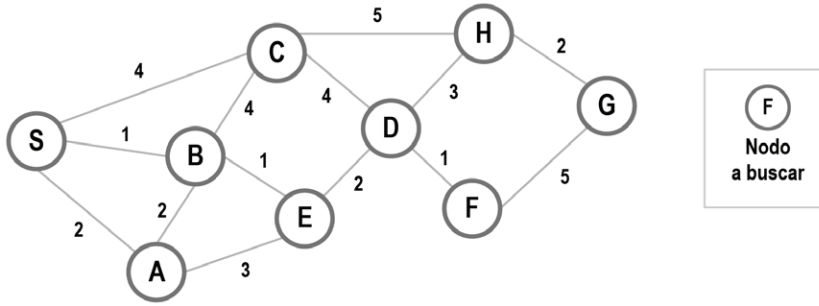


Figura 7.4 Grafo pesado con el nodo a ser buscado

Hay un algoritmo sistemático que intenta aprovechar la información de las aristas. Este algoritmo es una modificación de BFS, el cual usa una función **c(n)**, llamada **costo** (donde **n** es el nodo), que calcula el costo que hay de la raíz al nodo actual sumando los valores de las aristas por las que se ha pasado, los cuales son considerados costos:

- **Búsqueda de costo uniforme** (UCS, por sus siglas en inglés, *Uniform Cost Search*), expande primero el nodo de menor costo (BFS expande primero el de menor profundidad), el cual es calculado con la función **c(n)**, donde **n** es el nodo.

UCS logra encontrar el camino con el menor costo para ir desde el nodo inicial al nodo objetivo. Usa el mismo Algoritmo 7.1, solo que la estructura es una cola con prioridad (min-Heap), donde la prioridad es el costo **c(n)**, de esta forma, el siguiente nodo en salir de la estructura para ser explorado es el de menor prioridad, esto es, el de menor costo.

La Figura 7.5 muestra el árbol parcial de búsqueda usando UCS partiendo del nodo S del grafo de la Figura 7.4 con el costo **c(n)** calculado para cada nodo **n**.

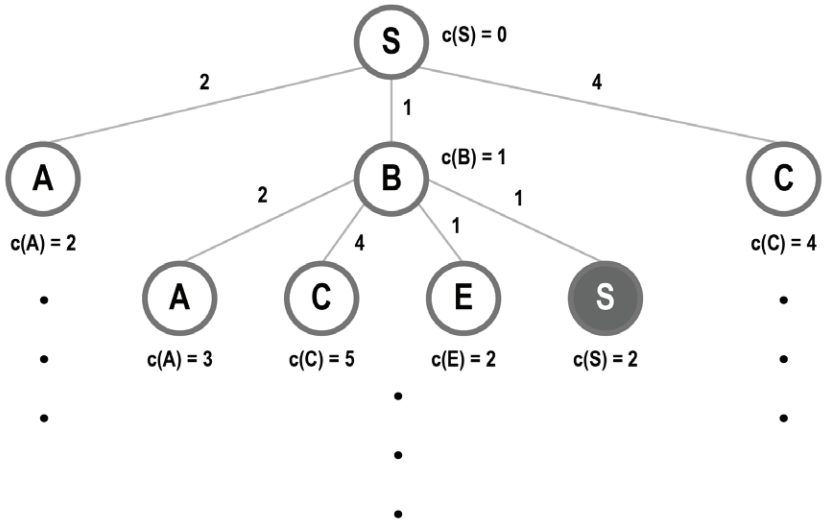


Figura 7.5 Árbol parcial de búsqueda con su valor **c(n)** para cada nodo **n**

En la Figura 7.5 se puede observar que el método UCS inicia en el nodo S y como no es la solución procede a su expansión, por lo que ese nodo se tiene que marcar como ya visitado. Genera sus tres hijos y les calcula sus costos colocándolos en una cola con prioridad, lo que significa que el siguiente nodo a ser explorado es el de menor costo, en este caso el nodo B con costo 1. Como ese nodo no es la solución se procede a expandirlo. Se generan sus hijos y se meten a la cola con prioridad. A continuación, se expande el de menor costo. En este caso podrían ser: el A, del primer nivel; el E, del segundo nivel o el S del segundo nivel. Supongamos que se selecciona el S, marcado con rojo. Puesto que ya fue visitado, se desecha, se selecciona otro y el proceso continúa.

Se pensó que se podrían generar algunos algoritmos de búsqueda sistemática que fueran más eficientes al considerar alguna información extra sobre el problema que se tuviera a la mano y que diera alguna idea de cuál es el nodo más **prometedor** para ser explorado, es decir, cuál podría llegar más rápido a la solución buscada. Así aparecieron los **algoritmos heurísticos**,

los cuales utilizan una función para estimar la distancia a la que un nodo específico está del objetivo. Esta función es una función heurística **h(n)**, donde **n** es el nodo, lo que les da su nombre. El algoritmo básico de este nuevo grupo se llama:

- **Búsqueda primero el mejor** (BFS, por sus siglas en inglés, *Best First Search*) expande primero el que se estima que le falta menos para llegar al objetivo, esto es, el que tenga un menor valor de la función heurística **h(n)**.

BFS usa el mismo Algoritmo 7.1 con una cola con prioridad (min-Heap) como estructura, donde la prioridad es el valor de la heurística **h(n)**, se exploran primero los nodos con menor heurística, esto es, a los que se estima que les falta menos para llegar al objetivo.

El problema principal de los algoritmos heurísticos es, precisamente, el diseño de la función heurística **h(n)**. No es simple diseñar una función heurística buena, ya que, normalmente, esta depende del tipo de problema que se está resolviendo. Una heurística debe cumplir con la condición de que **si n es el nodo meta**, la heurística debe regresar el **valor de 0**. Si la heurística fuera perfecta, el algoritmo llegaría al objetivo en el menor tiempo posible.

Mientras UCS no toma en cuenta lo que falta para llegar al objetivo y BFS no toma en cuenta lo que se lleva recorrido, existe un algoritmo llamado A* que utiliza una combinación entre UCS y BFS para estimar el camino completo usando una nueva función **f(n) = c(n) + h(n)**, donde **c(n)** es el costo de UCS y **h(n)** es la heurística de BFS:

- **A*** expande primero los nodos que tengan una menor **f(n)**, es decir, el nodo con el camino completo desde la raíz al objetivo con el menor costo estimado.

A* utiliza el mismo Algoritmo 7.1 con una cola con prioridad como estructura (min-Heap), donde la prioridad está dada por **f(n)** con lo que el nodo que sale primero es el que tenga la menor distancia total estimada.

A* tiene una propiedad interesante: Si **h(n)** cumple ciertas características A* es óptimo, esto es, logra encontrar el camino más corto para ir del nodo inicial al nodo objetivo. Las características que debe cumplir la heurística **h(n)** son:

- **Admisible:** es una heurística optimista, es decir, aquella que regresa un costo estimado menor que el que realmente es.

- **Consistente:** si para cada nodo **n** y para cada nodo vecino **v** el costo estimado para alcanzar la meta desde **n** es menor que el costo del paso para ir de **n** a **v** más el estimado para ir de **v** a la **meta**, es decir, si **h(n) < g(n,v) + h(v)**, donde **g(n,v)** es el costo real para ir del nodo **n** al nodo **v** (peso de la arista).

La heurística admisible más simple es pensar que en todo momento ya se llegó a la meta, es decir, **h(n) = 0** para todos los nodos. Esta heurística cumple con la condición de que siempre es menor que lo que falta realmente. De hecho, si en **A*** la heurística **h(n) = 0** entonces **f(n) = c(n)** y se convierte en **UCS**, el cual ya se sabe que es óptimo, esto es, que logra encontrar el camino más corto.

En muchos de los problemas reales con que la heurística cumpla la condición de ser admisible es suficiente para encontrar una buena solución suficientemente cercana a la óptima.

Como ya se comentó, las heurísticas se crean de forma muy especial para un problema. Una forma de garantizar que la heurística es admisible es usar como función heurística la solución para el **problema relajado**, esto es, para el problema original al que se le quitan una o más restricciones.

La complejidad de A* depende completamente de la heurística que se utilice, por lo que no tiene sentido hablar de la complejidad de A* como algoritmo.

*7.3.1 Una aplicación de A**

Si usamos **A*** para resolver el problema de encontrar la distancia más corta de un nodo a otro en un grafo que representa ciudades, una heurística que cumple con ser admisible y consistente es la **distancia en línea recta** de una ciudad a otra. La distancia recta siempre es menor que la distancia real y cumple con la desigualdad necesaria para ser consistente.

Apliquemos **A*** al grafo de la Figura 7.6 para encontrar el camino con la distancia más corta para ir del nodo **S** al nodo **G**. En la figura se muestra con **h(n) = H**, el valor de la heurística en cada nodo **n** que representa la distancia en línea recta de cada nodo **n** al nodo meta **G**. Se debe recordar que **A*** usa **f(n) = c(n) + h(n)** para evaluar un nodo **n**, donde **c(n)** es el costo para ir de la raíz (nodo inicial) al nodo **n**, el cual se calcula como en UCS y **h(n)** es la heurística que estima la distancia que le falta para llegar de **n** al nodo meta.

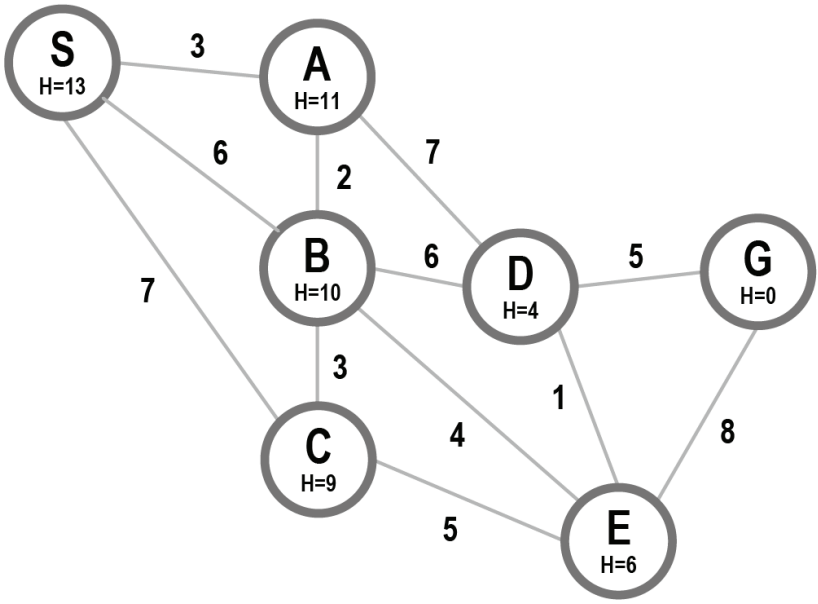


Figura 7.6 Grafo pesado con función heurística h(n) = H

El árbol de búsqueda completo, formado con el algoritmo A* para ir del nodo inicial S al nodo final G se muestra en la Figura 7.7.

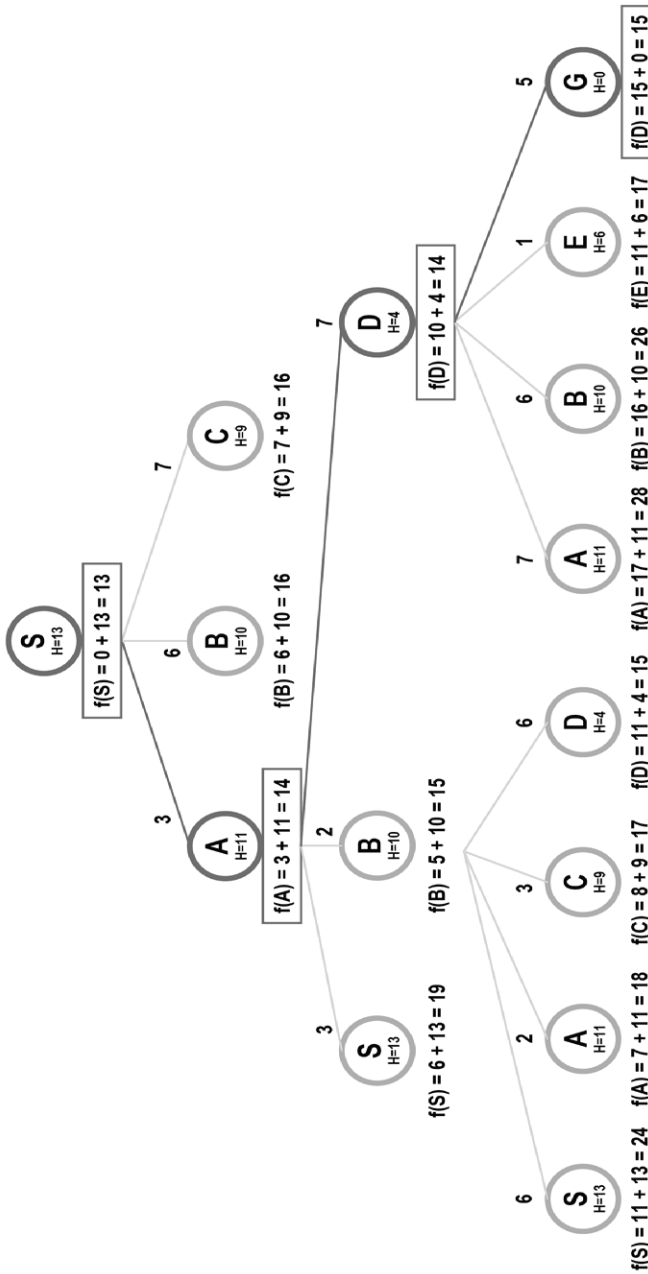


Figura 7.7 Árbol de búsqueda completo para ir del nodo S al nodo G. El camino encontrado se marca en rojo y es de costo mínimo (el camino más corto)

Aplicando el Algoritmo 7.1 con una cola con prioridad como estructurase obtienen los pasos para llegar a la solución con A*:

1. Para el nodo inicial **S**, calcular la función **f(S) = c(S) + h(S) = 0 + 13 = 13** y meterlo a la estructura.

2. Sacar un elemento de la estructura y hacerlo el nodo actual (el único es el nodo **S**).

3. Como el nodo actual no es el nodo meta **G**, **S** se marca como visitado, en este caso se mete a un conjunto de nodos **visitados = {S}** y se expande metiendo a sus hijos a la estructura.

4. Sus hijos son: **A** con **f(A) = c(A) + h(A) = 3 + 11 = 14, B** con **f(B) = c(B) + h(B) = 6 + 10 = 16** y **C** con **f(C) = c(C) + h(C) = 7 + 9 = 16.**

5. Sacar un nodo de la estructura y hacerlo el nodo actual. En este caso, el menor es el nodo A con **f(A) = 14** y como no ha sido visitado (no está en el conjunto **visitados = {S}**) se explora.

6. Como el nodo actual no es el nodo meta se marca como visitado haciendo **visitados = {S, A}** y se expande metiendo sus hijos a la estructura.

7. Sus hijos son: **S** con **f(S) = 6 + 13 = 19, B** con **f(B) = 5 + 10 = 15** y **D** con **f(D) = 10 + 4 = 14**.

8. Sacar un nodo de la estructura y hacerlo el nodo actual. En este caso el menor dentro de la estructura es **D** con **f(D) = 14** y como no se ha visitado, se explora.

9. Como no es el nodo meta se marca como visitado haciendo **visitados = {S, A, D}** y se expande.

10. Sus hijos son: **A** con **f(A) = 17 + 11 = 28, B** con **f(B) = 16 + 10 = 26, E** con **f(E) = 11 + 6 = 17** y **G** con **f(G) = 15 + 0 = 15**. Observa que, aunque ya se generó **G**, no se ha explorado porque sigue en la estructura por lo que el algoritmo continúa.

11. Sacar un nodo de la estructura y hacerlo el nodo actual. En este caso el de menor costo es **15**, pero hay dos con valor **15**: **B** y **G**, sin embargo, como es una cola con prioridad sale primero el que entró primero, es decir el **B** del tercer nivel del árbol.

12. Como el nodo actual no es el nodo meta se marca como visitado haciendo **visitados = {S, A, D, B}** y se expande.

13. Sus hijos son: **S** con **f(S) = 11 + 13 = 24, A** con **f(A) = 7 + 11 = 18, C** con **f(C) = 8 + 9 = 17** y **D** con **f(D) = 11 + 4 = 15**. Observa que, aunque **D** ya se visitó no se elimina porque no ha salido de la estructura.

14. Sacar un nodo de la estructura y hacerlo el nodo actual. En este caso el de menor costo es **15** y nuevamente hay dos con ese valor: **G** y **D**, sin embargo **G** entró primero y sale primero.

15. Como el nodo actual **G** es el nodo meta sí es el nodo meta, el proceso se termina y se obtiene el camino para ir de **S** a **G** que es: **S – A – D – G** con un costo de **15** y está marcado en rojo en la figura 7.6.

*7.3.2 Implementación de A**

A continuación se muestra una posible implementación de A* para resolver el problema de la distancia más corta entre dos nodos en un grafo. Se utilizará un grafo bidireccional y se recibe por medio de un archivo en forma de lista de nodos, cuyo primer renglón contiene el número **n** de nodos y el número **m** de aristas del grafo. Los siguientes n renglones contienen los valores de **h(n)** para cada nodo (normalmente la heurística se calcula con una función, pero por simplicidad del ejemplo se proporcionará como parte de los datos del grafo) y los siguientes **m** renglones contienen las aristas con tres números, los dos primeros son los nodos de la arista (bidireccional) y el tercero es el peso de la arista (distancia). Por último, los nodos se toman como números para facilitar la indexación, pero se podría hacer con caracteres con algunas pequeñas modificaciones. El problema por tratar es el mismo del punto anterior, pero ahora con números en los nodos como se muestra en la Figura 7.8.

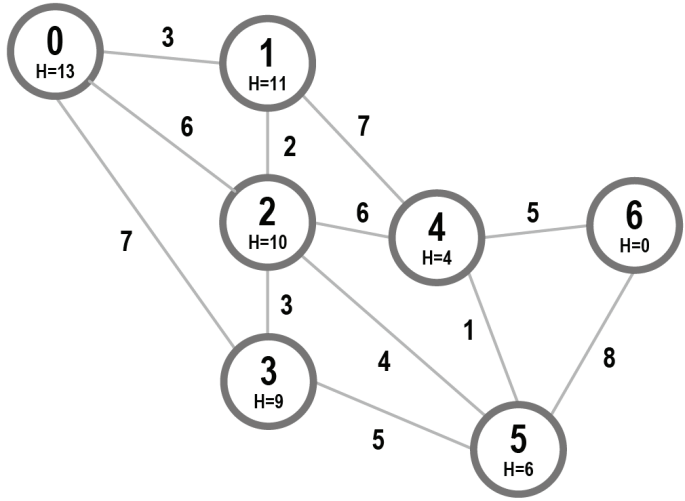


Figura 7.8 Grafo para el ejemplo de aplicación de A*

```cpp
#include <iostream> // IO
#include <vector>   // vector
#include <queue>    // priority_queue
#include <utility>  // pair
#include <fstream>  // file

using namespace std;

struct nodo{
  int H; // es la función heurística
  bool visitado;
};

struct nodoBusqueda{
  int i; // nombre del nodo
  int c; // costo desde la raíz hasta el nodo n
  int f; // función de evaluación f(n) = c(n) + h(n)
  string camino; // se va guardando el camino
};

struct comp{ // para un min-Heap
  bool operator()(nodoBusqueda a, nodoBusqueda b){
    return a.f > b.f; // regresa true si f(a) > f(b)
  }
};

// definimos una cola con prioridad como estructura
// pero la usamos para encontrar los mínimos, es decir, un min-Heap
priority_queue<nodoBusqueda, vector<nodoBusqueda>, comp> estructura;
vector<vector<pair<int,int> > > adj; // <otro nodo, peso>
vector<nodo> nodos; // lista de nodos visitados con H


void leeArchivo(string archivo){
  int n, m;
  ifstream miArchivo(archivo);

  if (miArchivo.is_open()){
    miArchivo >> n >> m;
    adj.resize(n);
    nodos.resize(n);
    for (int i=0; i<n; i++){
      miArchivo >> nodos[i].H;
      nodos[i].visitado = false;
    }
    for (int i=0; i<m; i++){
      int x, y, w;
      pair<int,int> n;
      miArchivo >> x >> y >> w;
      n.first = y; n.second = w;
      adj[x].push_back(n);
      n.first = x;
```

```
53        adj[y].push_back(n);
54      }
55      miArchivo.close();
56    } else
57        cout << "No se pudo abrir el archivo\n";
58 }
59
60 nodoBusqueda creaNodoBusqueda(int nodo, int c, string camino){
61   nodoBusqueda actual;
62   actual.i = nodo;
63   actual.c = c;
64   actual.f = actual.c + nodos[actual.i].H;
65   actual.camino = camino + to_string(nodo) + " ";
66   return actual;
67 }
68
69 pair<int,string> busquedaAStar(int inicial, int final){
70   nodoBusqueda actual;
71   estructura.push(creaNodoBusqueda(inicial, 0, "")); // nodo inicial
72
73   while (!estructura.empty()){
74     actual = estructura.top();
75     estructura.pop();
76     if (!nodos[actual.i].visitado)
77       if (actual.i == final){
78         pair<int, string> solucion(actual.f, actual.camino);
79         return solucion;
80       }
81       else{ // expande el nodo actual
82         nodos[actual.i].visitado = true;
83         for (pair<int,int> n:adj[actual.i]) // genera los vecinos
84           estructura.push(creaNodoBusqueda(n.first,
85                           actual.c + n.second, actual.camino));
86       }
87   }
88   pair<int, string> solucion(-1, "");
89   return solucion;
90 }
91
92 int main(){
93   leeArchivo("grafoAStar.txt");
94   pair<int, string> solucion = busquedaAStar(0, 6);
95   cout << "Costo: " << solucion.first << endl;
96   cout << "Camino: " << solucion.second;
97 }
```

La estructura **nodo** (línea 9) guarda un nodo con su función heurística H y nos dice si ya fue visitado o no y sirve para crear una lista de nodos visitados (línea 31). La estructura **nodoBusqueda** (línea 14) se utiliza para crear un nodo que se coloca en la estructura. Este nodo guarda su nombre **i** (número de nodo para

este ejemplo), costo **c**, su función **f** y el **camino** que se ha recorrido desde la raíz (nodo inicial) hasta el nodo en cuestión. Esta estructura se usa para formar los elementos que se colocarán en el árbol de búsqueda.

Como se desea encontrar la distancia mínima de un nodo a otro, la **estructura** usada para colocar los nodos generados en el árbol de búsqueda debe ser una cola con prioridad para los mínimos, la cual se logra con un min-Heap (línea 29). Como lo que se compara en el min-Heap son nodos, y estos son estructuras, es necesario indicarle cómo se va a realizar la comparación. Esto se hace con la estructura **comp** (línea 21) donde se indica que la comparación solo se hace con el valor de la función f (línea 23).

La función **leeArchivo** (línea 34) solo lee el archivo que contiene el grafo, cuyo nombre recibe como parámetro. El archivo tiene la estructura que se explicó anteriormente y se encarga de colocar el grafo en la lista de adyacencia (líneas 46 a 54) previamente definida como variable global (línea 30). También llena la función heurística en el arreglo de nodos y los inicializa como no visitados (líneas 42 a 45).

La función **creaNodoBusqueda** (línea 60) recibe la información necesaria para crear un nodo del árbol de búsqueda, el cual se colocará en la estructura (un vecino), lo crea y lo regresa. El costo lo crea sumando al costo de su papá, que se trata del peso de la arista que los conecta (línea 64). El camino lo hace de una forma similar al camino de su papá que le pega el nodo actual (línea 65).

Finalmente, la función **busquedaAStar** (línea 69) recibe el nodo inicial y el final para el que se quiere encontrar el camino e implementa exactamente el Algoritmo 7.1 para lograrlo, usando a la estructura como una cola con prioridad (min-Heap).

Se modifica un poco la salida para regresar un par, cuyo primer elemento es el costo del camino y el segundo es el camino mismo. Cuando logra encontrar el camino regresa los valores indicados (línea 79), cuando no encuentra el camino regresa un costo de -1 y un camino vacío (línea 88).

Para correrlo simplemente se llama a la lectura del archivo (línea 93) y luego al método A* (línea 94), imprimiendo los resultados encontrados (líneas 95 y 96).

*7.3.3 El algoritmo IDA**

El algoritmo A* tiene el mismo inconveniente de DFS, esto significa que se puede ir a explorar algo sumamente profundo cuando en la rama de al lado se podría encontrar la solución más rápidamente, es decir, a una profundidad menor. Este problema en DFS se puede minimizar usando un algoritmo con profundidad iterativa, esto es, a la profundidad se pone un límite **L** , el cual inicia en 1, para la exploración y si en ese límite no se encuentra la solución, se aumenta **L** en 1 y se vuelve a empezar. Aunque esto es muy tardado permite encontrar con DFS las soluciones menos profundas. A esta variante de DFS se le conoce como DFS con profundidad iterativa.

En A* se puede hacer exactamente lo mismo, creando así la variante propuesta por Richard Korf (1985): A* con profundidad iterativa (IDA* por sus siglas en inglés, *Iterative deepening A**), que se presenta en el Algoritmo 7.2.

Algoritmo 7.2 A* con profundidad iterativa (IDA*)

Entrada: el grafo donde se va a hacer la búsqueda y el valor que se quiere buscar o sus características.

Salida: el apuntador al nodo que contiene el valor buscado o **nulo** en caso de que no lo encuentre.

1. Hacer **L = 1**

2. Si **L** llegó al final regresar nulo (no se encontró la solución), ir a FIN

3. Hacer A* hasta el límite de profundidad L.

4. Si A* tuvo éxito, regresar la solución (se encontró la solución), ir a FIN

5. Si no, hacer L = L + 1, ir a 2

6. FIN

7.4 Búsqueda de escalada

Las búsquedas descritas en la sección 7.3 tienen el inconveniente de requerir mucha memoria, ya que se necesita ir guardando todo el árbol (como en BFS) o una parte de este (una rama completa en DFS) así como los nodos que ya han sido visitados. Cuando el grafo es muy grande estos algoritmos se deben aplicar con límites, de otra forma la memoria no alcanzaría.

Existe un grupo de algoritmos, llamados de **búsqueda local**, que intentan resolver precisamente el problema de los grandes requerimientos de memoria. Estos algoritmos sacrifican la exploración sistemática de todos los nodos por la disminución en

la memoria. Lo logran no guardando todos los nodos o todo el camino, por lo que no siempre llegan a la mejor solución posible (no son óptimos), ni tampoco funcionan cuando se requiere recordar el camino que se siguió para llegar a la solución, sin embargo, han logrado obtener buenos resultados para problemas donde no se conoce el nodo que se está buscando, solo se conocen las características que debe cumplir, por ejemplo: ser el máximo de una función.

El algoritmo de búsqueda local básico es el llamado **búsqueda de escalada** o **ascensión de colinas** (*Hill Climbing*, en inglés). Este algoritmo intenta encontrar, lo más pronto posible, el punto máximo de una función (de ahí el nombre de ascensión de colinas) y por eso a este tipo de algoritmos también se les conoce como **algoritmos de optimización**, ya que tratan de encontrar el máximo de una función (optimización).

Ascensión de Colinas también utiliza una función heurística **h(n)** para evaluar un nodo **n**, cuyo valor dice qué tan "bueno" es el nodo evaluado. A esta función se le conoce como **función objetivo** de la misma forma que en un problema de optimización porque es la misma función que se quiere maximizar. De la misma manera que la búsqueda heurística, el diseño de **h(n)** no es trivial y depende completamente del problema que se está resolviendo por lo que la función resulta ser muy específica para cada problema. La condición que debe cumplir la función heurística es que su punto máximo se logre en el nodo que tiene la solución buscada.

Como Ascensión de Colinas siempre quiere ir hacia arriba, entre más alto sea el valor de **h(n)** mejor será el nodo **n**, aunque como cualquier algoritmo de maximización, se puede modificar fácilmente para hacer de minimización. En cada paso del algoritmo solo se evalúan sus nodos vecinos (de ahí el nombre de

búsqueda local). Si la evaluación del **mejor vecino** encontrado es mayor que el nodo actual, el nodo actual se mueve hacia ese vecino y el proceso de búsqueda continúa. Si ninguno de los vecinos es mayor que el actual, entonces se decreta convergencia y el nodo actual se regresa como la mejor solución encontrada. Se puede observar que, debido a que siempre se selecciona el "mayor" vecino, el algoritmo de búsqueda de escalada es un algoritmo avaro (*greedy*) que tiene los mismos problemas que cualquier algoritmo avaro, es decir, cada vez que se corre, lo único que puede garantizar es llegar a un **óptimo local**, el cual no necesariamente es la solución encontrada. El algoritmo se presenta a continuación:

Algoritmo 7.3 Búsqueda de escalada

Entrada: el grafo donde se va a hacer la búsqueda.

Salida: el apuntador al nodo que contiene un máximo local.

1. Hacer **nodo actual = nodo inicial**

2. Evaluar el **nodo actual**

3. Expandir el **nodo actual** para generar todos sus vecinos

4. Evaluar cada uno de los vecinos con **h(n)**

5. Seleccionar el **mejor vecino** (el vecino con la mejor evaluación)

6. Si el **nodo actual** tiene una mejor evaluación que la del **mejor vecino**, el nodo actual es la

7. solución, ir a fin

8. Hacer el **nodo actual = mejor vecino**, ir a 2

FIN

Los algoritmos locales solo buscan en una vecindad del nodo actual y no recuerdan el camino, por eso se dice que es como si se quisiera escalar lo más rápido posible el Pico de Orizaba con neblina y con amnesia. Lo único que nos queda es ir hacia donde la pendiente sea mayor, esperando que nos lleve hacia el máximo absoluto, aunque muchas veces podemos llegar a un máximo local como lo muestra la Figura 7.9.

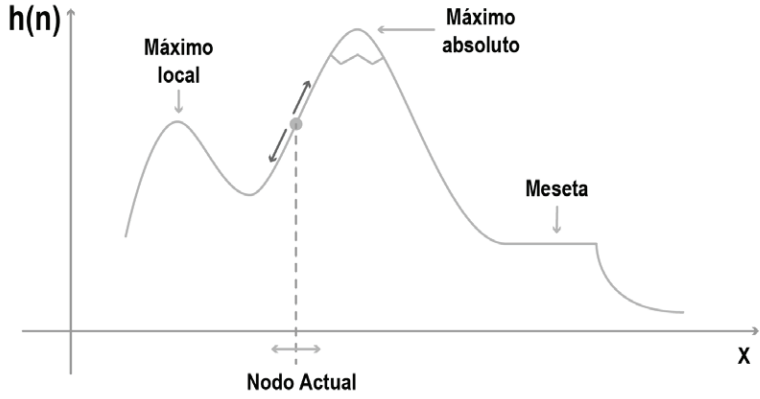


Figura 7.9 Ejemplo de una colina como el Pico de Orizaba

Para tratar de escapar de los óptimos locales se han diseñado varios mecanismos que han dado lugar a nuevos algoritmos al modificar ligeramente el algoritmo original. La forma más común es correr el algoritmo original varias veces partiendo siempre de un nodo inicial aleatoriamente seleccionado y quedarse con el que regresa la mejor solución. A esto se le conoce como **ascensión de colinas con reinicio aleatorio**, sin embargo esto no garantiza obtener siempre la solución óptima, pero entre más veces se corra, mayor será la probabilidad de obtenerla.

Lo que hace avaro a un algoritmo es que en cada paso siempre selecciona el mejor vecino, así que otro mecanismo desarrollado para tratar de salir de los óptimos locales es quitarle un

poco la "avaricia", esto es, no seleccionar siempre el mejor de todos. Esto ha dado lugar a dos algoritmos:

- **Ascensión de colinas aleatorio:** en el que se selecciona un vecino en forma aleatoria, es decir, de entre todos los que son mejores que el nodo actual.

- **Ascensión de colinas de primera opción:** en el que se selecciona un vecino en forma aleatoria. Si el vecino es mejor que el actual, se hace actual = vecino y el proceso continúa. Si no es mejor, se selecciona otro en forma aleatoria. Este es muy útil cuando el número de vecinos es muy grande.

Un algoritmo de búsqueda local no garantiza llegar al valor óptimo, además de que su comportamiento depende mucho de la heurística utilizada. Por esta razón, no se puede hablar de la complejidad de un algoritmo como tal.

El mecanismo completo de ascensión de colinas se comprenderá mejor al aplicarlo a la solución de un problema simple.

7.4.1 Aplicación de búsqueda de escalada

Las aplicaciones para el algoritmo de búsqueda de escalada tienen un espacio de búsqueda discreto (como un grafo). Por lo general, no se conoce el nodo final solo se conocen las características que debe tener y, normalmente, se puede iniciar en cualquier nodo del espacio, por lo que se puede seleccionar en forma aleatoria, aunque en algunos problemas se tiene que iniciar en un estado muy específico. Es por eso por lo que no se requiere guardar el camino, ya que con llegar al nodo que cumple con las características del nodo final se ha resuelto el problema.

Los problemas con espacios de búsqueda continuos (como las funciones) se pueden resolver con este método discretizándolos de alguna forma y, por lo general, se logra llegar muy cerca del resultado óptimo. Una forma común de discretizar un espacio continuo es mediante una representación especial de sus posibles soluciones en el espacio de búsqueda, por ejemplo, en el caso de una función **f(x)** que se quiera maximizar, las **x** se pueden representar como números binarios de **n** bits, donde **n** es un número finito. Esta representación deja fuera a muchos puntos, por ejemplo, el 2.1 no se puede representar en 3 bits, dado que los dos primeros son la parte entera y el tercero es la parte decimal, lo más cercano sería 2.0 (10.0 en binario), ya que el siguiente es 2.5 (10.1 en binario), pero los que quedan pueden ser suficientes para que el método se pueda acercar lo más posible al resultado óptimo.

Apliquemos el algoritmo de búsqueda de escalada al problema de encontrar la distancia mínima (no el camino, porque no se guarda, aunque si modificamos un poco el algoritmo lo podríamos obtener, pero perderíamos la ventaja de la poca memoria que requiere) que hay para llegar del nodo **S** al **G** en el grafo de la Figura 7.7.

Posiblemente no se llegue al camino más corto por quedarnos en un mínimo local, lo cual dependerá de cómo definamos la función heurística de evaluación.

Si usamos como función la h(n) el proceso sería el siguiente:

1. Hacer nodo actual = S, cuya evaluación es H=13. El costo es 0.

2. Como S no es G, generamos sus vecinos que son: A con 11, B con 10 y C con 9.

3. El mejor de los vecinos es el que tenga la menor evaluación porque queremos encontrar el camino con la menor distancia. En este caso es C con 9. Como 9 es mejor que 13 del nodo actual nos movemos a él haciendo nodo actual = C. El nuevo costo se obtiene sumando el costo actual más la distancia para ir de S a C, que es 7, lo que da un costo de 7.

4. Como C no es G, generamos sus vecinos que son: S con 13, B con 10 y E con 6.

5. El mejor de los vecinos es E con 6. Como 6 de E es mejor que 9 del nodo actual C, nos movemos a él haciendo nodo actual = E. El nuevo costo se obtiene sumando el costo actual 7 más la distancia para ir de C a E, que es 5, lo que da un costo de 12.

6. Como E no es G, generamos sus vecinos que son: C con 9, B con 10, D con 4 y G con 0.

7. El mejor de los vecinos es G con 0. Como 0 de G es mejor que 6 del nodo actual E, nos movemos a él haciendo nodo actual = G. El nuevo costo se obtiene sumando el costo actual 12 más la distancia para ir de E a G, que es 8, lo que da un costo de 20.

8. Como G es el nodo final, el proceso termina y regresamos un costo de 20.

La Figura 7.10 muestra el árbol de búsqueda para el proceso anterior.

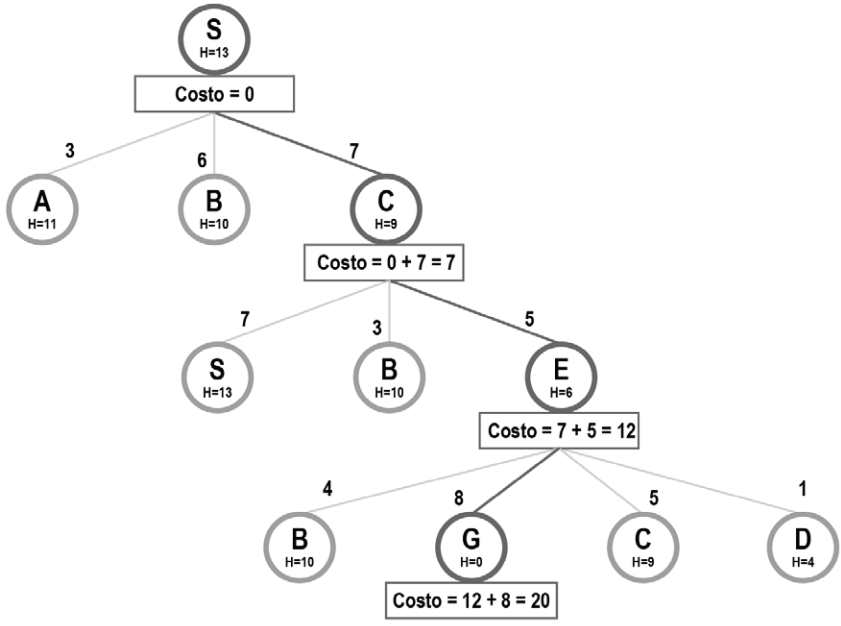


Figura 7.10 Árbol de búsqueda generado al aplicar el algoritmo de búsqueda en escalada al grafo de la Figura 7.7 para ir del nodo S al nodo G

Se puede observar el resultado obtenido con costo 20, que no es el camino más corto, el cual tiene un costo 15, obtenido con A*. Además, no se puede iniciar en un estado en forma aleatoria porque tenemos bien definido el estado inicial, por lo que siempre que se corra se obtendrá el mismo resultado. Sin embargo, esta aplicación muestra de una forma clara y simple la forma en la que se aplica búsqueda en escalada.

Hay algunos problemas clásicos que se resuelven bastante bien con este algoritmo, como es el *problema de las 8 reinas* del ajedrez, en el que se busca colocar 8 reinas en un tablero de ajedrez (de 8 X 8 casillas) sin que ninguna se ataque con otra.

7.4.2 Implementación de búsqueda de escalada

Una posible implementación para el problema del punto anterior es la siguiente:

```
#include <iostream> // IO
#include <vector>   // vector
#include <fstream>  // file

using namespace std;

vector<vector<pair<int,int> > > adj; // <otro nodo, peso>
vector<int> H; // heurística en cada nodo

void leeArchivo(string archivo){
  int n, m;
  ifstream miArchivo(archivo);

  if (miArchivo.is_open()){
    miArchivo >> n >> m;
    adj.resize(n);
    H.resize(n);
    for (int i=0; i<n; i++)
      miArchivo >> H[i];
    for (int i=0; i<m; i++){
      int x, y, w;
      pair<int,int> n;
      miArchivo >> x >> y >> w;
      n.first = y; n.second = w;
      adj[x].push_back(n);
      n.first = x;
      adj[y].push_back(n);
    }
    miArchivo.close();
  } else
      cout << "No se pudo abrir el archivo\n";
}

int hillClimbing(int inicial, int final){
  int actual = inicial;
  int costo = 0;

  while (true){
    if (actual == final)
      return costo; // solución
    else{ // expandirlo
      int mejor = adj[actual][0].first;
      int d = adj[actual][0].second;
      for (pair<int,int> n:adj[actual])
```

```
45          if (H[n.first] < H[mejor]){
46            mejor = n.first;
47            d = n.second;
48          }
49        if (H[mejor] < H[actual]){
50          actual = mejor;
51          costo = costo + d;
52        } else
53          return costo; // óptimo local
54      }
55    }
56    return costo; // sólo por seguridad
57 }
58
59 int main(){
60    leeArchivo("grafoAStar.txt");
61    cout << "Costo: " << hillClimbing(0,6) << endl;
62 }
```

La función **leeArchivo** (línea 10) lee los datos del grafo de un archivo que recibe como parámetro y lo pasa tanto a lista de adyacencia definida en la línea 7, como a la función heurística, en este caso colocada como un arreglo y definida en la línea 8.

La función **hillClimbing** (línea 34) recibe como parámetros tanto el nodo inicial como el final del camino buscado y aplica **búsqueda de escalada** para encontrar el camino de menor costo. Si el algoritmo llega al estado final regresa la solución encontrada (línea 40), si no, selecciona el mejor vecino (líneas 44 a 48) y si éste es mejor que el actual (línea 49), lo hace el actual (línea 50), actualiza el costo (línea 51) y continúa el proceso. Si el mejor vecino no es mejor que el actual (línea 53) significa que llegó a un óptimo local y no encontró el camino solicitado, por lo que regresa el costo encontrado hasta ese momento, el cual para este problema no sirve de mucho.

Para correrlo se debe leer el archivo con los datos del grafo (línea 60) y luego se llama a la función **hillClimbing** con el nodo inicial y el final del camino buscado (línea 61) que regresará el costo encontrado.

7.5 Recocido simulado

Desde hace varios siglos, los forjadores de espadas se dieron cuenta que si el metal de la espada se calentaba mucho y luego se dejaba enfriar muy lentamente la espada resultaba ser muy resistente. A este proceso se le conoce como **recocido**. Fue hasta finales del siglo XIX que, con el nacimiento de la Mecánica Estadística, el científico Ludwig Boltzmann logró explicar el funcionamiento del metal en este proceso. La idea fundamental es que los cristales que forman el metal al pasar por el proceso de recocido se logran acomodar en forma óptima y esto le da la máxima dureza a la espada. Se trata de un proceso físico que logra encontrar el máximo (máxima dureza) en un material, en otras palabras, es un proceso de optimización.

La explicación física queda fuera del alcance de este libro, pero se puede comentar que se utiliza la Distribución de Probabilidad de Boltzmann que proporciona la probabilidad P de que un sistema esté en un cierto estado **i** en función de la energía del estado $\mathbf{E_i}$, la temperatura **T** del sistema y la constante de Boltzmann **k**:

$$P(i) = \frac{1}{S} e^{-\frac{E_i}{kT}} \qquad (7.1)$$

S es simplemente una constante de normalización para que dé una probabilidad, es decir, un valor entre 0 y 1.

Considerando que el proceso de recocido da como resultado un arreglo óptimo de los cristales de un material, los científicos de la computación lo toman y lo usan para crear un algoritmo que logra replicar este proceso en la computadora para encontrar el valor óptimo (máximo o mínimo) de una función. Como

el recocido no se hace físicamente, sino que solamente se simula en la computadora, a este nuevo algoritmo se le llamó **recocido simulado** (*Simulated Annealing*, en inglés).

Recocido simulado es un algoritmo de búsqueda local por lo que es muy similar a Ascensión de Colinas descrito en el algoritmo 7.2 con una variación para quitarle la avaricia similar a la de Ascensión de Colinas de Primera Opción, la cual se puede explicar en tres puntos:

- Se genera un vecino en forma aleatoria.
- Si el vecino es mejor que el actual, se toma y se hace actual = vecino.
- Si el vecino es peor, se toma con una probabilidad que disminuye exponencialmente con qué tanto empeora la solución del actual $\Delta E=h(vecino)-h(actual)$ y también disminuye conforme la temperatura T disminuye.

En otras palabras, si el vecino tiene una evaluación peor que la del actual, entre más empeore la solución, es decir, entre más malo sea, menor será la probabilidad de ser seleccionado para continuar el proceso. También, entre más frío esté el material, es decir, entre menor sea la temperatura, menor será la probabilidad de ser seleccionado para continuar el proceso. Esta probabilidad es exactamente la que se obtiene con la distribución de Boltzmann descrita en (7.1).

Si la temperatura se disminuyera muy, pero muy lentamente, en pasos iguales a diferenciales, el algoritmo garantizaría encontrar el valor óptimo de la función, pero tardaría un tiempo infinito. Si no se dan pasos tan pequeños en la disminución de la temperatura se puede obtener una solución cercana a la óptima en un tiempo razonable.

El algoritmo completo se describe en el Algoritmo 7.4, en el cual la temperatura T se va disminuyendo de acuerdo a una función **t(i)** discreta (no puede ser continua porque es simulada en la computadora) donde **i** es el tiempo (paso) del algoritmo, es decir, **t(0)** es la temperatura inicial y es la máxima; y a medida que **i** tiende a infinito, **t(i)** tiende a 0. La función **t(i)** debe ser diseñada para que tenga el comportamiento adecuado.

Algoritmo 7.4 Recocido simulado

Entrada: el grafo donde se va a hacer la búsqueda y la función t(i).

Salida: el apuntador al nodo que contiene una solución.

1. Hacer **nodo actual = nodo inicial**
2. Para **i = 0** hasta ∞:
3. Calcular la temperatura actual con **T = t(i)**
4. Si **T = 0** (se considera completamente frío el material), ir a 12
5. Evaluar el **nodo actual**
6. Seleccionar un **vecino** en forma aleatoria
7. Evaluar el **vecino** con **h(n)**
8. Hacer $\Delta E = h(vecino) - h(actual)$
9. Si $\Delta E > 0$, hacer **actual = vecino**
10. Si no, hacer **actual = vecino** con una probabilidad $e^{\Delta E/T}$
11. Regresar el actual como resultado
12. FIN

Se puede observar que en el paso 11 se usa la distribución de Boltzmann (7.1), pero sin el signo menos, el cual no le hace falta porque solo llegaría al punto 12 si ΔE es negativa, por lo que ahí va el signo. Se puede observar que, la función e^{-x} decrece a medida que x se hace más grande (porque es negativa). En la distribución usada en el paso 11 hay dos formas de hacer más grande el exponente:

- Aumentando el numerador ΔE o (el nodo vecino es muy malo).
- Disminuyendo el denominador T (el material está muy frío).

Lo anterior coincide completamente con el comportamiento deseado.

7.5.1 Aplicación de recocido simulado

Las aplicaciones en las que se puede utilizar **recocido simulado** son muy amplias y variadas. Las condiciones en las que se puede aplicar son las mismas que las de todos los algoritmos de búsqueda local, esto es, es un espacio de búsqueda discreto y se requiere una función de evaluación. También es posible aplicar este algoritmo a problemas con espacios continuos, siempre y cuando podamos discretizarlos de alguna manera sabiendo que la solución va a ser muy aproximada, pero no necesariamente va a ser la solución óptima debido, entre otras cosas, debido a la discretización realizada.

Por otro lado, es complicado seguir un ejemplo completo a mano, como lo hemos hecho en secciones anteriores, debido a que algunos pasos se realizan en forma aleatoria, como la selección del vecino o la aceptación de este aun cuando no mejore la

solución actual. El paso aleatorio ocasiona que las selecciones no sean las mismas para todos complicando el seguimiento manual. Sin embargo, se pueden hacer algunos pasos para entender el mecanismo suponiendo ciertos números aleatorios.

A continuación, se presenta la aplicación de **recocido simulado** para la solución del siguiente problema de optimización muy simple. Se trata de encontrar el valor de x que maximiza la función $f(x) = x^2$. Es claro que la solución es que x debe tener un valor infinito, ya que entre más crezca **x** mayor será la función **f(x)**. Para poderlo usar como demostración vamos a restringir el problema de la siguiente forma:

$$\begin{aligned} &\text{Maximizar } f(x) = x^2 \\ &\text{Sujeto a:} \\ &\qquad x \text{ es un entero} \\ &\qquad 0 \leq x \leq 31 \end{aligned} \tag{7.2}$$

Desde luego que la solución de este problema es **x = 31**. La razón de la restricción en el valor que puede tomar **x** es que estos posibles valores de **x** los podamos codificar en binario en 5 bits y, de esta forma, discretizar el espacio de búsqueda. Los nodos serían números binarios de 5 bits. Si consideramos que los vecinos se forman cambiando un bit de valor, esto es, si es un 0 pasa a 1 y si es un 1 pasa a un 0 podemos formar el grafo de búsqueda y podemos resolverlo usando **recocido simulado**.

Por ejemplo, si nos encontramos en el nodo 00000, este tendría 5 vecinos: 10000, 01000, 00100, 00010 y 00001. Desde luego que se pueden idear otras formas de formar los vecinos y algunos son mejores que otras. Para este ejemplo vamos a utilizar la que se explicó. Veamos algunos pasos, suponiendo las selecciones aleatorias que se realizan.

Partamos del nodo 00000 y tratemos de encontrar el valor que maximiza a la función $f(x) = x^2$ con las restricciones establecidas en (7.2) usando **recocido simulado**. La función de evaluación puede ser la misma función que se quiere maximizar. Para usarla tomamos el valor del nodo que es el valor de x, pero en decimal y listo. Como función de temperatura vamos a usar una línea recta con una baja pendiente negativa para que la temperatura vaya disminuyendo muy lentamente a medida que pasa el tiempo. Vamos a suponer que la temperatura inicial la colocamos en 1000 grados y queremos llegar a 0 en un tiempo largo, digamos 10000 pasos. Entonces, la función de temperatura sería **T = -0.1t+1000**, donde **t** el es tiempo, también discretizado de tal forma que una unidad es un paso en el algoritmo. La Figura 7.11 nos ayudará a comprender la función seleccionada.

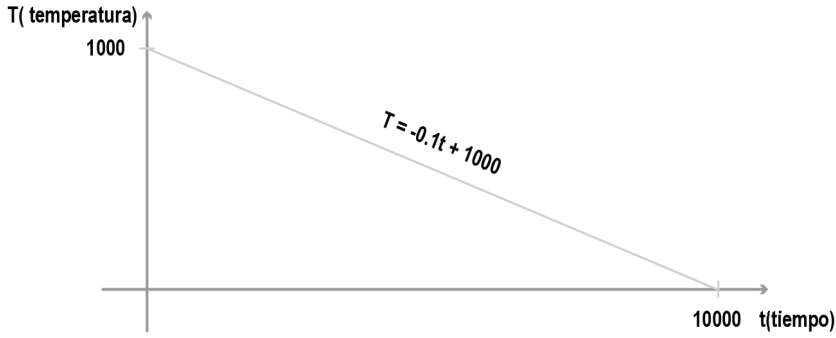


Figura 7.11 Gráfica de la ecuación usada para calcular la temperatura

1. Hacer la temperatura T = 1000 y el tiempo t = 0.

2. Hacer nodo actual = 00000, que es un 0 en decimal y su evaluación sería f(0) = 0.

3. Generamos aleatoriamente uno de sus vecinos, supongamos que fue 00100, que es un 4 en decimal y su evaluación sería f(4) = 16.

4. Como es mejor que el actual, lo tomamos haciendo actual = 00010 y hacemos t = 1.

5. Generamos aleatoriamente uno de sus vecinos, supongamos que fue 00110, que es un 6 en decimal y su evaluación sería f(6) = 36.

6. Como es mejor que el actual, lo tomamos haciendo actual = 00110 y hacemos t = 2.

7. Generamos aleatoriamente uno de sus vecinos, supongamos que fue 10110, que es un 22 en decimal y su evaluación sería f(6) = 484.

8. Como es mejor que el actual, lo tomamos haciendo actual = 10110 y hacemos t = 3.

9. Generamos aleatoriamente uno de sus vecinos, supongamos que fue 10100, que es un 20 en decimal y su evaluación sería f(6) = 400.

10. Este vecino no mejor que actual y empeora el valor con $\Delta E=20-22= -2$. Entonces, debemos decidir si tomarlo o no con una probabilidad $e^{\Delta E/T}$, donde T = -10(3)+1000 = 970, esto es, con una probabilidad $e^{-2/970}$=0.9979. Lo que hacemos para saber si la aceptamos o no es generar un número aleatorio entre 0 y 1. Si este número es menor que 0.9979, lo aceptamos, si es mayor, no lo aceptamos. Si lo aceptamos, modificamos el actual y continuamos con el proceso, si no lo aceptamos, no modificamos el actual y continuamos con el proceso. Se puede observar que la probabilidad de aceptarlo es muy grande, casi 1, aunque sea peor eso es precisamente lo que le quita la voracidad al algoritmo, que valores peores pueden ser seleccionados. Supongamos que el número aleatorio que generamos dio 0.2323, como es menor que 0.9979, se acepta. Hacemos actual = 10100.

11. Así continuamos con el proceso, hasta que la temperatura T=0, momento en el cual regresamos actual como la mejor solución encontrada.

El algoritmo es lento y aunque en este caso, como es un problema muy simple, en unos cuantos pasos el algoritmo logra encontrar el valor más grande, no se tiene forma de saber que es el valor más grande y tiene que continuar hasta que la T=0, pero logra llegar al máximo exacto. Vamos a su implementación.

7.5.2 Implementación de recocido simulado

Una posible implementación para resolver este problema de optimización con **recocido simulado** es la siguiente:

```
#include <iostream>
#include <math.h>
#include <vector>

using namespace std;

// calcula la temperatura dato el tiempo t
int temperatura(int t){
  return -0.1*t + 1000;
}

// genera un número aleatorio entre 0 y n-1
int randNum(int n){
  return rand() % n;
}

// cambia el bit en la posición i
vector<int> cambia(int i, vector<int> a){
  vector<int> b = a;
  b[i] = 1 - b[i];
  return b;
}

// función de evaluación
int f(int x){
  return pow(x,2);
}

// obtiene el decimal de un binario
int bintodec(vector<int> a){
```

```
31   int dec = 0;
32   for (int i=0; i<a.size(); i++)
33     dec = dec + a[i]*pow(2,i);
34   return dec;
35 }
36
37 // imprime nodo
38 void imprimeNodo(vector<int> a){
39   for (int i:a)
40     cout << i;
41   cout << "\n";
42 }
43
44 int recocidoSimulado(vector<int> inicial){
45   int t=0; // tiempo
46   int n = inicial.size(); // número de bits
47   int T = temperatura(t); // temperatura
48   vector<int> actual = inicial; // nodo actual
49   int evActual = f(bintodec(actual)); // evaluación del nodo
50   vector<int> vecino; // nodo vecino
51   int evVecino, E; // evaluación del vecino y diferencia E
52   float p, r; // probabilidad p y número aleatorio r
53
54   while (T > 0){
55     //imprimeNodo(actual);
56     vecino = cambia(randNum(n),actual);
57     evVecino = f(bintodec(vecino));
58     E = evVecino - evActual;
59     if (E > 0){
60       actual = vecino;
61       evActual = evVecino;
62     } else{
63       p = exp(E/T);
64       r = rand()/RAND_MAX;
65       if (r < p){
66         actual = vecino;
67         evActual = evVecino;
68       }
69     }
70     t++;
71     T = temperatura(t);
72   }
73   imprimeNodo(actual);
74   return bintodec(actual);
75 }
76
77 int main(){
78    vector<int> inicial{0,0,0,0,0};
79    cout << recocidoSimulado(inicial);
80 }
```

La implementación de **recocido simulado** es muy simple y se concentra en la función **recocidoSimulado** (línea 44) que recibe el nodo inicial e implementa el proceso descrito en el Algoritmo 7.3.

Los nodos para este problema se implementaron como un vector de enteros que contienen solo valores binarios {1, 0}, por ejemplo, en las líneas 48 y 50 donde se definen el vecino y el actual.

El resto de las funciones son auxiliares y se describen a continuación:

- **temperatura** (línea 8): recibe el tiempo en el que se quiere calcular la temperatura y regresa la temperatura de acuerdo con la función seleccionada.

- **randNum** (línea 13): genera un número aleatorio entre 0 y **n** que recibe como parámetro.

- **cambia** (línea 18): recibe una posición **i** dentro del vector **a** y regresa el vector que se forma al cambiar el bit de la posición **i**. Si es 0 lo pone en 1 y viceversa.

- **f** (línea 25): es la función de evaluación, la cual cambia de acuerdo con el problema a resolver. Recibe un posible valor de x obtenido de un nodo y regresa su evaluación de acuerdo con la función codificada.

- **bintodec** (línea 30): recibe un vector en binario y regresa su valor en decimal.

- **imprimeNodo** (línea 38): recibe un vector en binario y lo imprime para poderlo visualizar.

Para correr el proceso se define un nodo inicial como un vector en binario (línea 78) y se manda a llamar a la función **recocidoSimulado** con este vector inicial como parámetro (línea 79). El número de bits con los que se representa el número se obtiene automáticamente con el **size()** del vector inicial (línea 46), lo cual facilita aumentar los bits para probar la implementación para un rango mayor de valores. Por ahora se usan 5 bits y por eso el nodo inicial en la línea 78 se inicializa con el conjunto **{0, 0, 0, 0, 0}** que representa al número 00000. Cabe aclarar que el número se representa al revés para facilitar la conversión en la función **bintodec** (línea 30). Por ejemplo, el número binario 10110 se representa con el vector {0, 1, 1, 0, 1}, esto es, está al revés de como lo leeríamos normalmente así es como la función **imprimeNodo** (línea 38) lo imprime. En otras palabras, si el resultado final se imprime como 10000 significa que realmente es el número 00001, lo que no afecta para nada el proceso debido a que lo más importante es el valor que se encontró para la **x** y es lo que regresa la función.

Si el usuario quiere ir viendo los nodos que se van generando puede quitar el comentario a la línea 55.

La implementación mostrada, iniciando con 00000, encuentra la solución exacta que es 11111 equivalente a 31. Si se llama con un valor inicial de más bits, por ejemplo: 7 bits, es decir, 0000000, también encuentra el valor máximo 1111111 equivalente a 127.

7.6 Ejercicios del capítulo 7

1. Modifica la implementación del algoritmo de máscara de bits para generar todos los subconjuntos de un conjunto determinado.

2. Modifica la implementación del algoritmo de máscara de bits para generar todas las permutaciones de los dígitos de un número.

3. Implementa el algoritmo encontrase en el medio para el problema de la suma de subconjuntos.

4. Modifica la implementación del algoritmo A* para que pueda calcular la distancia entre en línea recta entre dos ciudades para usarla como la heurística en vez de que ya esté dada. Para esto, tendrá que poder dar las coordenadas de cada una de las ciudades en un plano cartesiano.

5. Modifica la implementación del algoritmo de búsqueda de escalada para que encuentre el máximo de la función f(x)=x2, con las restricciones establecidas en 7.2. Recuerda que tiene que discretizar el espacio de búsqueda. Utiliza un enfoque similar al usado en la implementación de **recocido simulado**.

6. Modifica la implementación del algoritmo de **recocido simulado** para que encuentre el máximo de la función $\mathbf{f(x) = x^2}$, pero con valores de punto flotante, dividiendo la representación de x en e bits para la parte entera y d bits para la parte decimal. Hazlo con 5 bits para la parte entera y 3 para la parte decimal.

7. Modifica la implementación del algoritmo de **recocido simulado** para que encuentre el máximo de la función $\mathbf{f(x,y) = x^2 - 2y + 1}$. Ahora, cada nodo deberá incluir las dos variables. Impleméntalo pensando en $\mathbf{k_1}$ bits para el valor de la **x** y $\mathbf{k_2}$ bits para el valor de la **y** de tal forma que el nodo completo está compuesto por $\mathbf{k_1 + k_2}$ bits.

8. Explica cómo podrías usar **recocido simulado** para encontrar el mínimo de una función.

Créditos

CIP TECNOLÓGICO DE MONTERREY

González Guerra, Luis Humberto, autor. | Cueva Hernández, Victor M. de la, autor. | Pérez Murueta, Pedro Óscar, autor.
Algoritmos : análisis, diseño e implementación / Luis Humberto González Guerra, Víctor Manuel de la Cueva Hernández, Pedro Óscar Pérez Murueta
466 p. cm.

LCSH: Computer algorithms. | Algorithms. | Computer science. | Electronic books. | Local: Algoritmos computacionales. | Algoritmos. | Computación. | Libros electrónicos.

LCC QA76.9.A43 DDC 005.1

Editorial Digital del Tecnológico de Monterrey
José Vladimir Burgos Aguilar. Director Nacional de Bilbiotecas.
Alejandra González Barranco. Líder de Editorial Digital.
Elizabeth López Corolla. Coordinadora editorial.
Dennise Alicia Cervantes. Correctora de estilo.

Innovación y diseño para la enseñanza y el aprendizaje.
Noemí Villarreal Rodríguez. Coordinación de proyectos institucionales y empresariales.
Jesús Alejandro Rocha Gámez. Administración de proyecto.
María Isabel Zendejas Morales. Diseño Editorial.
Gustavo Arteaga Mondragón. Diseño Editorial.

Aviso legal

Editorial: Instituto Tecnológico y de Estudios Superiores de Monterrey
Ave. Eugenio Garza Sada 2501 Sur Col. Tecnológico C.P. 64849 | Monterrey, Nuevo León | México.
Algoritmos : análisis, diseño e implementación
ISBN Obra Independiente: 978-607-501-699-3
Primera edición: marzo 2022.

Amazon Media EU S.à.r.l.
Luxemburgo, Luxemburgo
30 de marzo de 2022
100 ejemplares