INTRODUCCIÓN
AL ANÁLISIS Y AL DISEÑO DE
ALGORITMOS

El análisis de la complejidad de algoritmos, así como la discusión de las ventajas/desventajas, que tienen diferentes soluciones a los problemas clásicos, son temas que todo Ingeniero en Computación debe conocer y dominar al finalizar sus estudios.

En este libro se involucra al lector en los conceptos del análisis y diseño de algoritmos, exponiendo cada tema a través de ejemplos prácticos y utilizando un lenguaje sencillo que hace su lectura agradable y rápida. Se explican los conceptos sin un uso excesivo de notación matemática, lo que representa una forma atractiva de envolver e interesar al lector.

CUAJIMALPA

UNIVERSIDAD AUTÓNOMA METROPOLITANA

ISBN: 978-607-28-0225-4

Introducción al Análisis y al Diseño de **Algoritmos**

UNIVERSIDAD AUTÓNOMA METROPOLITANA

CUAJIMALPA

UNIVERSIDAD AUTÓNOMA METROPOLITANA

INTRODUCCIÓN
AL ANÁLISIS Y AL DISEÑO DE
ALGORITMOS

María del Carmen
Gómez Fuentes
Jorge Cervantes Ojeda

U N I V E R S I D A D A U T Ó N O M A

M E T R O P O L I T A N A

Introducción al Análisis y al Diseño de Algoritmos

Dra. María del Carmen Gómez Fuentes

Dr. Jorge Cervantes Ojeda

Junio 2014.

Editores
María del Carmen Gómez Fuentes
Jorge Cervantes Ojeda

Departamento de Matemáticas Aplicadas y Sistemas
División de Ciencias Naturales e Ingeniería
Universidad Autónoma Metropolitana, Unidad Cuajimalpa

Editada por:
UNIVERSIDAD AUTONOMA METROPOLITANA
Prolongación Canal de Miramontes 3855,
Quinto Piso, Col. Ex Hacienda de San Juan de Dios,
Del. Tlalpan, C.P. 14787, México D.F.

Introducción al Análisis y al Diseño de Algoritmos

Primera edición 2014

ISBN: 978-607-28-0225-4

Impreso en México
Impreso por Publidisa Mexicana S. A. de C.V.
Calz. Chabacano No. 69, Planta Alta
Col. Asturias C.P.

Agradecimiento

Agradecemos profundamente a los revisores anónimos de este libro por sus valiosos comentarios y por la encomiable labor de señalar con detalle las modificaciones y correcciones pertinentes. Gracias a su colaboración pudimos mejorar considerablemente la calidad y presentación de este material.

María del Carmen Gómez.

Jorge Cervantes.

Objetivos

Objetivo General:

Adquirir los conocimientos que desarrollen en el alumno la habilidad de analizar la complejidad de un algoritmo y de utilizar técnicas de diseño apropiadas para la construcción de algoritmos eficientes.

Objetivos Específicos:

1. Analizar la complejidad de un algoritmo por medio de las medidas asintóticas.
2. Evaluar la eficiencia y la calidad de los algoritmos.
3. Comprender las técnicas de diseño de algoritmos de utilidad práctica.

Conocimientos previos:

Para comprender este libro, el lector deberá tener conocimientos básicos de cálculo y de programación.

Contenido

Capítulo I Introducción

Un *algoritmo* es un método para resolver un problema, López et al. (2009) definen *algoritmo* como "un conjunto de pasos que, ejecutados de la manera correcta, permiten obtener un resultado (en un tiempo acotado)". Pueden existir varios algoritmos para resolver un mismo problema. Cuando se estudian los algoritmos es importante analizar tanto su diseño como su eficiencia.

Como se ilustra en la *Figura 1.1*, el diseño del algoritmo consiste en encontrar métodos y procedimientos que resuelvan determinado problema.

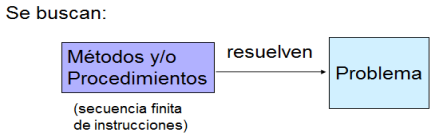


Figura 1.1: Se diseña un *algoritmo* para resolver un problema

Un algoritmo tiene las siguientes propiedades:

1.- Tiene definidas las ***entradas*** que se requieren, así como las ***salidas*** que se deben producir.

2.- Es ***preciso***: Se debe indicar sin ambigüedades el orden exacto de los pasos a seguir y la manera en la que éstos deben realizarse. Por ejemplo, una receta de cocina, aunque tiene todas las demás características de los algoritmos, muchas veces no es precisa: "añada una pizca de sal", "caliente en un recipiente pequeño". Las instrucciones deben ser tan claras que incluso una computadora pueda seguirlas.

3.- Es ***determinista***: para un mismo conjunto de datos proporcionado como entrada, el algoritmo debe producir siempre el mismo resultado.

4.- Es ***finito***: un algoritmo siempre termina después de un número finito de pasos. Si un procedimiento tiene las otras 4 características de un algoritmo, pero no es finito, entonces se le llama "método computacional".

5.- Es ***efectivo***: cada paso o acción se debe poder llevar a cabo en un tiempo finito y se debe lograr el efecto que se desea o espera.

Además, como se ilustra en la *Figura 1.2*, pueden existir diferentes formas de implementar un algoritmo, a cada una de éstas se le llama *implementación del algoritmo* y es un conjunto diferente de instrucciones elaboradas por el programador.

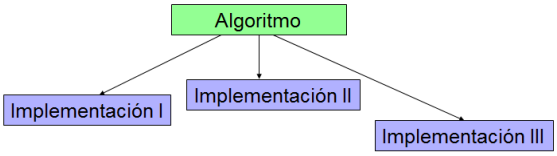


Figura 1.2: Un algoritmo puede tener varias implementaciones

Por otra parte, un problema se puede resolver con diferentes algoritmos, como se ilustra en la *Figura 1.3*.

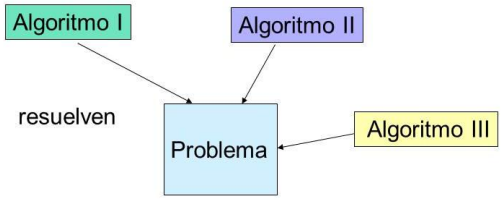


Figura 1.3: Pueden existir varios algoritmos para resolver el mismo problema

"El análisis de algoritmos pretende descubrir si estos son o no eficaces. Establece además una comparación entre los mismos con el fin de saber cuál es el más eficiente, aunque cada uno de los algoritmos de estudio sirva para resolver el mismo problema" [Villalpando, 2003]. Es responsabilidad del programador utilizar los recursos de la computadora de la manera más eficiente que se pueda. Hay diversos métodos y criterios para estudiar la eficiencia de los algoritmos. Por lo general los aspectos a tomar en cuenta para estudiar la *eficiencia* de un algoritmo son el tiempo que se emplea en resolver el problema y la cantidad de recursos de memoria que ocupa. Para saber qué tan eficiente es un algoritmo hacemos las preguntas:

- ¿Cuánto tiempo ocupa?
- ¿Cuánta memoria ocupa?

El tiempo de ejecución de un algoritmo depende de los datos de entrada, de la implementación del programa, del procesador y finalmente de la complejidad del algoritmo. Sin embargo, decimos que "el tiempo que requiere un algoritmo para resolver un problema está en función del tamaño n del conjunto de datos para procesar: $T(n)$" [López et al., 2009].

Según Guerequeta y Vallecillo (2000), hay dos tipos de estudios del tiempo que tarda un algoritmo en resolver un problema:

En los *estudios a priori* se obtiene una *medida teórica*, es decir, una función que acota (por arriba o por abajo) el tiempo de ejecución del algoritmo para unos valores de entrada dados.

En los *estudios a posteriori* se obtiene una *medida real*, es decir, el tiempo de ejecución del algoritmo para unos valores de entrada dados en determinada computadora.

Al tiempo que tarda un algoritmo para resolver un problema se le llama $T(n)$, donde n es el tamaño de los datos de entrada. Es decir, $T(n)$ depende del tamaño de los datos de entrada. El tamaño de la entrada es el número de componentes sobre los que se va a ejecutar el algoritmo. Estos pueden ser por ejemplo: el número de elementos de un arreglo que se va a ordenar, o el tamaño de las matrices que se van a multiplicar.

El tiempo de ejecución $T(n)$ de un algoritmo para una entrada de tamaño n no debe medirse en segundos, milisegundos, etc., porque $T(n)$ debe ser independiente del tipo de computadora en el que corre. En realidad lo que importa es la forma que tiene la función $T(n)$ para saber cómo cambia la medida del tiempo de ejecución en función del tamaño del problema, es decir, con $T(n)$ podemos saber si el tiempo aumenta poco o aumenta mucho cuando la n crece.

I.1 La eficiencia de un algoritmo

Principio de invarianza.- Dado un algoritmo y dos implementaciones suyas I_1 e I_2, que tardan $T_1(n)$ y $T_2(n)$ respectivamente, el *Principio de invarianza* afirma que existe una constante real $c > 0$ y un número natural n_0 tales que para todo $n \geq n_0$ se verifica que:

$$T_1(n) \leq cT_2(n).$$

Es decir, el tiempo de ejecución de dos implementaciones distintas de un algoritmo dado no va a diferir más que en una constante multiplicativa [Guerequeta y Vallecillo, 2000]. Esto significa que el tiempo para resolver un problema mediante un algoritmo depende de la naturaleza del algoritmo y no de la implementación del algoritmo.

Decimos entonces que el tiempo de ejecución de un algoritmo es asintóticamente de *del orden de* $T(n)$ si existen dos constantes reales c y n_0 y una implementación I del algoritmo tales que el problema se resuelve en menos de $cT(n)$, para toda $n > n_0$.

Cuando se quiere comparar la *eficiencia temporal* de dos algoritmos, tiene mayor influencia el tipo de función que la constante c. Esto lo podemos apreciar en el ejemplo de la *Figura 1.4*:

n	Algoritmo 1: $T_1(n) = 10^6 n^2$		Algoritmo 2: $T_2(n) = 5 n^3$
2	$10^6 \times 4$: 4,000,000	>	5×8: 40
200	$10^6 \times 40{,}000$: 4×10^{10}	>	$5 \times 8 \times 10^6$: 4×10^7
200,000	$10^6 \times 4 \times 10^{10}$: 4×10^{16}	=	$5 \times 8 \times 10^{15}$: 4×10^{16}
2,000,000	$10^6 \times 4 \times 10^{12}$: 4×10^{18}	<	$5 \times 8 \times 10^{18}$: 4×10^{19}

Figura 1.4: Influye más el tipo de función que la constante multiplicativa

A partir de cierto valor de *n*, la función cúbica es siempre mayor a pesar de que la constante es mucho menor a ésta.

I.2 El tiempo de ejecución $T(n)$ de un algoritmo

Para medir $T(n)$ usamos el número de *operaciones elementales*. Una operación elemental puede ser:

- Operación aritmética.
- Asignación a una variable.
- Llamada a una función.
- Retorno de una función.
- Comparaciones lógicas (con salto).
- Acceso a una estructura (arreglo, matriz, lista ligada…).

Se le llama *tiempo de ejecución*, no al tiempo físico, sino al número de operaciones elementales que se llevan a cabo en el algoritmo.

I.3 Ejercicios

I.3.1.- Análisis de operaciones elementales en un algoritmo que busca un número dado dentro de un arreglo ordenado

Veamos cuantas operaciones elementales contiene el algoritmo del ejemplo de la *Figura 1.5*.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int ArregloOrdenado[9]={1,3,7,15,19,24,31,38,40};   1 OE (asignación)
    int buscado, j=0;   1 OE (asignación)

    cout<<"¿Que numero entero quieres buscar en el arreglo?";   1 OE (salida)
    cin>>buscado;   1 OE (entrada)

    while(ArregloOrdenado[j] < buscado && j<9 )   4 OE (1 acceso + 2 comparaciones+ 1 AND)
      j = j+1;   2 OE ( incremento + asignación)

    if ( ArregloOrdenado[j] == buscado)   2 OE (acceso + comparación)
      cout<<"tu entero si esta en el arreglo";   1 OE (salida)

    else
      cout<<" lastima! no esta";   1 OE (salida)
    system("PAUSE");   1 OE (pausa)

    return EXIT_SUCCESS;   1 OE (regreso)
}
```

¿Cuántas operaciones elementales se realizan en este algoritmo?

Figura 1.5: Análisis de operaciones elementales en un algoritmo que busca un número dado dentro de un arreglo ordenado

I.3.2.- Análisis de operaciones elementales en un algoritmo que busca un número dado dentro de un arreglo ordenado (mejor caso)

Ahora, en la *Figura 1.6*, analizaremos el número de operaciones elementales de este mismo algoritmo tomando en cuenta *el mejor caso*, es decir, cuando el número buscado se encuentra inmediatamente en el primer elemento del arreglo, entonces:

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int ArregloOrdenado[9]={1,3,7,15,19,24,31,38,40};   1 OE (asignación)
    int buscado, j=0;   1 OE (asignación)

    cout<<"¿Que numero entero quieres buscar en el arreglo?";   1 OE (salida)
    cin>>buscado;   1 OE (entrada)

    while(ArregloOrdenado[j] < buscado && j<9 )   2 OE (1 acceso + 1 comparación)
      j = j+1;

    if ( ArregloOrdenado[j] == buscado)   2 OE (acceso + comparación)
      cout<<"tu entero si esta en el arreglo";   1 OE (salida)

    else
      cout<<" lastima! no esta";
    system("PAUSE");   1 OE (pausa)
    return EXIT_SUCCESS;   1 OE (regreso)
}
```

EL MEJOR CASO: T(9)= 4+2+2+3=11

Figura 1.6: Análisis de operaciones elementales en el mejor caso, es decir, cuando se encuentra inmediatamente el número buscado

Nótese que en el `while` sólo se ejecutan 2 OE porque como la primera comparación es falsa ya no se hace la segunda, ni tampoco se ejecuta la operación lógica AND, entonces, el número de total de operaciones elementales está dado por: T(9)= 4+2+2+3=11.

I.3.3.- Análisis de operaciones elementales en un algoritmo que busca un número dado dentro de un arreglo ordenado (peor caso)

En la *Figura 1.7*, analizamos el algoritmo para *el peor caso* analizaremos, es decir, cuando se recorre todo el arreglo y no se encuentra al elemento buscado.

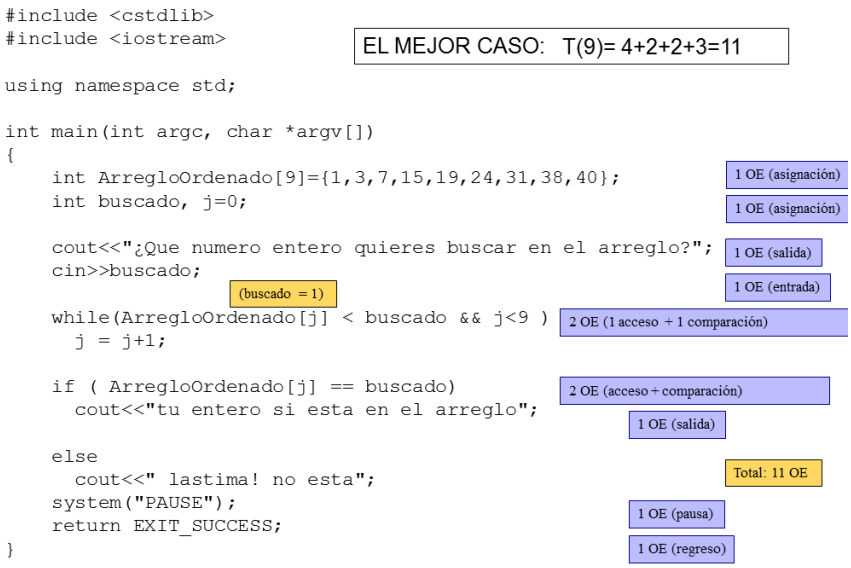


Figura 1.7: Análisis de operaciones elementales, en el peor caso, es decir, cuando no se encuentra el elemento buscado

Hay que tomar en cuenta que el ciclo `while` se ejecuta *n* veces, donde *n* es el tamaño del arreglo, en este caso 9. Además, una vez que se ejecuta este ciclo 9 veces, se hacen otras 4 operaciones elementales para verificar que se terminó la búsqueda, de tal manera que el número total de OE está dado por:

$$T(n) = 4 + \sum_{j=0}^{n-1}(4+2) + 4 + 2 + 3 = 4 + 6n + 9$$

Para $n = 9$ $T(n)$ es 67.

I.3.4.- Cálculo del tiempo de ejecución en un ciclo `for`

Normalmente se considera *el peor caso*, cuando se calcula el número de OE. Consideramos que el tiempo de una OE es, por definición, de orden 1. El tiempo de ejecución de una secuencia consecutiva de instrucciones se calcula sumando los tiempos de ejecución de cada una de las instrucciones.

Para calcular el tiempo de ejecución del ciclo `for` se puede usar su forma equivalente, como se indica en la *Figura 1.8.*

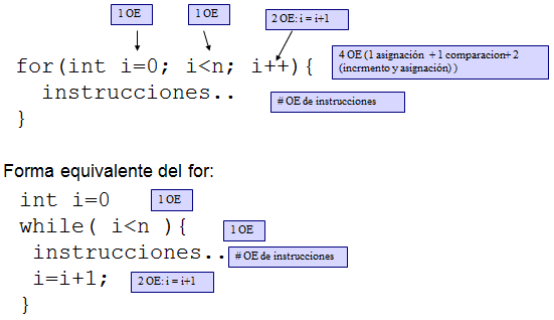


Figura 1.8: Equivalencia entre los ciclos `while` y `for`

El tiempo total en el peor caso es:

$$1+1+\sum_{i=0}^{n-1}\left(\#\text{OE de instrucciones}+2+1\right)$$

El primer 1 es el de la asignación, el segundo 1 es el de la última comparación (cuando ya no se cumple la condición) y al final hay que sumar el resultado de multiplicar *n* veces el número de OE de las instrucciones dentro del ciclo más 2 OE del incremento más 1 OE de la comparación, es decir *n*(#OE de instrucciones +2+1).

I.4 Resumen del capítulo

En el capítulo I definimos lo que es un algoritmo y asentamos que el análisis de algoritmos se encarga de estudiar la eficiencia de éstos en el tiempo y en el espacio, es decir, el tiempo que se emplea en resolver el problema y la cantidad de recursos de memoria que ocupa. El tiempo que requiere un algoritmo para resolver un problema está en función del tamaño *n* del conjunto de datos para procesar, y se le llama *tiempo de ejecución T*(*n*). También estudiamos el *Principio de invarianza* el cual afirma que el tiempo para resolver un problema depende de la naturaleza del algoritmo empleado y no de la implementación del algoritmo. Para analizar la complejidad de un algoritmo se utiliza la noción de *orden* del algoritmo. En el siguiente capítulo estudiaremos las medidas asintóticas que permiten expresar la complejidad de los algoritmos en términos de su orden.

Capítulo II Análisis de la complejidad algorítmica

II.1 Medidas asintóticas

Las *cotas de complejidad*, también llamadas *medidas asintóticas* sirven para clasificar funciones de tal forma que podamos compararlas. Las *medidas asintóticas* permiten analizar qué tan rápido crece el tiempo de ejecución de un algoritmo cuando crece el tamaño de los datos de entrada, sin importar el lenguaje en el que esté implementado ni el tipo de máquina en la que se ejecute.

Existen diversas notaciones asintóticas para medir la complejidad, las tres cotas de complejidad más comunes son: la notación **O** (o mayúscula), la notación **Ω** (omega mayúscula) y la notación **θ** (theta mayúscula) y todas se basan en *el peor caso*.

II.1.1 Cota superior asintótica: Notación O (o mayúscula)

O($g(n)$) es el conjunto de todas las funciones f_i para las cuales existen constantes enteras positivas k y n_0 tales que para $n \geq n_0$ se cumple que:

$$f_i(n) \leq kg(n)$$

$kg(n)$ es una "cota superior" de toda f_i para $n \geq n_0$

Cuando la función $T(n)$ está contenida en **O**($g(n)$), entonces la función $cg(n)$ es una cota superior del tiempo de ejecución del algoritmo para alguna c y para toda $n \geq n_0$. Lo que indica que dicho algoritmo nunca tardará más que: $k\ g(n)$. Recordar que el *tiempo de ejecución* es el número de operaciones elementales que se llevan a cabo y que n es el tamaño de los datos de entrada.

Por ejemplo, si el tiempo de ejecución $T(n)$ de un algoritmo es **O**(n^2) se tiene que:

$$T(n) \leq k\, n^2 \quad \text{para} \quad n \geq n_0$$

donde k y n_0 son constantes enteras y positivas.

Entonces, $T(n) \in \mathbf{O}(n^2)$ significa que el programa nunca tardará más de kn^2 cuando $n \geq n_0$. Lo anterior puede comprenderse aún mejor con la gráfica de la *Figura 2.1*, en la cual se observa que $kg(n)$ es una función que acota por arriba a $f(n)$, ya que $f(n)$ nunca será mayor que $kg(n)$ a partir de $n = n_0$ para un valor de k suficientemente grande.

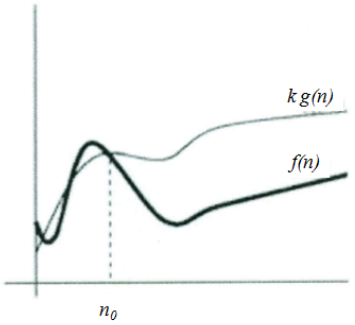


Figura 2.1: $f(n) \in \mathbf{O}(g(n))$ [Villalpando, 2003]

En resumen, si $f(n) \leq kg(n)$ para todo $n > n_0$ implica que $f(n)$ es $\mathbf{O}(g(n))$. Se puede decir entonces que, cuando el valor de n se hace grande, $f(n)$ está acotada por $kg(n)$. Este es un concepto importante en la práctica ya que, cuando el tiempo de ejecución de un algoritmo está acotado por alguna función, podemos predecir un máximo de tiempo para que termine en función de n.

II.1.2 Cota inferior asintótica: Notación Ω (omega mayúscula)

$\mathbf{\Omega}(g(n))$ es el conjunto de todas las funciones f_i para las cuales existen constantes enteras positivas k y n_0 tales que para $n \geq n_0$ se cumple que:

$$f_i(n) \geq kg(n)$$

$kg(n)$ es una "cota inferior" de toda f_i para $n \geq n_0$

Por ejemplo:

Si el tiempo de ejecución $T(n)$ de un programa es $\mathbf{\Omega}(n^2)$ implica que:

$$T(n) \geq k\, n^2 \qquad n \geq n_0$$

Donde k y n_0 son constantes enteras y positivas.

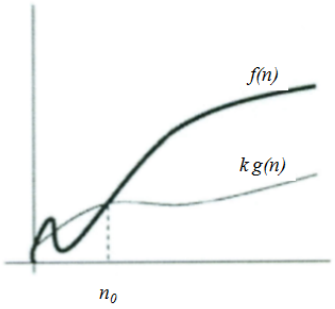


Figura 2.2: $f(n) \in \Omega(g(n))$ [Villalpando, 2003]

Entonces, $T(n) \in \mathbf{\Omega}(n^2)$ significa que el programa nunca tardará menos de kn^2. Lo anterior puede comprenderse aún mejor con la gráfica de la *Figura 2.2*, en la cual se observa que $kg(n)$, para nuestro ejemplo kn^2, es una función que acota por debajo a $f(n)$, ya que $f(n)$ nunca será mayor que $kg(n)$. También puede apreciarse que esto se cumple solo para valores de n en los que se cumple que $n \geq n_0$.

En resumen, $f(n) \geq kg(n)$ implica que $f(n)$ es $\mathbf{\Omega}(g(n))$, es decir, $f(n)$ crece asintóticamente más rápido que $g(n)$ cuando $n \rightarrow \infty$.

II.1.3 Orden exacto o cota ajustada asintótica: Notación θ (theta mayúscula)

$\mathbf{\theta}(g(n))$ es el conjunto de todas las funciones f_i para las cuales existen constantes enteras positivas k_1 ,k_2 y n_0 tales que para $n \geq n_0$ se cumple que:

$k_1 g(n) \leq f_i(n) \leq k_2 g(n)$

En la *Figura 2.3* se puede apreciar que una función $f(n)$ que es $\mathbf{\theta}(g(n))$ está acotada por arriba por $k_1 g(n)$ y por debajo está acotada por la función $k_2 g(n)$.

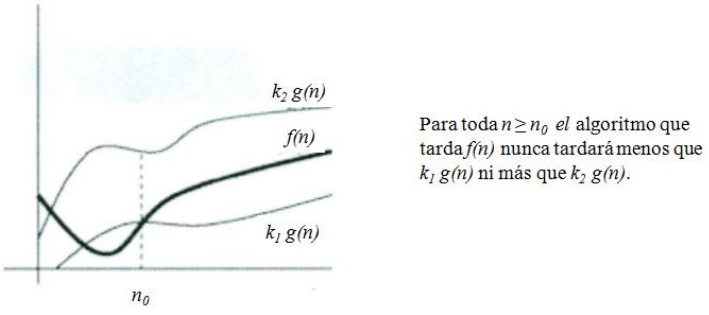


Figura 2.3: $f(n) \in \mathbf{\theta}(g(n))$ [Villalpando, 2003]

En resumen, $k_1 g(n) \leq f(n) \leq k_2 g(n)$ implica que $f(n)$ es $\mathbf{\theta}(g(n))$, es decir, $f(n)$ y $g(n)$ crecen asintóticamente a la misma velocidad cuando $n \rightarrow \infty$.

En la *Figura 2.4* se muestra la relación que existe entre las funciones acotadas por **Ω**(*g*(*n*)) las acotadas por **θ** (*g*(*n*)) y las acotadas por **O**(*g*(*n*)).

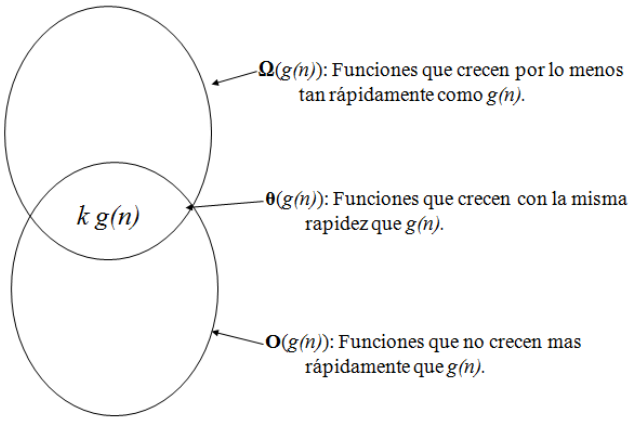


Figura 2.4: *f* (*n*) ∈ **θ** (*g*(*n*)) [Abellanas, 1990]

II.2 Propiedades de la notación O

La notación **O** indica la manera en la que aumenta el tiempo de ejecución de un algoritmo en función del tamaño de la entrada. La forma en la que esto sucede se denota por una función *g*(*n*). Cuando el tiempo de ejecución *T*(*n*) de un algoritmo es **O**(*f*(*n*)) entonces la función *f*(*n*) indica la forma de una cota superior del tiempo de ejecución del algoritmo. Esto indica que dicho algoritmo nunca tardará más que *k f* (*n*). En estas notas trabajaremos con la notación **O** para analizar la complejidad de los algoritmos.

Para cualesquiera funciones *f, g* y *h*, la notación **O** tiene las siguientes propiedades:

1. *Autocontención:* $f \in \mathbf{O}(f)$.
2. *Subconjunto:* $f \in \mathbf{O}(g) \Rightarrow \mathbf{O}(f) \subset \mathbf{O}(g)$.
3. *Identidad:* $\mathbf{O}(f) = \mathbf{O}(g) \Leftrightarrow f \in \mathbf{O}(g)$ y $g \in \mathbf{O}(f)$. Nota: no implica que $f = g$.
4. *Transitividad:* Si $f \in \mathbf{O}(g)$ y $g \in \mathbf{O}(h) \Rightarrow f \in \mathbf{O}(h)$.
5. *El menor:* Si $f \in \mathbf{O}(g)$ y $f \in \mathbf{O}(h) \Rightarrow f \in \mathbf{O}(\min(g,h))$.

6. Regla de la suma:

   Si $f_1 \in \mathbf{O}(g)$ y $f_2 \in \mathbf{O}(h) \Rightarrow f_1 + f_2 \in \mathbf{O}(\max(g,h))$.

7. Regla del producto:

   Si $f_1 \in \mathbf{O}(g)$ y $f_2 \in \mathbf{O}(h) \Rightarrow f_1 \cdot f_2 \in \mathbf{O}(g \cdot h)$.

8. Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k$, dependiendo del valor de k, tenemos:

   a) *Prueba de Identidad:* Si $k \neq 0$ y $|k| < \infty$, entonces $\mathbf{O}(f) = \mathbf{O}(g)$.

   b) *Prueba de inclusión:* Si $k = 0$, entonces $f \in \mathbf{O}(g)$, es decir,

   $\mathbf{O}(f) \subset \mathbf{O}(g)$, además se verifica que $g \notin \mathbf{O}(f)$.

   c) *Prueba de exclusión:* Si $k = \pm \infty$, entonces $f \notin \mathbf{O}(g)$, es decir,

   $\mathbf{O}(f) \supset \mathbf{O}(g)$, además se verifica que $g \in \mathbf{O}(f)$.

II.2.1 Ejercicios con la notación O

Para facilitar a los alumnos la resolución de los ejercicios, proporcionamos la regla de L'hopital, una tabla con las derivadas que utilizaremos y también las leyes de los exponentes.

Regla de L'hopital:

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$$

Mini-tabla de derivadas:

$$\frac{dx}{dx} = 1$$

$$\frac{dv^m}{dx} = mv^{m-1}\left(\frac{dv}{dx}\right)$$

$$\frac{d\log_b v}{dx} = \left(\frac{dv}{dx}\right)\left(\frac{1}{v\ln b}\right)$$

$$\frac{da^x}{dx} = a^x \ln a$$

$$\frac{du\cdot v}{dx} = u\cdot\frac{dv}{dx} + v\frac{du}{dx}$$

Leyes de los exponentes:

$$\frac{a^m}{b^m} = \left(\frac{a}{b}\right)^m$$

$$a^m a^n = a^{m+n}$$

$$\frac{a^m}{a^n} = a^{m-n}$$

$$\left(a^m\right)^n = a^{m\cdot n}$$

Primero resolveremos algunos de los ejercicios propuestos por [Guerequeta y Vallecillo, 2000].

Indicar si las siguientes afirmaciones son ciertas.

1. $n^2 \in \mathbf{O}(n^3)$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = k$$

Para este caso:

$$\lim_{n\to\infty}\frac{n^2}{n^3} = \lim_{n\to\infty}\frac{1}{n} = 0$$

Entonces, por la propiedad 8b la proposición es cierta.

2. $n^3 \in \mathbf{O}(n^2)$

Obtendremos $\lim_{n\to\infty}\frac{f(n)}{g(n)} = k$ como en el ejercicio anterior:

$$\lim_{n\to\infty}\frac{n^3}{n^2} = \lim_{n\to\infty} n = \infty$$

Entonces, como el límite es ∞, por la propiedad 8c sabemos que la proposición es falsa.

3. $2^{n+1} \in \mathbf{O}(2^n)$

Obteniendo el límite:

$$\lim_{n\to\infty} \frac{2^{n+1}}{2^n} = \lim_{n\to\infty} \frac{2^n \cdot 2}{2^n} = 2$$

Podemos concluir, de la propiedad 8a, que

$$\mathbf{O}(2^n+1) = \mathbf{O}(2^n)$$

4. $(n+1)! \in \mathbf{O}(n!)$

Al obtener:

$$\lim_{n\to\infty} \frac{(n+1)!}{n!} = \lim_{n\to\infty} \frac{(n+1)n!}{n!} = \lim_{n\to\infty} \frac{n+1}{1} = \infty$$

Observamos que el límite es ∞, por lo que concluimos que la proposición es falsa.

5. $3^n \in \mathbf{O}(2^n)$

$$\lim_{n\to\infty} \frac{3^n}{2^n} = \infty$$

3^n crece más rápido que 2^n, el límite es ∞ y la proposición es falsa.

6. $log_2 n \in \mathbf{O}(n^{1/2})$

Obteniendo el límite siguiente:

$$\lim_{n\to\infty} \frac{log_2 n}{n^{1/2}} = \lim_{n\to\infty} \frac{\frac{1}{nln(2)}}{\frac{1}{2}n^{-1/2}} = \lim_{n\to\infty} \frac{2}{\ln(2)n^{1-1/2}} = \lim_{n\to\infty} \frac{2}{\ln(2)n^{1/2}} = 0$$

Por la propiedad 8b concluimos que: $log_2 n \in \mathbf{O}(n^{1/2})$ pero $n^{1/2} \notin \mathbf{O}(log_2 n)$

7.- Determinar la complejidad de la siguiente función:

$f_1(n) = 100n^2 + 10n + 1$

Si obtenemos:

$$\lim_{n\to\infty} \frac{100n^2 + 10n + 1}{n^2} = \lim_{n\to\infty} (100 + \frac{10}{n} + \frac{1}{n^2}) = 100$$

Podemos concluir que, el orden de la función $f_1(n) = 100n^2 + 10n + 1$ es n^2, es decir:

$\mathbf{O}(\ 100n^2 + 10n + 1) = \mathbf{O}(\ n^2\ )$

8.- Cual es la complejidad de la función:

$f_2(n) = 5n^3 + 2n$

Si obtenemos:

$$\lim_{n\to\infty} \frac{5n^3 + 2n}{n^3} = \lim_{n\to\infty} (5 + \frac{2}{n^2}) = 5$$

Podemos concluir que el orden de la función $f_2(n) = 5n^3 + 2n$ es n^3, es decir:

$\mathbf{O}(5n^3 + 2n) = \mathbf{O}(\ n^3\ )$

II.3 Clasificando algoritmos con la notación O

La notación **O** sirve para identificar si un algoritmo tiene un orden de complejidad mayor o menor que otro, un algoritmo es más eficiente mientras menor sea su orden de complejidad. En los siguientes ejercicios utilizaremos la *propiedad 8* de la notación **O** para ordenar de menor a mayor los órdenes de complejidad de diferentes funciones.

Ejercicio 1.- ordenar de mayor a menor:

1.- 2^n 2.- $n \log (n)$ 3.- n^8

Para determinar el orden de complejidad entre 2^n y $n \log (n)$ calcularemos

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = k$$

y observaremos el valor de *k*.

$$\lim_{n\to\infty}\frac{n\log_2(n)}{2^n}=\lim_{n\to\infty}\frac{\frac{n}{\ln(2)n}+\log_2(n)}{\ln(2)\,2^n}=\lim_{n\to\infty}\frac{\frac{1}{\ln(2)n}}{\ln^2(2)2^n}=$$

$$=\lim_{n\to\infty}\frac{1}{\ln^3(2)n2^n}=0$$

En este caso, como $k = 0$, $f \in O(g)$, es decir, $O(f) \subset O(g)$. De lo anterior podemos concluir que $\mathbf{O}(n\log_2 n) \subset \mathbf{O}(2^n)$.

De una forma similar evaluaremos las funciones $n \log_2 (n)$ y n^8:

$$\lim_{n\to\infty}\frac{nlog_2(n)}{n^8}=\lim_{n\to\infty}\frac{log_2(n)}{n^7}=\lim_{n\to\infty}\frac{\frac{1}{\ln(2)\,n}}{7n^6}=\lim_{n\to\infty}\frac{1}{7\ln(2)n^7}=0$$

De lo anterior podemos concluir que $\mathbf{O}(n\log_2 n) \subset \mathbf{O}(n^8)$

Ahora solo falta analizar como es 2^n con respecto a n^8:

$$\lim_{n\to\infty}\frac{2^n}{n^8}=\lim_{n\to\infty}\frac{\ln(2)2^n}{8n^7}=\infty$$

Por la propiedad 8c concluimos que $\mathbf{O}(n^8) \subset \mathbf{O}(2^n)$.

Si invirtiéramos el orden del cociente:

$$\lim_{n\to\infty}\frac{n^8}{2^n}=\lim_{n\to\infty}\frac{8n^7}{\ln(2)2^n}=\lim_{n\to\infty}\frac{56n^6}{\ln^2(2)2^n}=\ldots=\lim_{n\to\infty}\frac{8!}{\ln^8(2)2^n}=0$$

Por la propiedad 8b llegamos a la misma conclusión. Por lo tanto, el orden de complejidad de las tres funciones, de menor a mayor es el siguiente:

$$\mathbf{O}(n\log_2 n) \subset \mathbf{O}(n^8) \subset \mathbf{O}(2^n)$$

El algoritmo cuyo orden de complejidad es $n\log_2 n$ resulta ser el más eficiente, mientras que el algoritmo cuyo orden de complejidad es 2^n es el menos eficiente.

Ejercicio 2.- ¿Cuál de las siguientes funciones tiene menor complejidad?

1.- $\log_2 (n)$ 2.- $n \log_2 (n)$

Si obtenemos el siguiente límite:

$$\lim_{n\to\infty}\frac{\log_2(n)}{n\log_2(n)}=\lim_{n\to\infty}\frac{1}{n}=0$$

Podemos concluir que **O**(log *n*) ⊂ **O**(*n* log *n*), es decir, **O**(*n* log *n*) es peor que **O**(log *n*).

II.4 Los diferentes tipos de complejidad

Existen diferentes tipos de complejidad, lo que se desea de un algoritmo es que su complejidad sea la menor posible. A continuación se presentan diferentes tipos de complejidad (las más comunes) ordenadas de menor a mayor [Villalpando, 2003].

O(1) **Complejidad constante**.- Es la más deseada. Por ejemplo, es la complejidad que se presenta en secuencias de instrucciones sin repeticiones o ciclos (ver ejemplo 1 en la sección II.4.1).

O(log *n*) **Complejidad logarítmica**.- Puede presentarse en algoritmos iterativos y recursivos (ver ejemplo 2 en la sección II.8).

O(*n*) **Complejidad lineal**.- Es buena y bastante usual. Suele aparecer en un *ciclo* principal simple cuando la complejidad de las operaciones interiores es constante (ver ejemplo 2 en la sección II.4.1).

O(*n* log *n*) **Complejidad n·log·n.-** Se considera buena. Aparece en algunos algoritmos de ordenamiento (ver ejemplo 7 en la sección II.4.1).

O(n^2) **Complejidad cuadrática**.- Aparece en ciclos anidados, por ejemplo *ordenación por burbuja*. También en algunas recursiones dobles (ver ejemplo 3 en la sección II.4.1).

O(n^3) **Complejidad cúbica**.- En ciclos o en algunas recursiones triples. El algoritmo se vuelve lento para valores grandes de *n*.

O(n^k) **Complejidad polinomial**.- Para (k∈ **N**, k>3) mientras más crece *k*, el algoritmo se vuelve más lento (ver ejemplo 4 en la sección II.4.1).

O(C^n) **Complejidad exponencial**.- Cuando *n* es grande, la solución de estos algoritmos es prohibitivamente costosa. Por ejemplo, problemas de explosión combinatoria.

Cuando se emplean las cotas asintóticas para evaluar la complejidad de los algoritmos, el análisis tiene sentido para entradas de tamaño considerable, ya que en ocasiones, comparar algoritmos utilizando un número pequeño de datos de entrada puede conducir a conclusiones incorrectas sobre sus bondades [López et al., 2009]. Esto se debe a que, cuando *n* es pequeña, varias funciones tienen un comportamiento hasta cierto punto similar, por ejemplo $\mathbf{O}(n^3)$, $\mathbf{O}(n^2)$, $\mathbf{O}(n \log(n))$ y $\mathbf{O}(n)$. Sin embargo, conforme *n* crece, el comportamiento de las funciones cambia de manera radical. Esto se puede apreciar en la *Figura 2.5*.

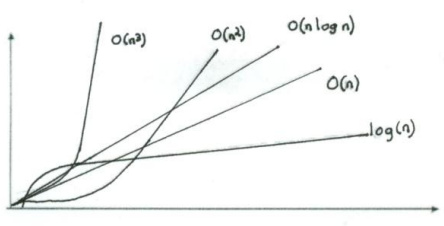


Figura 2.5: Complejidad algunas funciones

Para saber qué programa es mejor, el que implementa un algoritmo con tiempo de ejecución $\mathbf{O}(n^2)$ u otro que implementa un algoritmo con tiempo $\mathbf{O}(n^3)$ obtenemos el límite:

$$\lim_{n\to\infty} \frac{n^2}{n^3} = \lim_{n\to\infty} \frac{1}{n} = 0$$

lo que indica que el tiempo de ejecución del primero crece más lentamente que el segundo conforme *n* crece. De lo anterior podemos concluir que $\mathbf{O}(n^2) \subset \mathbf{O}(n^3)$, es decir un $\mathbf{O}(n^2)$ es más rápido y por lo tanto más eficiente que $\mathbf{O}(n^3)$.

Ahora, si tenemos dos casos particulares de lo anterior:

1. $f_1(n) = 100n^2$ $\quad f_1(n) \subset \mathbf{O}(n^2)$
2. $f_2(n) = 5n^3$ $\quad f_2(n) \subset \mathbf{O}(n^3)$

¿Cuál algoritmo es más rápido, el que tiene un tiempo de ejecución $f_1(n) = 100n^2$ o el que tiene un tiempo de ejecución $f_2(n) = 5n^3$?

Para saber como influye el valor de la constante multiplicativa, la Tabla 2.1 muestra el comportamiento de cada algoritmo para diferentes valores de *n*.

n	Algoritmo1 $f_1(n)=100n^2$	Algoritmo 2 $f_2(n)= 5n^3$
5	2,500	625
10	10,000	5,000
15	22,500	16,875
20	40,000	40,000
50	250,000	625,000
200	4,000,000	40,000,000

Tabla 2.1: Comportamiento de dos algoritmos para diferentes tamaños de entrada

De la *Tabla 2.1* podemos apreciar que para $n < 20$ el algoritmo 2 es mejor que el primero, en general, para tamaños de entrada grande, el algoritmo 1 es mejor que el segundo, ya que $\mathbf{O}(n^2) \subset \mathbf{O}(n^3)$, es decir el tiempo de ejecución de un algoritmo de orden $\mathbf{O}(n^2)$ es menor que uno de orden $\mathbf{O}(n^3)$. Se han hecho experimentos utilizando una computadora muy lenta para un algoritmo de orden $\mathbf{O}(n^2)$ y una computadora muy rápida para un algoritmo de orden $\mathbf{O}(n^3)$ [López et al, 2009] y con estos experimentos se comprobó que el orden asintótico tiene una mayor influencia que el tipo de computadora en la que se ejecute el algoritmo.

II.4.1 Funciones de complejidad más usuales

En la *tabla 2.2* se presenta un resumen de las funciones de complejidad más usuales ordenadas de la mejor a la peor, es decir, desde la más eficiente hasta la menos eficiente.

- **O(1). Complejidad constante.**
- **O(log *n*). Complejidad logarítmica.**
- **O(*n*). Complejidad lineal.**
- **O(*n* log *n*).**
- ***O*(n^2). Complejidad cuadrática.**
- **O(n^3). Complejidad cúbica.**
- **O(n^k). Complejidad polinomial.**
- **O(2^n). Complejidad exponencial.**

Tabla 2.2: : Algunas funciones ordenadas de menor a mayor complejidad.

Ahora analizaremos algunas de las funciones de la *tabla 2.2*:

1.- La complejidad constante **O**(1).

$f(n) = c$ es **O**(1) porque $\lim_{n\to\infty} \frac{c}{1} = c$ *y se cumple que* $0 \le c < \infty$

Además, existe una constante c tal que $f(n) \le c$ para toda $n > n_0$, esto se expresa de la siguiente manera en lenguaje matemático:

$$\exists\, c \;\mid\; f(n) \le c \quad \forall \; n > n_0$$

Así por ejemplo, $f(n) = 5$ es **O**(1) porque $\lim_{n\to\infty} \frac{5}{1} = 5$

Cuando $f(n) = c$, el tiempo que tarda el algoritmo es constante y por lo tanto no depende del tamaño de la entrada.

Un algoritmo con orden de complejidad $f(n) = c \in \mathbf{O}(1)$ es el algoritmo más rápido.

2.- La complejidad lineal **O**(n).

$f(n)$ es **O**(n) si y solo si $\exists\, c \;\mid\; f(n) \le cn \quad \forall \; n > n_0$

Así, por ejemplo, $f(n) = 2400+3n$ es **O**(n) porque

$$\lim_{n\to\infty} \frac{2400+3n}{n} = \lim_{n\to\infty} \frac{3}{1} = 3$$ (por la regla de L'hopital)

Este límite existe, además puede observarse que un algoritmo es **O**(n) cuando la función $f(n)$ es cualquier función polinomial de primer grado.

Un algoritmo con orden de complejidad **O**(n) es un algoritmo bastante rápido.

3.- La complejidad cuadrática **O**(n^2).

$f(n)$ es **O**(n^2) si y solo si $\exists\, c \;\mid\; f(n) \le cn^2 \quad \forall \; n > n_0$

Así por ejemplo $f(n) = 37+100n+12n^2$ es **O**(n^2) porque:

$$\lim_{n\to\infty} \frac{37+100n+12n^2}{n^2} = \lim_{n\to\infty} \frac{100+24n}{2n} = \lim_{n\to\infty} \frac{24}{2} = 12$$

Este límite existe, además puede observarse que un algoritmo es **O**(n^2) cuando la función $f(n)$ es cualquier función polinomial de segundo grado.

Un algoritmo cuyo orden de complejidad es $f(n) \in \mathbf{O}(n^2)$ es un algoritmo no tan rápido como uno de orden **O**(n).

4.- La complejidad polinomial **O** (n^k).

$f(n)$ es **O**(n^k) si y solo si $\exists\, c \;\mid\; f(n) \le cn^k \quad \forall \; n > n_0$

Un algoritmo cuyo orden de complejidad sea $f(n) \in \mathbf{O}(n^k)$ será más lento conforme aumenta k.

En general, $f(n)$ es $\mathbf{O}(n^k)$ para cualquier función polinomial de la forma:

$$a_k n^k + a_{k-1} n^{k-1} + a_{k-2} n^{k-2} + ... + a_1 n + a_0$$

Nótese que, por la propiedad 8, para $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k$, Si $k \neq 0$ y $k < \infty$ entonces $\mathbf{O}(f) = \mathbf{O}(g)$.

Cuando una función dada $g(n)$ es más sencilla que $f(n)$, y se cumple el inciso a) de la propiedad 8, podemos decir que $\mathbf{O}(f) = \mathbf{O}(g)$ así:

$f(n) = 5$ es $\mathbf{O}(1)$	$g(n) = 1$
$f(n) = 2400+3n$ es $\mathbf{O}(n)$	$g(n) = n$
$f(n) = 37+100n+12n^2$ es $\mathbf{O}(n^2)$	$g(n) = n^2$
$f(n) = a_k n^k + a_{k-1} n^{k-1} + ... + a_1 n + a_0$ es $\mathbf{O}(n^k)$	$g(n) = n^k$

5.- La complejidad factorial $\mathbf{O}(n!)$ vs. la complejidad exponencial $\mathbf{O}(2^n)$.

Con la complejidad factorial $\mathbf{O}(n!)$ se tiene que $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot ..... \cdot n$

Un ejemplo de complejidad exponencial es $\mathbf{O}(2^n)$ en la cual se tiene que 2 se multiplica n veces: $2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot ..... \cdot 2$ (n veces)

Si hacemos la división de $\mathbf{O}(n!)$ entre $\mathbf{O}(2^n)$ podemos observar que $\mathbf{O}(n!)$ crece más rápido:

$$\frac{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot ..... \cdot n}{2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot ..... \cdot 2}$$

Por lo tanto $\mathbf{O}(n!)$ es peor que $\mathbf{O}(2^n)$.

6.- La complejidad exponencial cuando la base es un cociente.

En la *Figura 2.6* se aprecia lo diferente que se comporta un algoritmo cuyo orden de complejidad es exponencial cuando la base es un cociente <1 y cuando el cociente es >1.

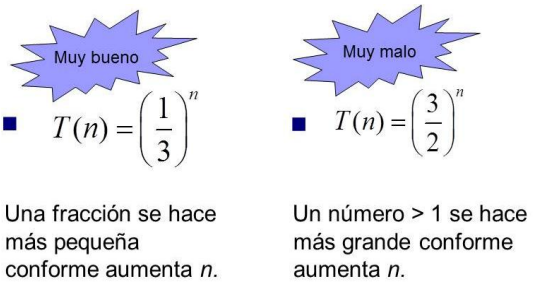


Figura 2.6: diferentes comportamentos cuando el cociente es inferior o superior a la unidad

Sin embargo nunca habrá algoritmos que tarden menos cuando se les encarga más trabajo.

7.- La complejidad **O**(*n* log *n*).

Recordemos que en un árbol binario completo el número total de nodos *n* es: $n = 2^{h+1} - 1$, como se aprecia en la *Figura 2.7*, en donde se indica la cantidad de nodos en cada nivel en función de la altura *h*. Así, por ejemplo, para h = 2 el número total de nodos es la suma del número de nodos en el nivel $h = 0$, más el número de nodos en el nivel $h = 1$, más el número de nodos en el nivel $h = 2$, es decir, para h=2 el número total de nodos será 1+2+4=7.

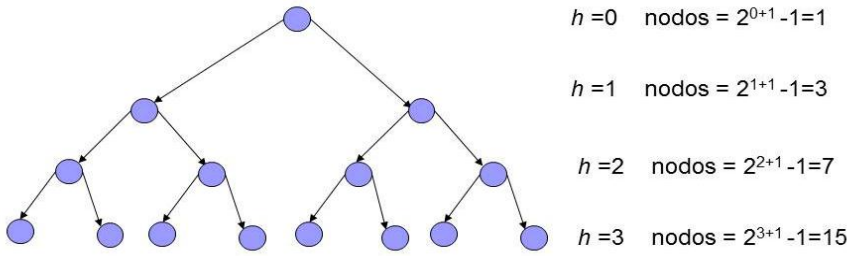


Figura 2.7: número de nodos en un árbol binario completo

Para saber cómo varía la altura *h* respecto al número de nodos en un árbol binario completo, despejaremos *h* de $n = 2^{h+1} - 1$:

$$n+1 = 2^{h+1}$$

$$2^{h+1} = n+1$$

$$h+1 = \log_2(n+1)$$

$$h = \log_2(n+1) - 1$$

Entonces podemos comprobar que:

$h \in \mathbf{O}(\log n)$

es decir, la altura *h* es logarítmica en el número de nodos. Entonces, la complejidad de la búsqueda binaria aumenta con la altura del árbol binario, por eso se dice que este tipo de búsqueda es **O**(log(*n*)).

Ejercicio.- Ordenar las siguientes funciones en orden de menor a mayor complejidad:

$f(n) = 3.4\, n^3$	$f(n) \subset \mathbf{O}(n^3)$
$f(n) = 46\, n \log n$	$f(n) \subset \mathbf{O}(n \log n)$
$f(n) = 2^n$	$f(n) \subset \mathbf{O}(2^n)$
$f(n) = 33n$	$f(n) \subset \mathbf{O}(n)$
$f(n) = 13\, n^2$	$f(n) \subset \mathbf{O}(n^2)$

Las funciones ordenadas de menor a mayor complejidad son las siguientes:

$f(n) = 33n$	$f(n) \subset \mathbf{O}(n)$
$f(n) = 46\, n \log n$	$f(n) \subset \mathbf{O}(n \log n)$
$f(n) = 13\, n^2$	$f(n) \subset \mathbf{O}(n^2)$
$f(n) = 3.4\, n^3$	$f(n) \subset \mathbf{O}(n^3)$
$f(n) = 2^n$	$f(n) \subset \mathbf{O}(2^n)$

II.5 Cálculo de *T*(*n*)

En el análisis de algoritmos se estudia el tiempo y el espacio que requiere el algoritmo durante su ejecución. En cuanto al tiempo, se estudia el número de operaciones elementales en *el peor* caso, en *el mejor caso* o en el caso *promedio.* En este curso nos concentraremos en *el peor caso*. Si estudiamos el peor caso tendremos una idea de cuanto será lo más que podrá tardar el algoritmo. Cuando sea de interés, estudiaremos también el caso promedio, por ejemplo, en el análisis de algunos métodos de ordenamiento.

Como ya se mencionó anteriormente, cuando se estudia el tiempo que requiere un algoritmo durante su ejecución, el tiempo que se emplea para la medición no es el físico (minutos, segundos, miliseg, etc.) sino el número de Operaciones Elementales (OE) realizadas, bajo el supuesto de que todas las operaciones básicas se realizan en una unidad de tiempo. Entonces para el cálculo de *T(n)* es importante tomar en cuenta lo siguiente:

- Considerar que el tiempo de una OE es, por definición, de orden **O**(1).
- El tiempo de ejecución de una secuencia consecutiva de instrucciones se calcula sumando los tiempos de ejecución de cada una de las instrucciones.

A continuación se presentan las ***Reglas generales para el cálculo de T(n)***:

Regla 1: Ciclos.- El tiempo de ejecución de un ciclo es, a lo más, el tiempo de ejecución de las instrucciones que están en el interior del ciclo por el número de iteraciones, como se muestra en la *Figura 2.8*.

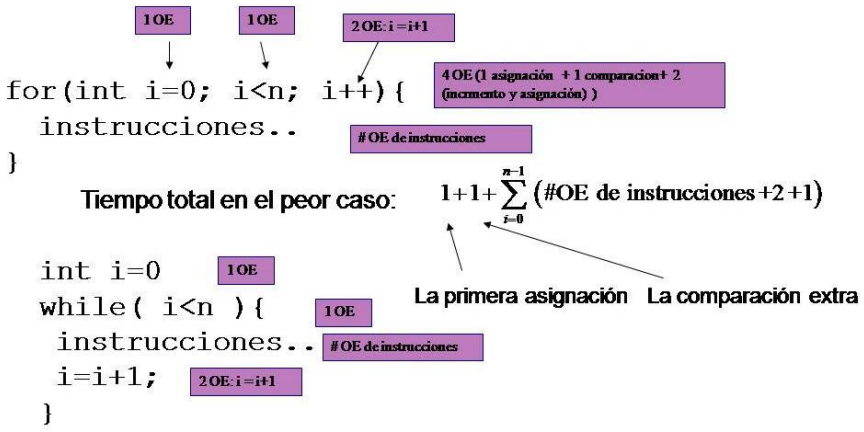
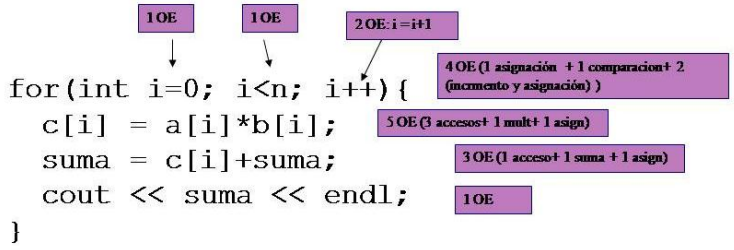


Figura 2.8: Tiempo total de ejecución en el caso en el que se ejecutan todos los ciclos. Equivalencia entre un ciclo `for` y un ciclo `while`

Utilizar la forma equivalente del ciclo `for` ayuda a comprender mejor el número de OE que se ejecutan dentro del ciclo, como se puede apreciar en el ejemplo de la *Figura 2.9*.



Tiempo total en el peor caso: $1+1+\sum_{i=0}^{n-1}(\#OE \text{ de instrucciones}+2+1)$

$$1+1+\sum_{i=0}^{n-1}(9+2+1)=2+12n \in \mathbf{O}(n)$$

Figura 2.9: Tiempo total de ejecución en un ciclo `for`

El valor de la sumatoria de *x* desde *i = j* hasta *i=n* es:

$$\sum_{i=j}^{n} x = (n - j + 1)x$$

Regla 2: Ciclos Anidados.- Analizarlos de adentro hacia fuera. Para calcular el tiempo de ejecución total de ciclos anidados se sigue la misma regla de los ciclos. Como en el siguiente ejemplo de la *Figura 2.10*.

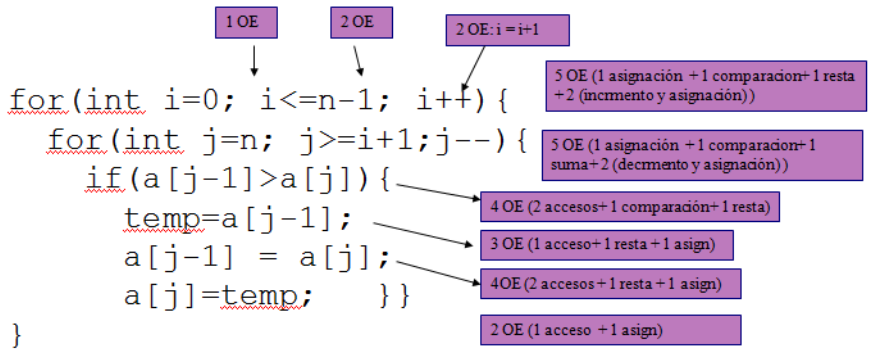


Figura 2.10: Tiempo de ejecución de un algoritmo que ordena un arreglo de menor a mayor

Para calcular el número total de instrucciones:

$$1 + 2 + \sum_{i=0}^{n-1} (\#forinterno + 2 + 2)$$

Para calcular el número de instrucciones del *for* interno:

$$1 + 2 + \sum_{j=i+1}^{n} (13 + 2 + 2) = 3 + \sum_{j=i+1}^{n} 17 = 3 + (i + 1 - n + 1)17 = 17i - 17n + 37$$

Sustituyendo el número de instrucciones del for interno en el número total

$$1 + 2 + \sum_{i=0}^{n-1} (\#forinterno + 2 + 2) = 3 + \sum_{i=0}^{n-1} ((17i - 17n + 37) + 2 + 2) =$$

$$3 + \sum_{i=0}^{n-1} (17i - 17n + 41) = 3 + \frac{17(n-1)n}{2} + (n - 1 + 1)(-17n + 41)$$

$$= \frac{-17}{2} n^2 + \frac{65}{n} + 3 \in O(n^2)$$

Nota: La complejidad de dos ciclos anidados cuya condición de control depende de *n* es **O**(n^2) mientras que la complejidad de ciclos `for` separados es **O**(*n*). Así es que cuando *n* es grande, hay que procurar poner los ciclos `for` por separado cuando sea posible.

A continuación se muestra el cálculo del tiempo total en el peor caso para el `for` anidado:

Número de operaciones elementales para el *for* anidado en el peor caso**:**

$$1 + 1 + \sum_{i=0}^{n-1} (\#OE\ for\ interno + 2 + 1)$$

Sea *x* el número de instrucciones dentro del ciclo interno y el número de operaciones elementales es, en el peor caso**:**

$$1 + 1 + \sum_{i=0}^{n-1} (x + 2 + 1)$$

Entonces el número de OE del ciclo `for` anidado, en el peor caso, es**:**

$$\mathbf{1} + 1 + \sum_{i=0}^{n-1} \left[1 + 1 + \sum_{j=0}^{n-1} (x + 2 + 1) + 2 + 1 \right] = 2 + n(2 + n(3 + x) + 3)$$

$$= 2 + 5n + (3 + x)n^2 \ \in O(n^2)$$

Ahora, si calculamos el número de OE para dos ciclos `for` consecutivos, tenemos que si x_1 es el número de instrucciones dentro del primer ciclo `for`, y x_2 es el número de instrucciones dentro del segundo ciclo `for`, el número de operaciones elementales en el peor caso es, para el primer ciclo**:**

$$1 + 1 + \sum_{i=0}^{n-1} (x_1 + 2 + 1)$$

Y para el segundo ciclo:

$$1 + 1 + \sum_{i=0}^{n-1} (x_2 + 2 + 1)$$

En total:

$$1 + 1 + \sum_{i=0}^{n-1}(x_1 + 2 + 1) + \ 1 + 1 + \sum_{i=0}^{n-1}(x_2 + 2 + 1)$$

$$= \ 4 + n(\ 3 + x_1) + 2 + n(3 + x_2) \ \in O(n)$$

Regla 3: La suma.- supóngase que $T_1(n)$ y $T_2(n)$ son los tiempos de ejecución de dos algoritmos A_1 y A_2 y que:

- $T_1(n)$ es **O**($f(n)$)
- $T_2(n)$ es **O**($g(n)$)

Entonces el tiempo de ejecución de A_1 seguido de A_2 $T_1(n) + T_2(n)$ es:

O(máx($f(n)$,$g(n)$))

Por ejemplo, si:

$T_1(n) \in \mathbf{O}(n^2)$ $\qquad$ $T_2(n) \in \mathbf{O}(n^3)$

Entonces el tiempo de la suma será:

$T_1(n^2) + T_2(n^3) \in \mathbf{O}(max(n^2, n^3))$
$T_1(n^2) + T_2(n^3) \in \mathbf{O}(n^3)$

Regla 4: El producto.- Si $T_1(n)$ y $T_2(n)$ son **O**($f(n)$) y **O**($g(n)$) respectivamente,

entonces $T_1(n)$. $T_2(n)$ es: **O**($f(n)$. $g(n)$))

Recordar que, **O**(c·$f(n)$) significa lo mismo que **O**(f(n)) donde c es una constante.

Por ejemplo, para el caso particular **O**(n^2/2) es lo mismo que **O**(n^2), ya que:

Si $T_1(n)$ y $T_2(n)$ son **O**(c) y **O**(n^2/2) respectivamente,

donde c es una constante, entonces $T_1(n)$. $T_2(n)$ es:

O(c n^2/2)

y **O**(c n^2/2) es lo mismo que **O**(n^2) porque:

$$\lim_{n\to\infty} \frac{\frac{cn^2}{2}}{n^2} = \lim_{n\to\infty} \frac{cn^2}{2n^2} = \frac{c}{2}$$

Regla 5: Llamados a funciones.- El tiempo de ejecución de una llamada a una función *F(P*1, *P*2,..., *Pn)* es 1 (por la llamada), más el tiempo de evaluación de los parámetros *P*1, es decir, el tiempo que se emplea en obtener el valor de cada parámetro, *P*2,..., *Pn*, más el tiempo que tarde en ejecutarse *F*, esto es, $\boldsymbol{T} = 1 + T(P1) + T(P2) + ... + T(Pn) + T(F)$.

Regla 6: El condicional "if".- El tiempo de ejecución de una instrucción condicional "if" es el costo de las instrucciones que se ejecutan condicionalmente, más el tiempo para evaluar la condición.

O(Tiempo(if)) = **O**(Tiempo(condición)) + **O**(Tiempo(cuerpo)).

Esto se ilustra con el ejemplo de la *Figura 2.11*:

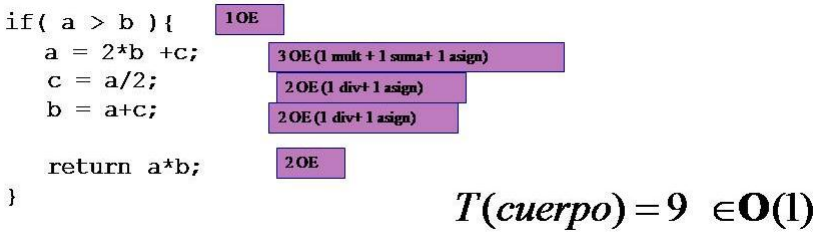


Figura 2.11: Tiempo de ejecución de un algoritmo con el condicional `if`

O(Tiempo(*if*)) = **O**(1) + **O**(1) = **O**(1)

Regla 7: El condicional "if-else".- El tiempo de ejecución es el tiempo para evaluar la condición más el mayor entre los tiempos necesarios para ejecutar las proposiciones cuando la condición es verdadera y el tiempo de ejecución de las proposiciones cuando la condición es falsa

O(Tiempo(if-else)) = **O**(Tiempo(condición)) +

Max(**O**(Tiempo(then),**O**(Tiempo(else))

Lo anterior se ilustra en la *Figura 2.12*:

```
if( a > b ){          1 OE
   a = 2*b +c;        3 OE (1 mult + 1 suma+ 1 asign)
   c = a/2;           2 OE (1 div+ 1 asign)
   b = a+c;           2 OE (1 div+ 1 asign)

   return a*b;        2 OE
}
else{
   for(int i=0;i<n;i++)
     a= a+i;
   return;
}                     1 OE
```

$$T(then) = 9 \in \mathbf{O}(1)$$

Figura 2.12: Tiempo de ejecución de un algoritmo con el condicional `if-else`

El tiempo de ejecución del else es el siguiente (se incluye el 1 del return):

$$Tiempo(else) = 1 + 1 + \left[\sum_{i=0}^{n-1} 2 + 2 + 1\right] + 1 = 3 + 5n$$

$$O\big(Tiempo(if - else)\big) = O(1) + \max\big(O(1), O(n)\big) = O(n)$$

II.6 El pseudocódigo

A continuación presentamos algunos ejercicios en donde se calcula el orden de complejidad del algoritmo en base al tiempo de ejecución en el peor caso: T(*n*).

Un algoritmo puede representarse con *pseudocódigo*. El *pseudocódigo* sirve para representar un algoritmo independientemente de un lenguaje de programación específico. Es decir, se representa el algoritmo genéricamente y éste puede implementarse después en cualquier lenguaje de programación, como se ilustra en la *Figura 2.13.*

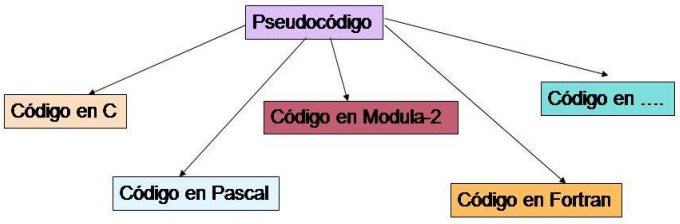


Figura 2.13: el pseudocódigo puede implementarse en cualquier lenguaje

En el pseudocódigo se indica el nombre del algoritmo, enseguida, entre paréntesis, los parámetros que recibe indicados con una flecha hacia abajo ↓, cuando se pasan por valor.

Cuando los parámetros se pasan por referencia se indican con dos flechas ↓↑. Después se indican las constantes (o conjunto vacío si no las tiene). Posteriormente las variables y finalmente las acciones. La asignación se representa con una flecha, y las instrucciones estructuradas, como el condicional y los ciclos, deben tener la indicación de donde comienzan y donde terminan.

Por ejemplo, sea el pseudocódigo del siguiente *Algoritmo1*:

```
Procedimiento Algoritmo1 (↓↑a ∈ Entero Vector[1..n] )
 Constantes
  ∅
 Variables
  temp,i,j ∈ Entero
 Acciones
  Para i ← 1..n-1 Hacer
   Para j ← n..i+1 Hacer
     Si (a[j-1]>a[j]) Entonces
     temp ← a[j-1]
     a[j-1] ← a[j]
       a[j] ← temp
  Fin Para
```

El pseudocódigo del Algoritmo1 se puede implementar en el lenguaje de programación que se elija, por ejemplo, en Modula-2:

```
Procedure Algoritmo1(VAR a:vector);
  Var i, j: CARDINAL;
      temp: INTEGER;
BEGIN
 FOR i:=1 TO n-1 DO
   FOR j:=n TO i+1 BY -1 DO
     IF(a[j-1]>a[j])THEN
       temp := a[j-1];
       a[j-1]:= a[j];
       a[j]:= temp;
     END
   END
 END
END Algoritmo1
```

O bien en lenguaje C:

```
void Algoritmo1(int *a, int n){
 int temp;
 for(int i=0; i<= n-2; i++){
   for(int j=n-1;j>=i;j--){
     if(a[j-1]>a[j]){
       temp = a[j-1];
       a[j-1]= a[j];
       a[j]= temp;
     };
   };
 };
};
```

II.7 Ejercicios del cálculo de T(*n*)

Para calcular el tiempo de ejecución *T(n)* puede usarse directamente el pseudocódigo o bien, implementar el pseudocódigo en el lenguaje de programación con el que se esté más familiarizado y posteriormente hacer el cálculo.

Ejercicio 1.- Calcular *T(n)* para el siguiente algoritmo:

```
Si (N MOD 2 = 0) Entonces
  Para I ← 1..N Hacer
    X ← X + 1
  Fin Para
Fin Si
```

El algoritmo anterior implementado en lenguaje C++, es:

```
if(n%2 == 0){
  for(int i=1; i<= n; i++)
     x++;
}
```

Podemos calcular el orden de complejidad de este algoritmo calculando el número de operaciones elementales, y así tenemos que:

$$T(n) = 2 + \left(1 + 1 + \left(\sum_{i=1}^{n} 2 + 2 + 1\right)\right) = 4 + 5n \in \mathbf{O}(n)$$

Nótese que como el polinomio T(*n*) es de primer grado, por lo que el algoritmo es **O**(*n*).

Ejercicio 2.- Calcular *T(n)* para el algoritmo "intercambiar":

```
Procedimiento Intercambiar (↓↑x, ↓↑y ∈ Entero)
  Constantes
    ∅
  Variables
    aux ∈ Entero
  Acciones
    aux ⟵ x
    x ⟵ y
    y ⟵ aux
```

El algoritmo anterior implementado en lenguaje C++, es:

```
void intercambiar (int *x, int *y){
   int *aux;
   *aux = *x;
   *x = *y;
   *y = *aux;
}
```

En este caso se trata de una función (subrutina) la cual recibe y regresa dos parámetros, entonces hay que tomar en cuenta la *regla 5*:

$$T(n) = 1 + Tiempo(P1) + Tiempo(P2) + ... + Tiempo(Pn) + Tiempo(F).$$

Como solo se tienen dos parámetros y *Tiempo(F)* = 3, tenemos que

$$Tiempo(Intercambiar) = 1 + 1+1+3 = 6 \in \mathbf{O}(1)$$

Ejercicio 3.- Calcular *T(n)* para el algoritmo *Sumatoria*:

```
Procedimiento Sumatoria (n ∈ Entero)
  Constantes
    ∅
  Variables
    suma ∈ Real
  Acciones
    suma ⟵ 0
    Para i⟵ 1...n hacer
      suma ⟵ suma + i
    Fin Para
    regresar suma
```

El algoritmo anterior implementado en lenguaje C++, es:

```
float Sumatoria (int n){
   float suma=0;
   for( int i=1; i<=n; i++ )
      suma = suma+i;
    return suma;
}
```

Podemos calcular el orden de complejidad de este algoritmo calculando el número de operaciones elementales y así tenemos que:

$$T(n) = 1 + 1 + 1 + 1 + 1 + \left[\sum_{i=1}^{n} 2 + 2 + 1\right] + 1 = 6 + 5n \ \in \mathbf{O}(n)$$

Llamado Parámetro suma=0 Regreso

Nótese que el polinomio *T(n)* es de primer grado, el algoritmo es **O**(*n*).

II.8 Análisis de la complejidad de algoritmos recursivos.

Un algoritmo ***recursivo*** consiste en obtener la solución total de un problema a partir de la solución de casos más sencillos de ese mismo problema. Estos casos se resuelven invocando al mismo algoritmo, el problema se simplifica hasta llegar al caso más sencillo de todos llamado *caso base*, cuya solución se da por definición. El aspecto de un algoritmo recursivo es el siguiente:

```
ALGORITMO Recursivo(caso)
INICIO
  if caso = casoBase
     regresa solucionPorDefinicion
   else
     .....
     s1 = Recursivo(casoReducido1)
     s2 = Recursivo(casoReducido2)
     ...
     regresa SolucionDelCaso(s1,s2,...)
FIN
```

Por ejemplo, usando un algoritmo recursivo para obtener el factorial de un número natural tenemos la siguiente expresión, a la cual se le llama *ecuación de recurrencia*, porque la función está dada en términos de la propia función.

$$factorial(n) = \begin{cases} 1, & si\ n = 0 \\ n * factorial(n-1), & si\ n > 0 \end{cases}$$

Programando en C/C++ la ecuación de recurrencia anterior tenemos que:

```
long int factorial( long int n ) {
   long int s1;
   if  (n <= 1)
      return 1;
   else {
      s1= factorial(n-1);
      return n*s1;
   };
};
```

La función se llama a sí misma.

El tiempo de ejecución de un algoritmo recursivo se expresa por medio de una *ecuación de recurrencia*, la cual es una función $T(n)$. En las *recurrencias* se separa el tiempo de ejecución del caso básico del tiempo de ejecución del caso recursivo. En este último caso, se utiliza la misma función $T(n)$ para representar el tiempo de ejecución de la llamada recursiva, por ejemplo:

En el caso del factorial, si consideramos que el caso base tiene un tiempo de ejecución k_1 y que el resto de los casos tienen un tiempo de ejecución $T(n\text{-}1) + k_2$ podemos substituir n por n-1 para obtener $T(n\text{-}1)$. Así, el factorial queda expresado con la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} k_1 & n = 0 \\ T(n\text{-}1) + k_2 & n > 0 \end{cases}$$

II.9 Ejercicios del cálculo de $T(n)$ con solución de recurrencias por sustitución

Resolver ecuaciones de recurrencia consiste en encontrar una expresión no recursiva de T, y por lo general no es una labor fácil. Los tres métodos más utilizados para resolver ecuaciones de recurrencia son: inducción matemática, resolución de la ecuación

característica y la solución por sustitución. En esta sección estudiaremos ejemplos en los que se emplea la solución de recurrencias por sustitución.

Ejercicio 1.- Dada la ecuación de recurrencia de la sección anterior, correspondiente al algoritmo que obtiene el factorial de manera recursiva, encontrar su solución y su cota superior de complejidad (notación **O**).

Solución:

Primero supondremos que, además del caso base, el algoritmo se llama a sí mismo en una ocasión. Para encontrar el tiempo de ejecución de $T(n\text{-}1)$, partimos de la expresión original:

$$T(n) = T(n\text{-}1) + k_2 \quad (1)$$

La expresión de $T(n)$ para este caso se obtiene sustituyendo n por n-1 en la ecuación 1:

$$T(n\text{-}1) = T(n\text{-}1\text{-}1) + k_2 = T(\text{n-}2) + k_2 \quad (2)$$

Y posteriormente substituyendo la ecuación 2 en la ecuación 1:

$$T(n) = T(n\text{-}2) + k_2 + k_2 = T(n\text{-}2) + 2k_2 \quad (3)$$

Para encontrar el tiempo de ejecución de $T(n\text{-}2)$, partimos nuevamente de la ecuación 1, substituyendo n por n-2:

$$T(n\text{-}2) = T(n\text{-}2\text{-}1) + k_2 = T(\text{n-}3) + k_2 \quad (4)$$

Substituyendo la ecuación 4 en la ecuación 3 se obtiene:

$$T(n) = T(n\text{-}3) + k_2 + 2\,k_2 = T(n\text{-}3) + 3k_2 \quad (5)$$

Podemos inferir que el tiempo de ejecución $T(n)$ está dado por:

$$T(n) = T(n\text{-}i) + ik_2 \quad (6)$$

En el *caso base* se cumple que n-i = 1 de donde obtenemos $i = n$-1 que, sustituyéndolo en la ecuación 6 se obtiene:

$$T(n) = T(1) + (n\text{-}1)k_2 \quad \text{para } n > 0 \quad (7)$$

Como se trata del caso base, entonces $T(1) = k_1$

$$T(n) = k_1 + (n\text{-}1)k_2 \quad \text{para } n > 0 \quad (8)$$

En esta última expresión hemos encontrado una expresión no recursiva de $T(n)$ la cuál es **O**(n).

Ejercicio 2.- Si tenemos la siguiente función recursiva

```
int Recursiva1( int n ) {
   if (n <= 1)
     return (1);
   else
     return ( 2*Recursiva1(n/2) );
}
```

Encontrar la solución a su ecuación de recurrencia y la cota superior de la complejidad de esta función.

Solución: La ecuación de recurrencia para la función `Recursiva1` es:

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

Para encontrar el tiempo de ejecución de $T(n/2)$, partimos de la expresión original:

$T(n) = T(n/2) + 1$ (1)

La expresión de $T(n)$ para este caso se obtiene sustituyendo n por $n/2$ en la ecuación 1:

$T(n/2) = T((n/2)/2) + 1 = T(n/2^2) + 1$ (2)

Y posteriormente substituyendo la ecuación 2 en la ecuación 1:

$T(n) = T(n/2^2) + 1 + 1 = T(n/2^2) + 2$ (3)

Siguiendo este mismo procedimiento encontramos lo siguiente:

$T(n) = T(n/2^3) + 1 + 1 + 1 = T(n/2^3) + 3$

$= T(n/2^4) + 1 + 1 + 1 + 1 = T(n/2^4) + 4$

$= \ldots$

Podemos inferir que el tiempo de ejecución $T(n)$ está dado por:

$T(n) = T(n/2^i) + i$ (4)

En el *caso base* se cumple que $n/2^i = 1$ de donde obtenemos

$2^i = n$

$i = \log_2 n$

que, sustituyéndolo en la ecuación 4 se obtiene:

$T(n) = T(1) + \log_2 n$ para $n > 1$

$= 1 + \log_2 n$

Por lo tanto $T(n)$ es **O**(log n).

Ejercicio 3.- Si tenemos la siguiente función recursiva

```
int Recursiva2 (int n, int x){
   int i;
   if (n <= 1)
     return (1);
   else{
     for( i=1; i<=n; i++)
        x = x + 1;

     return( Recursiva2(n-1, x) );
   };
}
```

Encontrar la solución a su ecuación de recurrencia y la cota superior de la complejidad de esta función.

Solución: La ecuación de recurrencia para Recursiva2 es:

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

Para encontrar el tiempo de ejecución de $T(n\text{-}1)$, partimos de la expresión original:

$$T(n) = T(n\text{-}1) + n \quad (1)$$

La expresión de $T(n)$ para este caso se obtiene sustituyendo n por n-1 en la ecuación 1:

$$T(n\text{-}1) = T((n\text{-}1) \text{-}1) + (\text{n-}1) = T(n\text{-}2) + n - 1 \quad (2)$$

Y posteriormente substituyendo la ecuación 2 en la ecuación 1:

$$T(n) = (T(n\text{-}2) + (n - 1)) + n = \quad (3)$$

La expresión de $T(n)$ para n-2 se obtiene sustituyendo n por n-2 en la ecuación 1:

$$T(n\text{-}2) = T((n\text{-}2) \text{-}1) + (\text{n-}2) = T(n\text{-}3) + n - 2 \quad (4)$$

Y posteriormente substituyendo la ecuación 4 en la ecuación 3:

$$T(n) = (T(n\text{-}2) + (n - 1)) + n = \quad (5)$$

Siguiendo este mismo procedimiento podemos inferir lo siguiente:

$T(n) = (T(n\text{-}3) + (n - 2)) + (n\text{-}1) + n = \ldots.$

$= T(n\text{-}i) + \sum_{k=0}^{i-1}(n-k)$

En el *caso base* se cumple que $i = n\text{-}1$ de donde obtenemos

$$T(n) = T\,(n-i) + \sum_{k=0}^{i-1}(n-k)$$

$$= T\big(n-(n-1)\big) + \sum_{k=0}^{n-2}(n-k)$$

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

Por inducción matemática se sabe que:

$$\sum_{k=1}^{n+1} k = \sum_{k=1}^{n} k + (n+1) = \frac{n(n+1)}{2} + (n+1) = \frac{(n+1)(n+2)}{2}$$

Entonces:

$$T(n) = T(1) + \sum_{k=0}^{n-2} n - \sum_{k=0}^{n-2} k = 1 + n(n-1) - \frac{(n-2)(n-1)}{2}$$

Podemos observar que la ecuación anterior es una expresión no recursiva de $T(n)$ y además es $\mathbf{O}(n^2)$.

Ejercicio 4.- Si tenemos la siguiente función recursiva

```
int Rec3 (int n){
   if (n <= 1)
     return (1);
   else
     return ( Rec3(n-1) + Rec3(n-1) );
}
```

Encontrar la solución a su ecuación de recurrencia y la cota superior de la complejidad de esta función.

Solución: La ecuación de recurrencia para Rec3 es:

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

Para encontrar el tiempo de ejecución de $T(n\text{-}1)$, partimos de la expresión original:

$$T(n) = 2T(n\text{-}1) + 1 \qquad (1)$$

La expresión de $T(n)$ para este caso se obtiene sustituyendo n por n-1 en la ecuación 1:

$$T(n\text{-}1) = 2T((n\text{-}1)\text{-}1) + 1 = 2T(n\text{-}2) + 1 \qquad (2)$$

Y posteriormente substituyendo la ecuación 2 en la ecuación 1:

$$T(n) = 2(2T(n\text{-}2) + 1) + 1 = 4T(n\text{-}2) + 3 \qquad (3)$$

La expresión de $T(n)$ para este caso se obtiene sustituyendo n por n-2 en la ecuación 1:

$$T(n\text{-}2) = 2T((n\text{-}2)\text{-}1) + 1 = 2T(n\text{-}3) + 1 \qquad (4)$$

Y posteriormente substituyendo la ecuación 4 en la ecuación 3:

$$T(n) = 4(2T(n\text{-}3) + 1) + 3 = 8T(n\text{-}3) + 7 \qquad (5)$$

Siguiendo este mismo procedimiento encontramos lo siguiente:

$$\begin{aligned} T(n) &= 2T(n\text{-}1) + 1 \\ &= 4T(n\text{-}2) + 3 \\ &= 8T(n\text{-}3) + 7 \\ &= \ldots \end{aligned}$$

Podemos inferir que el tiempo de ejecución $T(n)$ está dado por:

$$T(n) = 2^i\, T(n\text{-}i) + (2^i - 1) \qquad \text{ec. 6}$$

En el *caso base* se cumple que $n\text{-}i = 1$ por lo que

$$i = n\text{-}1$$

que, sustituyéndolo en la ecuación 6 se obtiene:

$$\begin{aligned} T(n) &= 2^{n-1}\, T(1) + (2^{n-1} - 1) \\ &= 2^{n-1} + 2^{n-1} - 1 = 2 \bullet 2^{n-1} - 1 = 2^n - 1 \end{aligned}$$

Por lo tanto $T(n)$ es $\mathbf{O}(2^n)$.

II.10 Resumen del capítulo

En el capítulo II vimos que las *medidas asintóticas* (o *cotas de complejidad*) permiten analizar qué tan rápido crece el tiempo de ejecución de un algoritmo, cuando crece el tamaño de los datos de entrada. Existen tres cotas de complejidad; la cota superior, que se expresa mediante la notación **O**(o mayúscula), la cota inferior, que se expresa mediante la notación **Ω** (omega mayúscula), y la cota ajustada asintótica, expresada con la notación **θ** (theta mayúscula). Hicimos énfasis en la notación **O** y en sus propiedades.

Utilizamos la notación **O** para identificar si un algoritmo tiene un orden de complejidad mayor o menor que otro. Determinamos que un algoritmo es más eficiente mientras menor sea su orden de complejidad. Presentamos los tipos de complejidad más comunes expresados con la notación **O**. También estudiamos la representación de algoritmos mediante pseudocódigo, como hacer el cálculo de operaciones elementales T(*n*) en un algoritmo no recursivo. Finalmente calculamos T(*n*) para los algoritmos recursivos.

En el siguiente capítulo pondremos en práctica los conocimientos adquiridos sobre la notación **O** para analizar la eficiencia de los principales algoritmos directos de ordenamiento.

Capítulo III Análisis de la eficiencia de algoritmos de ordenamiento

III.1 Definición y clasificación de los algoritmos de ordenamiento

Los problemas de *ordenamiento* juegan un papel muy importante en las ciencias de la computación. Este tipo de problemas consisten en poner en orden, ya sea ascendente o descendente un conjunto de *n* objetos.

Ordenamiento es el problema que se presenta cuando tenemos un conjunto de *n* elementos $\{a_1, a_2, \ldots, a_n\}$ y queremos obtener el mismo conjunto pero de tal forma que:

$$a_i \leq a_{i+1} \qquad i = 1, 2, \ldots, n$$

O bien que:

$$a_i \geq a_{i+1} \qquad i = 1, 2, \ldots, n$$

En la vida real se presentan conjuntos de datos que pueden ser bastante complejos. Sin embargo, estos problemas de ordenamiento se pueden reducir al ordenamiento de números enteros si utilizamos las claves de los registros o bien índices. Por lo anterior es importante estudiar el problema de ordenamiento de números enteros. Existen numerosos algoritmos para resolver el problema de ordenamiento y podemos clasificarlos en *directos, indirectos*, y *otros*.

Entre los algoritmos de ordenamiento *directos* tenemos:

- Ordenamiento por intercambio
- Ordenamiento por selección
- Ordenamiento por inserción

- Ordenamiento por burbuja

Entre los algoritmos de ordenamiento *indirectos* tenemos:

- Ordenamiento por mezcla (Mergesort)
- Ordenamiento rápido (Quicksort)
- Ordenamiento por Montículos (Heap sort)

También existen *otros* algoritmos de ordenamiento, como por ejemplo:

- Ordenamiento por Incrementos (Shell sort).
- Ordenamiento por Cubetas (Bin sort).
- Ordenamiento por Resíduos (Radix).

En este capítulo analizaremos los métodos de ordenamiento directo de la burbuja, de selección y de inserción. Para todos estos casos, $T(n)$ depende no solo del tamaño de los datos de entrada, sino también de lo ordenados que éstos se encuentren (ver *Figura 3.1*).

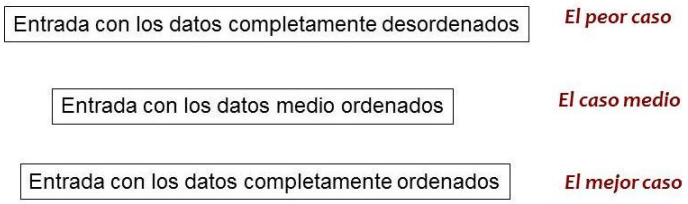


Figura 3.1: $T(n)$ depende de lo ordenados que estén los datos de entrada

Cuando los datos están completamente desordenados ninguno se encuentra en su lugar y cuando están completamente ordenados, todos están en su lugar correspondiente, ya sea de mayor a menor o viceversa.

III.2 El método de la Burbuja

El algoritmo de ordenamiento de la burbuja es un método muy conocido, sencillo y fácil de implementar. El nombre *ordenamiento por burbuja* se deriva del hecho de que los valores más pequeños en el arreglo flotan o suben hacia la parte inicial (primeras posiciones) del arreglo, mientras que los valores más grandes caen hacia la parte final (últimas posiciones) del arreglo. A continuación tenemos el pseudocódigo del método de la burbuja.

```
Procedimiento Burbuja (↓↑A ∈ Entero Vector[1..N])
 Constantes
  ∅
 Variables
  i, j ∈ Entero
 Acciones
  Para i ← 1 .. N-1 Hacer
   Para j ← N .. i + 1 Hacer
    Si A[j - 1] > A[j] Entonces
     Intercambiar(A[j], A[j-1])
    Fin Si
   Fin Para
  Fin Para
```

Podemos observar que `Burbuja` hace uso de la función Intercambiar, cuyo pseudocódigo se presenta a continuación.

```
Procedimiento Intercambiar (↓↑x, ↓↑y ∈ Entero)
 Constantes
  ∅
 Variables
  aux ∈ Entero
 Acciones
  aux ← x
  x ← y
  y ← aux
```

El código en C++ de Intercambiar es el siguiente:

```
void Intercambiar (int *x, int *y){
 int *aux;
 *aux = *x;
 *x = *y;
 *y = *aux;
}
```

Para calcular el número de operaciones elementales del llamado a la función `Intercambiar` tenemos que:

$$T(n) = 1 + Tiempo(P1) + Tiempo(P2) + ... + Tiempo(Pn) + Tiempo(F)$$

$$T(n)= 1 + 1+1+3 = 6 \in \mathbf{O}(1)$$

La función `Intercambiar` es $\mathbf{O}(1)$ y la utilizaremos también en los métodos de inserción y selección.

Ahora, el código en C de `Burbuja` es el siguiente:

```
void Burbuja (int *A, int n){
 for(int i=0;i<=n-2;i++){
   for(int j=n-1;j>=i+1;j--){
     if(A[j-1]>A[j])
       Intercambiar(A[j], A[j-1]);
   }
 }
}
```

Para obtener el número de operaciones elementales en el peor caso de `Burbuja`:

$$T_{burbuja}(n) = 1 + 2 + \left[1 + 2 + \sum_{i=0}^{n-1}\left(2 + 1 + \sum_{j=i}^{n-1}(4 + 4 + 1 + 2)\right) + 2 + 2\right]$$

$$= 6 + \sum_{i=0}^{n-1}\left(3 + \sum_{j=i}^{n-1}(11) + 4\right)$$

$$= 6 + \sum_{i=0}^{n-1}(7 + (n-i)11) = 6 + (7 + 11n)n - 11\sum_{i=0}^{n-1} i$$

$$T_{burbuja}(n) = 6 + 7n + 11n^2 - \frac{11(n-1)n}{2} = \frac{11n^2}{2} + \frac{25n}{2} + 6 \quad \in O(n^2)$$

El orden de complejidad del método de la burbuja es $\mathbf{O}(n^2)$. Este método es uno de los más sencillos sin embargo es de los más ineficientes, por lo que es de los menos utilizados.

Recordemos el *Principio de invarianza*, el cual nos dice que el tiempo de ejecución de dos implementaciones distintas, I_1 e I_2, de un algoritmo dado, no va a diferir más que en una constante multiplicativa, es decir:

$$T_1(n) \leq cT_2(n)$$

Sea I_1 la siguiente implementación del método de la burbuja:

```
void BurbujaI1(int *A, int n){
 for(int i=0;i<=n-2;i++){
   for(int j=n-1;j>=i+1;j--){
     if(A[j-1]>A[j])
       Intercambiar(A[j], A[j-1]);
   }
 }
}
```

Y sea I_2 la siguiente implementación del método de la burbuja:

```
void BurbujaI2(int *A, int n){
 for(int i=0;i<=n-1;i++){
   for(int j=0;j<n-1-i;j++){
     if(A[j]>A[j+1])
       Intercambiar(A[j], A[j+1]);
   }
 }
}
```

Con I_1, que es $\mathbf{O}(n^2)$, el elemento más pequeño se desplaza hasta el principio, como se muestra en la *Figura 3.2*.

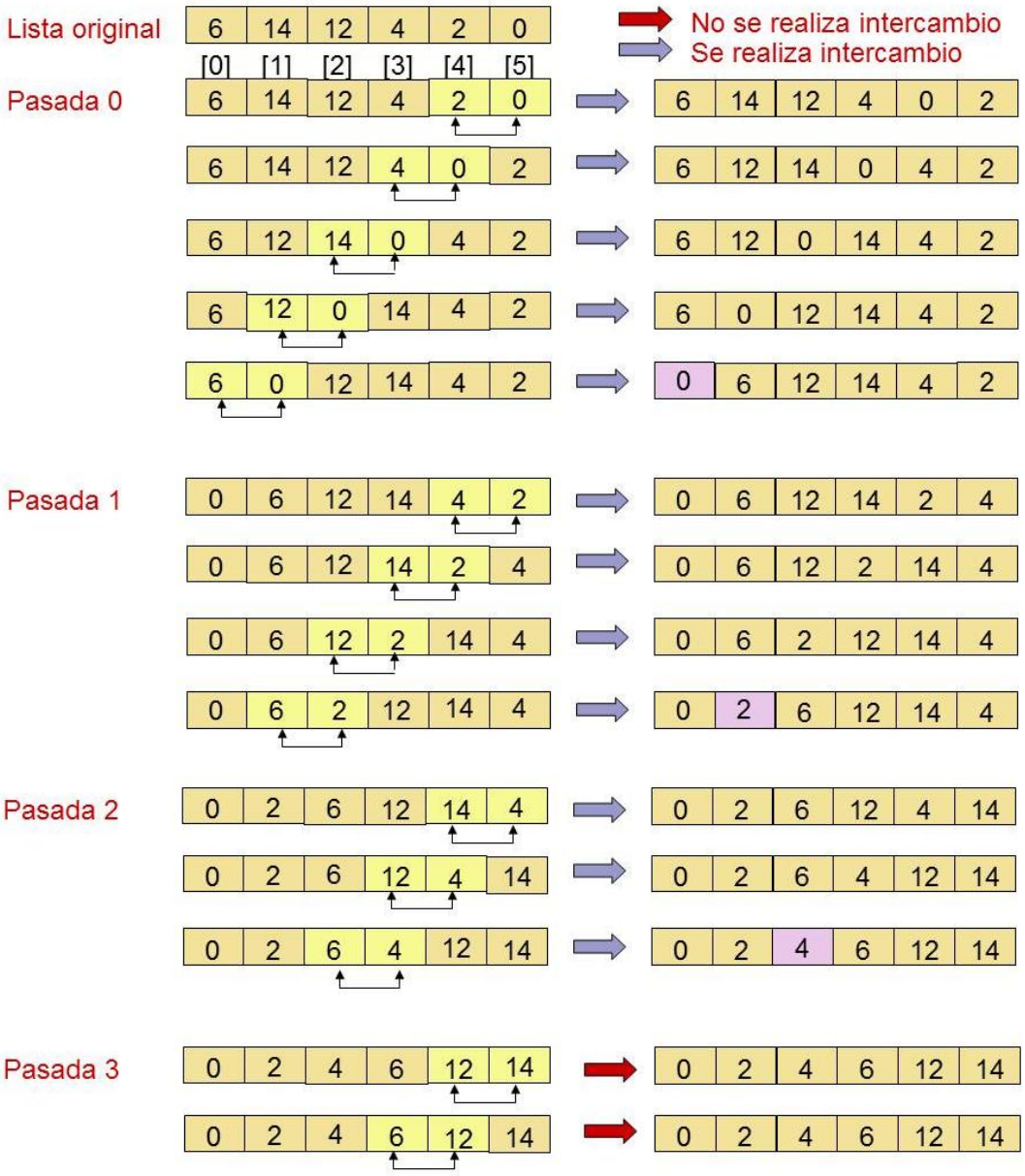


Figura 3.2: Ordenamiento por burbuja *Implementación 1*

Mientras que con I_2, que también es $\mathbf{O}(n^2)$, el elemento más grande se desplaza hasta el final, como se muestra en la *Figura 3.3*.

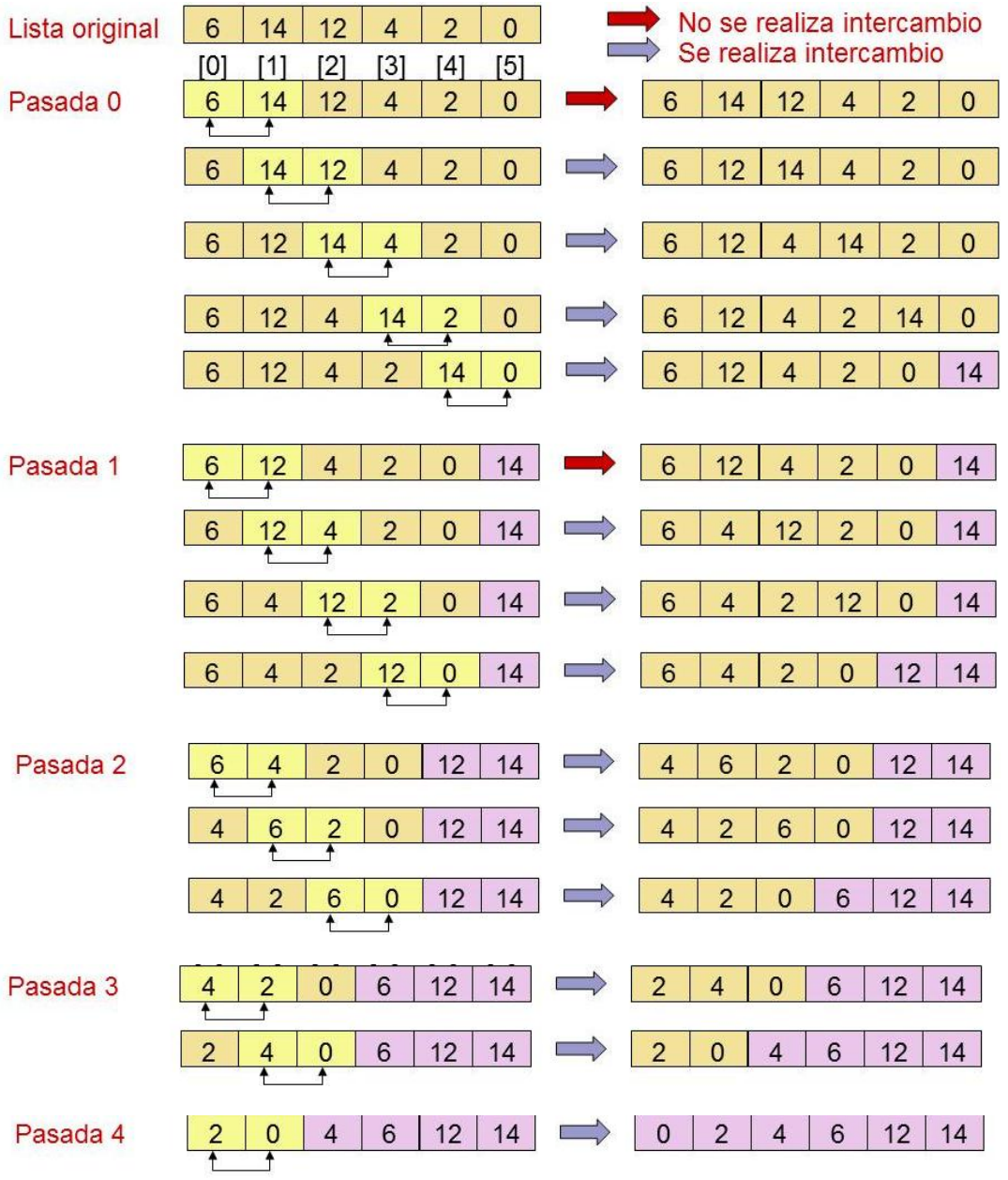


Figura 3.3: Ordenamiento por burbuja *Implementación 2*

En las *Figuras 3.2* y *3.3* se puede observar que los datos iniciales representan al peor caso, ya que ninguno de ellos se encuentra en su lugar.

Si analizamos las implementaciones I_1 e I_2 , como se muestra en la *Figura 3.4*, tenemos que las dos son del mismo orden.

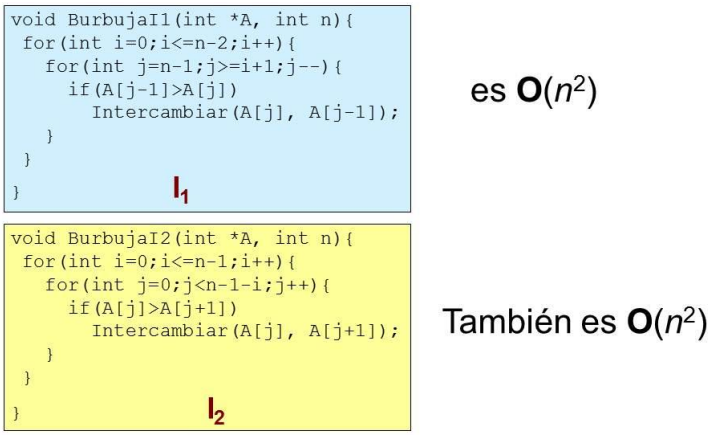


Figura 3.4: Tiempo de ejecución de las dos implementaciones del ordenamiento por burbuja

Por el principio de invarianza, si I_1 es $\mathbf{O}(n^2)$ entonces I_2 también es $\mathbf{O}(n^2)$. Es decir, el tiempo para ejecutar un algoritmo depende del algoritmo y no de la implementación del algoritmo.

III.3 El método de Inserción

El algoritmo de ordenamiento por inserción ordena un arreglo de *n* elementos en orden ascendente, insertando directamente cada elemento de la lista en la posición adecuada y recorriendo hacia la derecha (de [i] a [i+1]) los elementos restantes de la lista que son mayores al elemento insertado. Sea "a" el arreglo con los elementos a ordenar, el algoritmo comienza considerando una lista inicial compuesta por un solo elemento: a[0].

A continuación se selecciona a[1] y si éste es menor que a[0], entonces a[1] se inserta en la posición [0] y este último se recorre una posición hacia la derecha, de lo contrario a[1] se inserta en la posición [1].

El proceso continúa seleccionando consecutivamente cada elemento de la sublista a[i], a[i+1], ..., a[n-1], buscando la posición correcta para su inserción en la sublista a[i-1], a[1], ..., a[0] y recorriendo hacia la derecha una posición todos los elementos mayores al elemento insertado. En la *Figura 3.5* se muestra un ejemplo del algoritmo de ordenamiento por inserción.

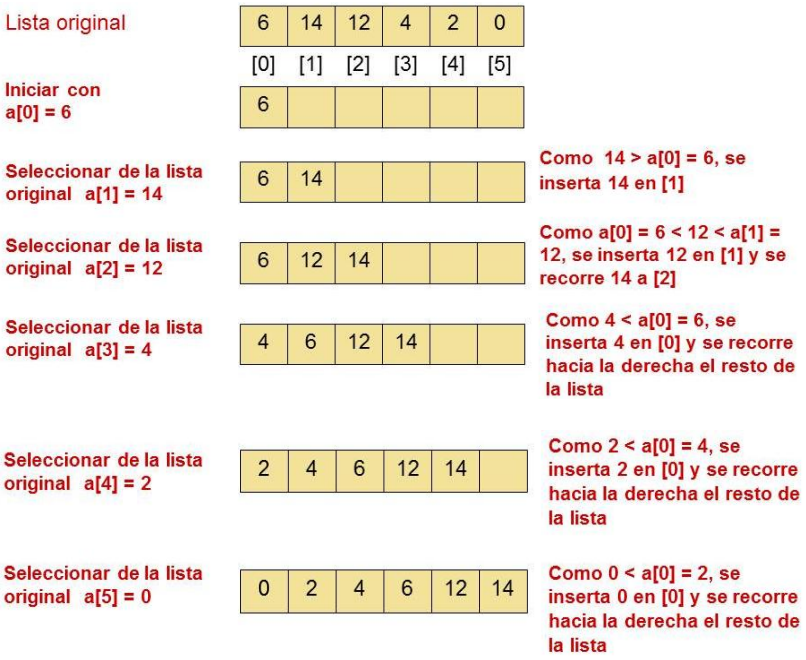


Figura 3.5: Ejemplo de ordenamiento por inserción
(Ilustración del material de clase del Dr. Pedro Pablo González Pérez)

A continuación tenemos una función que ordena por inserción, recibe como parámetro el arreglo `A` y su tamaño *n*.

```
void Insercion (int *A, int n){
  int j,temp;
  for( int i= 1; i < n; i++){
   temp = A[i];
   for( j = i-1; j>=0 && temp < A[j]; j-- )
     A[j+1] = A[j];
   A[j+1] = temp;
  };
};
```

Lo que se hace normalmente es encontrar como están los ciclos `for`. Si es un ciclo sencillo, el algoritmo es de orden $\mathbf{O}(n)$, pero si existe un ciclo anidado dentro de otro ciclo, entonces el algoritmo es $\mathbf{O}(n^2)$. Como podemos apreciar en la función `Insercion` hay un `for` anidado, por lo que el algoritmo es $\mathbf{O}(n^2)$. Siguiendo este criterio, un algoritmo que tiene 3 ciclos anidados es $\mathbf{O}(n^3)$.

En la siguiente sección analizaremos el orden de complejidad del algoritmo de ordenamiento por selección.

III.4 El método de Selección

El algoritmo ejecuta varias pasadas (o iteraciones) y en cada una de ellas remplaza el elemento del arreglo que está en la posición a substituir por el mínimo elemento encontrado. La primera posición a substituir es a[0] y se analizará a[0], a[1], a[2], …, a[n-1]. La segunda posición a substituir es a[1] y se analizará a[1], a[2], a[3], … a[n-1], y así sucesivamente. En un arreglo de n elementos se ejecutan n-1 iteraciones.

En la *Figura 3.6* se muestra un ejemplo del método de ordenamiento por selección.

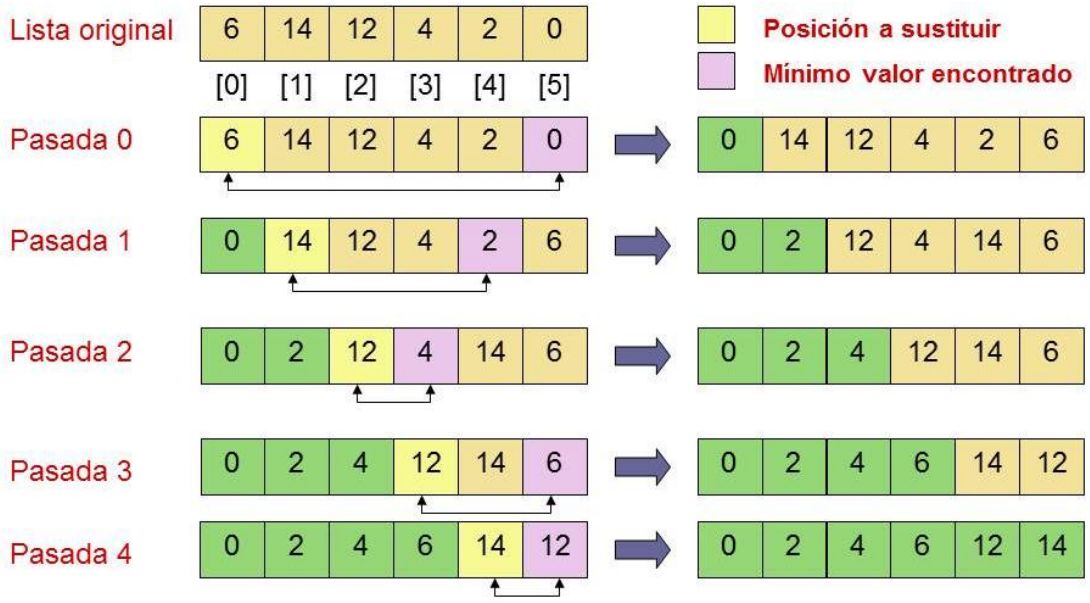


Figura 3.6: ejemplo de ordenamiento por selección
(Ilustración del material de clase del Dr. Pedro Pablo González Pérez)

El algoritmo de ordenamiento por selección tiene un ciclo anidado, como se puede observar en la siguiente implementación:

```
void Seleccion (int *a, int n){

int menor, temp, indice;

for( int i= 0; i < n-1; i++){
  menor = a[i];
  indice = i;
  for( j = i+1; j < n; j++ ){
    if( menor > a[j]){
      menor = a[j];
      indice = j;
    };
   };
  temp = a[i];
  a[i] = menor;
  a[indice] = temp;
};
```

Por lo tanto el algoritmo de selección es $\mathbf{O}(n^2)$.

Los algoritmos de ordenamiento por inserción y selección requieren un tiempo cuadrático en el peor caso. Ambos son excelentes cuando el número de datos a ordenar *n* es pequeño. Existen otros algoritmos de ordenamiento que son más eficientes cuando *n* es grande, por ejemplo el ordenamiento por mezcla o fusión (*mergesort*) y el algoritmo de ordenamiento rápido (*quicksort*) los cuales se basan en el paradigma divide y vencerás, que se estudia en el siguiente capítulo.

III.5 Hashing

El *hashing* (tabla de dispersión) es una técnica que tiene diferentes usos, en esta sección estudiaremos el uso del hashing para el ordenamiento y búsqueda de datos.

Existen dos técnicas convencionales para almacenar los datos, el *almacenamiento en memoria estática* y el *almacenamiento en memoria dinámica*.

El *almacenamiento en memoria estática* se hace a través de los arreglos. La ventaja de este tipo de almacenamiento es que el tiempo de búsqueda es mínimo, sin embargo, si los elementos son más que los espacios disponibles en el arreglo, entonces, es imposible almacenarlos, y si, por el contrario, son menos que el tamaño del arreglo, entonces se desperdicia el espacio de la memoria.

El *almacenamiento en memoria dinámica* funciona utilizando la *memoria heap*. La *memoria heap* también llamada memoria dinámica o pedacería de memoria es un montón de pedazos de memoria que han sido "liberados" por los programas que ya no la necesitan. El sistema operativo tiene un *administrador de memoria* que es un conjunto de programas

para administrar esta pedacería y asignar pedazos a los programas que soliciten memoria. Alojar y liberar memoria dinámicamente hace más eficiente su uso, y para tener acceso a la memoria *heap* se utilizan los *apuntadores*.

Recordemos lo que es un apuntador. Un apuntador es un dato que indica la posición de otro dato. Los apuntadores se representan gráficamente con flechas. Físicamente, los apuntadores son direcciones de memoria en la computadora, y se guardan en variables de tipo apuntador. En la *Figura 3.7* tenemos la variable `x` que es de tipo entero y guarda el número 1024 y la variable `y` que también es un entero y guarda el número 12. Podemos observar que la variable `p` es de tipo apuntador a entero y guarda una dirección (representada por la flecha) que apunta hacia la posición del dato contenido en `x`.

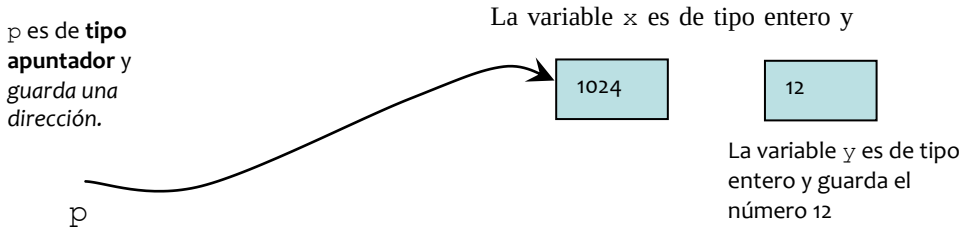


Figura 3.7: Concepto de apuntador

Los apuntadores sirven para crear *estructuras dinámicas*. Las *listas ligadas* son un ejemplo de estructura dinámica.

Una lista ligada es un conjunto de elementos *ligados* o encadenados mediante *ligas* o enlaces, como se muestra en la *Figura 3.8*. Las *listas ligadas* son "cajas" con información que se alojan en la memoria dinámica, cada una de estas "cajas" se crea conforme se va necesitando y contiene un *elemento* con información y una *liga* o *enlace* que le permitirá unirla a la siguiente caja. Cada vez que se aloja una nueva "caja" se liga a la caja anterior, de tal forma que se puede recorrer la lista siguiendo las ligas que unen una caja con otra. Las *listas ligadas* permiten trabajar con muchos o con muy pocos elementos usando solamente la memoria necesaria.

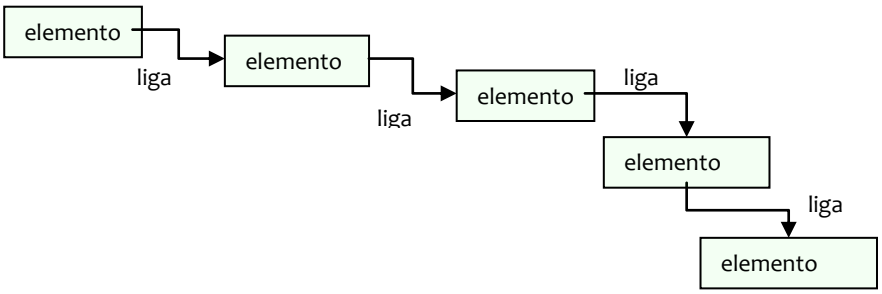


Figura 3.8: Listas ligadas

La ventaja de las listas ligadas es que se utiliza exactamente la cantidad de memoria necesaria, lo que es muy útil cuando la cantidad de datos a guardar es muy variable, sin embargo, el tiempo de búsqueda para encontrar un elemento dentro de la lista puede llegar a ser muy grande, ya que para buscar un elemento, hay recorrer la lista desde el principio cada vez. El peor caso se tiene cuando la lista ligada es enorme y el elemento buscado es el del final de la lista.

La técnica del hashing se utiliza para combinar las ventajas de la memoria dinámica con las de la memoria estática. Es una combinación de arreglos con listas ligadas.

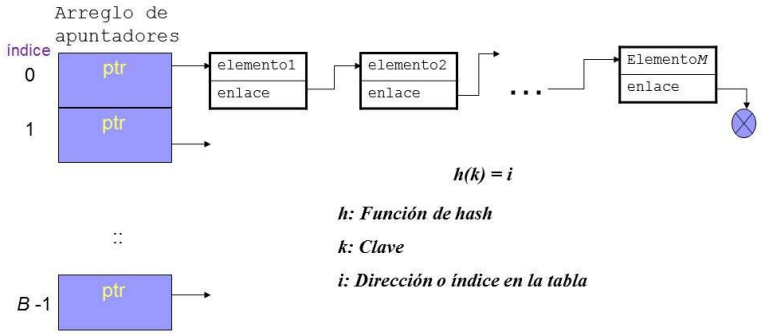


Figura 3.9: Hashing: combinación de listas ligadas con arreglo

El hashing combina mediante una función de dispersión, la *memoria estática* con la *memoria dinámica.* Como se ilustra en la *Figura 3.9.*

Función hash o de dispersión.- Es una función que mapea la clave de un elemento a un índice de un arreglo llamado *tabla de dispersión.*

Cada elemento tiene una *clave*, y aplicando la *función hash* a esta clave se obtiene el índice *i* del arreglo o *tabla de dispersión* en donde se almacenará el elemento. Lo anterior se expresa como:

h(k) =i

La función de Hash determina en donde se encuentra el apuntador a la lista ligada en donde se guardará o buscará el elemento, como se muestra en la *Figura 3.10*.

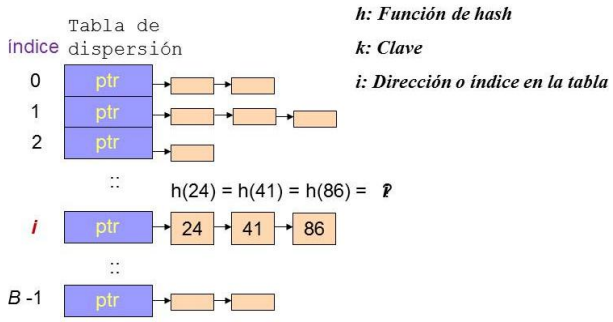
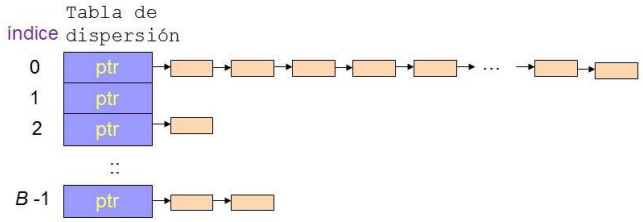
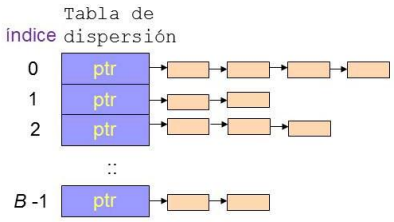


Figura 3.10: la función hash determina dónde se guardará o buscará el elemento

El hashing puede ser *abierto* o *cerrado*. En estas notas nos concentraremos en el *hashing abierto.* Una buena función de hash debe balancear la cantidad de elementos de los diferentes "pedazos" de lista. En las *figuras 3.11 a)* y *3.11 b)* se observa la diferencia entre elementos balanceados y no balanceados en una tabla de dispersión.



a): tabla de dispersión con elementos **no balanceados**



b): tabla de dispersión con elementos **balanceados**

Figura 3.11: diferencia entre elementos balanceados y no balanceados en una tabla de dispersión

En la *Figura 3.11 a)* se observa que el tiempo de búsqueda para el último elemento de la lista ligada que comienza en el índice 0 es mucho mayor que el de la del índice 2, que tiene

solo un elemento, mientras que en la tabla de la *Figura 3.11 b)* los tiempos de búsqueda en cada una de las listas de la tabla son similares.

Ejemplo.- A continuación se muestra una función hash para ordenar

Se parte de las siguientes premisas:

- Tenemos *N* elementos que se van a ordenar en *B* clases.
- Consideramos a los caracteres como enteros (su código ASCII).
- Suponemos que cada cadena tiene un máximo de 10 caracteres.

```
   int hashFunction(char *nombre){
  int suma=0;
  for(int i=0;i<10;i++){
   suma = suma + (int)nombre[i];

  return (suma%B);
}
```

suma Mod *B* es un entero entre o y *B*-1

El código anterior es una función que regresa un número entre 0 y B-1, este número es el índice dentro de la tabla de dispersión, la cual es de tamaño B. El valor de `hashFunction` depende del valor ASCII de cada uno de los caracteres dentro de la cadena.

El hashing también tiene otras importantes aplicaciones, por ejemplo:

- *Función de localización*: se aplica una función hash para asociar un número pequeño y manejable a un número grande.
- *Encriptación de datos*: se aplica una función hash a los datos y el resultado es el que se guarda en la base de datos, de esta manera, si alguien roba la base de datos no podrá interpretar su contenido si no conoce la función de hash.
- *Verificación de datos*: cuando se envían datos de un módulo a otro, por ejemplo, en el procesamiento paralelo, se envían los datos junto con el resultado de su hash, así se envían los datos y su versión corta. El módulo receptor conoce la función hash, así que puede probar si el dato recibido es el mismo que el que recibió encriptado con el hash, de esta forma se determina si la recepción de los datos es correcta.

III.6 Ejercicios

1.- Detalla los pasos del método de la burbuja para ordenar de menor a mayor los números 17, 5, 1, 32, 4, 0.

2.- Detalla los pasos del método de inserción para ordenar de menor a mayor los números 17, 5, 1, 32, 4, 0.

3.- Detalla los pasos del método de selección para ordenar de menor a mayor los números 17, 5, 1, 32, 4, 0.

4.- Haz la propuesta de un arreglo de números que se encuentre cerca del peor caso para ordenar de menor a mayor. Ordénalo por los tres métodos anteriores y cuenta el número de pasos que se llevó en cada uno ¿Qué diferencias encuentras entre cada uno los métodos?

5.- Haz la propuesta de un arreglo de números que, sin estar completamente ordenado, esté muy cerca del mejor caso. Ordénalo por los tres métodos anteriores y cuenta el número de pasos que se llevó en cada uno ¿Qué diferencias encuentras entre cada uno los métodos?

6.- ¿Cuál es el menor número de comparaciones que se requieren para encontrar el número más grande en una lista de n elementos?

Solución.- Se requiere de una sola comparación para encontrar el mayor de los primeros dos elementos. Si hubiera otro elemento más, se deberá comparar éste solamente contra el mayor de los anteriores incrementándose la cuenta de comparaciones en uno. Así, por cada elemento adicional se hace una comparación más. Para n elementos, los primeros 2 requieren 1 comparación, como restan n-2, hay que sumar una comparación por cada uno, por lo tanto el total es $1+n-2 = n-1$.

7.- Si un algoritmo de ordenamiento es $\mathbf{O}(n^2)$ y se tarda un segundo en ordenar 1000 elementos ¿Cuánto tardaría en ordenar 10,000 elementos?

Solución.-
$T(n) = K\, n^2$. Para n=1000, $T(1000) = K\,(1000)^2 = 1$ seg. De donde $K=10^{-6}$.
Ahora $T(10000) = 10^{-6}\,(10000)^2 = 10^2$ seg. El algoritmo tardará 100 veces más.

6.- Si un algoritmo de ordenamiento es $\mathbf{O}(n \log n)$ y se tarda un segundo en ordenar 1000 elementos ¿Cuánto tardaría en ordenar 10,000 elementos?

Solución.- $T(n) = K\, n \log n$. Para n=1000, $T(1000) = K\ 1000 \log 1000 = 1$ seg. De donde $K = 1/3000$. Ahora, $T(10000) = 1/3000\ (\ 10000 \log 10000\ ) \approx 13.333$ seg.

III.7 Resumen del capítulo

En este capítulo analizamos tres métodos de ordenamiento directo: el de la burbuja, el de selección y de inserción. Observamos que en los tres casos el orden del algoritmo es $\mathbf{O}(n^2)$. También vimos un ejemplo del principio de invarianza al analizar la complejidad de dos implementaciones diferentes del método de la burbuja. Finalmente estudiamos otro método de ordenamiento: el *Hashing*, el cual es una técnica que combina las ventajas de la memoria dinámica con las de la memoria estática.

En los siguientes cuatro capítulos estudiaremos diferentes paradigmas que han demostrado ser útiles en la práctica y que se utilizan para diseñar algoritmos, a estos se les llama *patrones de diseño*. Comenzaremos con el paradigma *divide y vencerás*.

Capítulo IV Divide y vencerás

IV.1 Uso de patrones de diseño

El diseño de un algoritmo es una labor creativa. Se requiere de experiencia y, con frecuencia, de varios intentos. Podría pensarse en la necesidad de un análisis por separado de cada problema en particular y crear desde cero un algoritmo que lo resuelva. Sin embargo, la experiencia ha demostrado que existen diversos patrones de diseño en los que el diseñador se puede basar para resolver ciertos grupos de problemas. Existen cinco técnicas de diseño muy conocidas, las cuales ya han demostrado su utilidad en varios problemas. El programador puede basar su diseño en alguna de estas técnicas y adaptarla a su problema particular. Las técnicas mencionadas son:

- Divide y vencerás.
- Algoritmos ávidos o voraces.
- Programación dinámica.
- Algoritmos de búsqueda con retroceso (backtracking).
- Ramificación y poda (acotación).

En este capítulo abordaremos la técnica de divide y vencerás. Las demás se describen en los capítulos subsecuentes.

IV.2 Definición del paradigma divide y vencerás

Divide y vencerás es una técnica de diseño de algoritmos que consiste en resolver un problema a partir de la solución de subproblemas del mismo tipo, pero de menor tamaño. Posteriormente se combinan los resultados parciales para obtener la solución del problema original. Si los subproblemas son todavía relativamente grandes, se aplica de nuevo esta técnica hasta alcanzar subproblemas lo suficientemente pequeños como para encontrar una solución directa. Para aplicar esta técnica se requiere que los subproblemas sean del mismo tipo que el problema original.

Si el problema x es de tamaño n y los tamaños de los subproblemas $x_1, x_2, \ldots, x_k$ son, respectivamente $n_1, n_2, \ldots, n_k$, describiremos el tiempo de ejecución T(n) del algoritmo divide y vencerás por medio de la siguiente ecuación de recurrencia [Martí et al., 2004]:

$$\mathrm{T}(n) = \begin{cases} g(n), & n \leq n_0 \\ \sum_{j=1}^{k} \mathrm{T}(n_j) + f(n), & n > n_0 \end{cases}$$

Donde T(n) es el tiempo del algoritmo divide y vencerás cuando el problema de entrada es de tamaño n, n_0 es el tamaño *umbral* que indica que ya no se va a subdividir el problema en partes más pequeñas, $g(n)$ es el tiempo del *método directo* y $f(n)$ es el tiempo de descomponer en subproblemas y combinar los resultados de éstos.

IV.3 Pasos de divide y vencerás

El paradigma de divide y vencerás puede definirse mediante los siguientes pasos planteados por Guerequeta y Vallecillo (2000):

1.- Plantear el problema de forma que pueda descomponerse en k subproblemas del mismo tipo, pero de menor tamaño.

Es decir, si el tamaño de la entrada es n, debemos dividir el problema en k subproblemas (donde $1 < k \leq n$) cada uno con una entrada cuyo tamaño aproximado es de n/k. A esta tarea se le conoce como *división*.

2.- Resolver independientemente todos los subproblemas, ya sea directamente si son elementales o bien de forma recursiva. El hecho de que el tamaño de los subproblemas sea estrictamente menor que el tamaño original del problema nos garantiza la convergencia hacia los casos elementales, también denominados casos *base*, en los que se utiliza un *método directo* para obtener la solución.

3.- Por último, *combinar* las soluciones obtenidas en el paso anterior para construir la solución del problema original.

El uso de este paradigma implica utilizar parámetros de acumulación para poder hacer esta combinación de soluciones.

IV.4 Características deseables

Es muy importante conseguir que los subproblemas sean independientes, es decir, que no exista traslape entre ellos. De lo contrario, el tiempo de ejecución de estos algoritmos será exponencial. Para aquellos problemas en los que la solución solo pueda construirse a partir de las soluciones de subproblemas entre los que necesariamente haya solapamiento, existe otra técnica de diseño más apropiada, llamada programación dinámica, la cual permite eliminar el problema de la complejidad exponencial debida a la repetición de cálculos. La programación dinámica se aborda en el capítulo VI.

Otra consideración importante a la hora de diseñar algoritmos divide y vencerás es el reparto de la carga entre los subproblemas, puesto que es importante que la división en subproblemas se haga de la forma más equilibrada posible. Para que valga la pena aplicar este método, las operaciones que descomponen el problema en otros más sencillos y las operaciones que combinan las soluciones parciales deben ser bastante eficientes. Además, el número de subproblemas que se generen no debe ser grande.

IV.5 Las torres de Hanoi

El problema de las torres de Hanoi consiste en pasar los discos, uno por uno (o uno a la vez), del poste A, al poste B con la condición de no poner un disco más grande sobre otro más pequeño. Se puede utilizar el poste C como auxiliar. Si intentamos describir la solución del problema para una torre de 4 o más discos, la descripción resultaría muy complicada, sin embargo, si aplicamos el enfoque de divide y vencerás, entonces la solución es mucho más clara. El problema de pasar los n discos más pequeños de A a B se reduce a lo siguiente: considerar el problema como dos problemas de tamaño n-1. Primero se mueven los n-1 discos más pequeños del poste A al C, dejando el n-esimo disco en el poste A. Se mueve este disco de A a B y, finalmente se mueven los n-1 discos más pequeños de C a B. El movimiento de los n-1 discos más pequeños se efectuará por medio de la aplicación recursiva del método, pero utilizando el poste no ocupado como auxiliar.

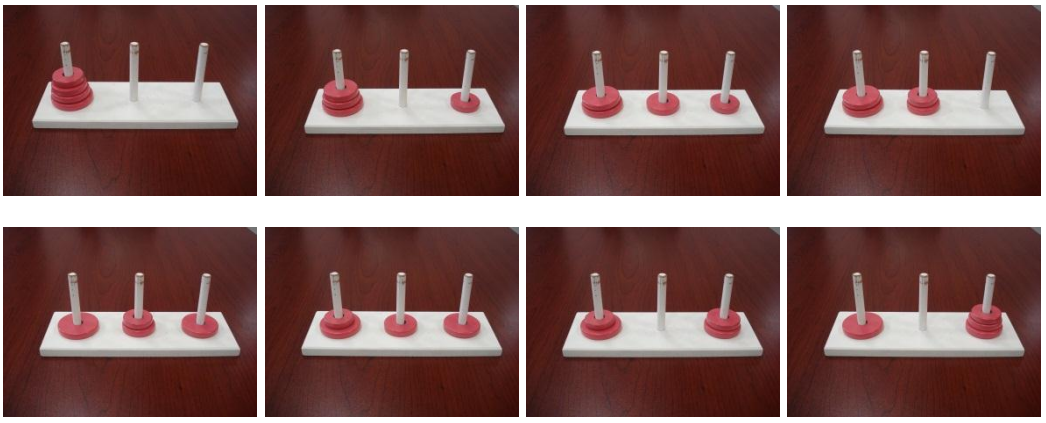


Figura 4.1 El problema de las torres de Hanoi: pasar los *n*-1 discos al poste auxiliar

En la *Figura 4.1* se ilustra como pasar los *n*-1 discos al poste auxiliar C, de tal forma que el disco más grande queda libre para ponerlo hasta abajo del poste B. En la *Figura 4.2* se ilustra el paso del disco más grande al poste B. Ahora, el problema se reduce a pasar los n-1 discos del poste C al B, lo cual se hace de manera similar a la secuencia presentada en la *Figura 4.1*.

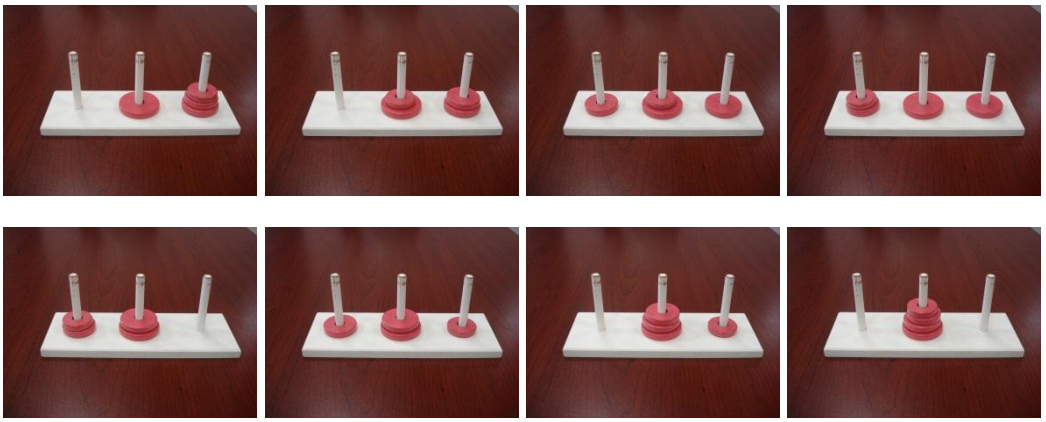


Figura 4.2 El disco más grande queda hasta abajo del poste B. Después pasar los *n*-1 discos al poste B

La solución es sencilla de entender con este enfoque, aunque se complica al llevarla a la práctica conforme aumenta el número de discos a mover de A a B. El tiempo de ejecución de este algoritmo es el siguiente.

$$T(n) = \begin{cases} 1 & \textit{Caso base: } n = 1 \\ T(n\text{-}1) + 1 + T(n\text{-}1) & n > 1 \end{cases}$$

El tiempo de ejecución T(*n*) para pasar n discos del poste A al poste B es la suma del tiempo T(*n*-1) para pasar *n*-1 discos de A a C, más el tiempo T(1) para pasar el disco *n* a B, más el tiempo T(*n*-1) para pasar los *n*-1 discos de C a B. El caso base se presenta cuando solo hay que mover un disco.

IV.6 El problema del torneo de tenis

Un conocido problema para ilustrar la aplicación de divide y vencerás es el problema del torneo de tenis. En este problema necesitamos organizar un torneo de tenis con *n* jugadores, en donde cada jugador ha de jugar exactamente una vez contra cada uno de sus posibles *n*–1 competidores y, además, ha de jugar un partido cada día.

Si *n* es potencia de 2, implementaremos un algoritmo para construir un cuadrante de partidas del torneo que permita terminarlo en *n*–1 días.

El caso más simple se produce cuando sólo tenemos dos jugadores, cuya solución es fácil pues basta enfrentar uno contra el otro, como se muestra en la *Figura 4.3*, en la que los jugadores 1 y 2 se enfrentan.

	d1
J1	2
J2	1

Figura 4.3 El problema del torneo de tenis: dos jugadores se enfrentan el día uno

Si n > 2, aplicamos la técnica divide y vencerás para construir la tabla, suponiendo que ya tenemos calculada una solución para la mitad de los jugadores, esto es, que está completo el cuadrante superior izquierdo de la tabla, que enfrenta entre sí a todos los jugadores de la mitad superior de la tabla. El cuadrante inferior izquierdo debe enfrentar entre sí a todos los jugadores de la mitad inferior de la tabla (3 y 4). Como se puede apreciar en el ejemplo de la *Figura 4.4*.

	d1	d2	d3
J1	2	3	4
J2	1	4	3
J3	4	1	2
J4	3	2	1

Figura 4.4 El problema del torneo de tenis: cuatro jugadores se enfrentan entre sí a lo largo de tres días

El cuadrante superior derecho enfrenta a los jugadores de la mitad superior con los de la mitad inferior y se puede obtener enfrentando a los jugadores numerados 1 a $n/2$ contra $(n/2)+1$ a n, respectivamente, en el día $n/2$ y después rotando hacia abajo los valores $(n/2)+1$ a n cada día.

El cuadrante inferior derecho enfrenta a los jugadores de la mitad superior contra los de la mitad superior y se puede obtener enfrentando a los jugadores $(n/2)+1$ a n contra 1 a $n/2$, respectivamente, en el día $n/2$ y después rotando hacia arriba los valores 1 a n cada día, en sentido contrario a como lo hemos hecho para el cuadrante superior derecho.

Así, para $n = 8$ jugadores tenemos la tabla derecha de la *Figura 4.5*.

	d1
J1	2
J2	1

	d1	d2	d3
J1	2	3	4
J2	1	4	3
J3	4	1	2
J4	3	2	1

	d1	d2	d3	d4	d5	d6	d7
J1	2	3	4	5	6	7	8
J2	1	4	3	6	7	8	5
J3	4	1	2	7	8	5	6
J4	3	2	1	8	5	6	7
J5	6	7	8	1	4	3	2
J6	5	8	7	2	1	4	3
J7	8	5	6	3	2	1	4
J8	7	6	5	4	3	2	1

Figura 4.5 El problema del torneo de tenis: ocho jugadores se enfrentan entre sí a lo largo de siete días

En el diseño de algoritmos siempre hay que afrontar varios compromisos. Cuando se trata de competencias hay que balancear los costos siempre que sea posible. Es mejor que los subproblemas tengan un tamaño aproximadamente igual.

En resumen, divide y vencerás es una de las principales técnicas para el diseño de algoritmos. Consiste en "la descomposición de un problema en n problemas más pequeños, de modo que a partir de la solución de dichos problemas sea posible construir con facilidad una solución al problema completo. Dividir un problema en subproblemas iguales o casi iguales es crucial para obtener un buen rendimiento." [Aho et al., 1998].

Los problemas típicos que pueden atacarse con divide y vencerás son: ordenamiento por mezcla o fusión (mergesort), ordenamiento rápido (quicksort) y búsqueda binaria, los cuales estudiaremos en las próximas secciones. Cabe mencionar también la multiplicación de enteros grandes y multiplicación de matrices, los cuales pueden consultarse en [Brassard y Bratley, 2008; Aho et al., 1998; Guerequeta y Vallecillo, 2000].

IV.7 El método mergesort (ordenamiento por mezcla)

El algoritmo de ordenamiento por mezcla (mergesort) utiliza la técnica de divide y vencerás para realizar la ordenación de un arreglo. Su estrategia consiste en dividir un arreglo en dos subarreglos que sean de un tamaño tan similar como sea posible, dividir estas dos partes mediante llamadas recursivas, y finalmente, al llegar al caso base, mezclar los dos partes para obtener un arreglo ordenado.

El algoritmo para mezclar es el siguiente: se parte el arreglo a la mitad y se trabaja con dos índices: i_1 e i_2. Con i_1 se recorre la primera mitad del arreglo y con i_2 se recorren los elementos de la segunda mitad del arreglo, como se ilustra en la *Figura 4.6*. Se compara el contenido del arreglo en el índice i_1 , con el contenido del índice i_2 , se elige el más pequeño (cuando se requiere orden ascendente) para colocarlo en el arreglo final y se incrementa en uno el índice cuyo contenido se seleccionó. En caso de que los dos elementos sean iguales, es decisión del programador elegir por default el elemento del primer o el del segundo arreglo.

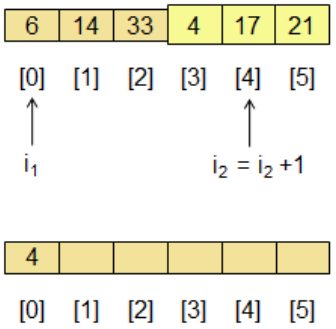


Figura 4.6: La mezcla requiere dos índices, uno para la primera mitad y otro para la segunda mitad del arreglo

A continuación se presenta el código en C/C++ de la función de ordenamiento por mezcla. Recibe como parámetros el apuntador al arreglo, el índice del inicio y el índice del final del arreglo. Se puede observar que la función hace dos llamadas recursivas, para ordenar, respectivamente, la primera y la segunda mitad del arreglo. Cuando se hace la segunda llamada, se parte de la base de que la primera mitad ya está ordenada. La llamada a la función `mezcla` se hace una vez que ya se tienen ordenadas tanto la primera mitad como la segunda.

```
void merge_sort(int *a, int ini, int fin) {
  int med;
  if (ini ≤ fin) {
    med = (ini + fin)/2;
    merge_sort(a, ini, med);
    merge_sort(a, med + 1, fin);
    mezcla(a, ini, med, fin);
  }
}
```

La función `mezcla` recibe como parámetros el apuntador al inicio del arreglo o vector, el índice al inicio del arreglo, el índice a la mitad del arreglo y el índice del fin del arreglo.

Se utiliza un arreglo auxiliar en el que se guarda la mezcla de la primera y la segunda mitad del arreglo recibido. Cuando se termina el procedimiento de mezcla, se copia el contenido del arreglo auxiliar al arreglo recibido.

```
void mezcla(int *a, int ini, int med, int fin) {
    int *aux;
    aux = new int[fin - ini + 1];
    int i = ini;      // Índice sub vector izquierdo
    int j = med + 1; // Índice sub vector derecho
    int k = 0;       // Índice del vector aux

    // Mientras ninguno de los indices llegue a su fin se hacen
    // comparaciones. El elemento más pequeño se copia a "aux"
    while (i <= med && j <= fin) {
        if (a[i] < a[j]) {
            aux[k] = a[i];
            i++;
        }
        else {
            aux[k] = a[j];
            j++;
        }
        k++;
    }

    // Uno de los dos sub-vectores ya ha sido copiado del todo,
    //  falta copiar el otro sub-vector
    while (i <= med) {
        aux[k] = a[i];
        i++;
        k++;
    }

    while (j <= fin) {
        aux[k] = a[j];
        j++;
        k++;
    }

   // Copiar los elementos ordenados de aux al vector "a"
    for (int m = 0; m < fin-ini+1; m++)
      a[ini + m] = aux[m];
    delete [] aux;
}
```

Ejemplo 1.- Hacer la prueba de escritorio para el algoritmo de `mergesort` con los siguientes datos de entrada: a={6,14,12,4,2}.

Al principio, se hacen llamados recursivos a `mergesort` partiendo los sub-arreglos por la mitad con cada llamada recursiva, hasta que se obtiene un sub-vector de tamaño 1, entonces llegamos al caso base, en la *Figura 4.7* se ilustra lo que sucede con el lado izquierdo del vector del ejemplo.

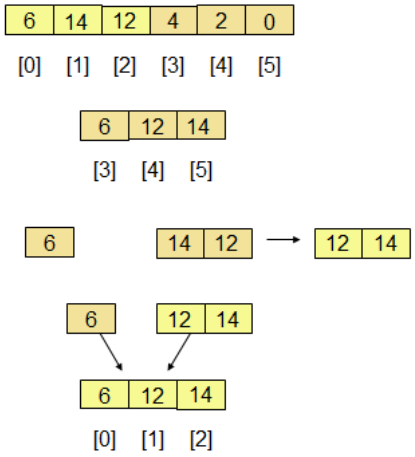


Figura 4.7: Subdivisión y mezcla del sub vector {6,14,12}

En la *Figura 4.7* se observa que cuando partimos el vector {6,14,12} a la mitad, queda un sub vector de tamaño 1={6} a la izquierda, con el caso base y un sub vector de tamaño 2 = {14, 12} a la derecha. Entonces solo se llama a `mergesort` para el sub-vector derecho y se obtiene {14} a la derecha y {12} a la izquierda. Cuando se obtienen los casos base comienza el regreso de las llamadas recursivas, entonces se procede a combinar {14} con {12} y obtenemos el vector {12, 14. Posteriormente, se combina el {6} con el {12, 14} y se obtiene el sub vector de la mitad izquierda ordenado: {6, 12, 14}.

El procedimiento para ordenar el primer sub vector de la derecha es similar y se muestra en la prueba de escritorio de la *Figura 4.8*.

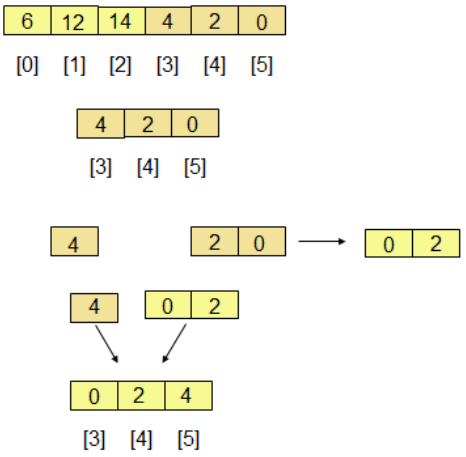


Figura 4.8: Subdivisión y mezcla del sub vector {4,2,0}

Ahora ya tenemos los dos sub-vectores principales ordenados por lo que solo resta mezclarlos. En el ejemplo 2 describimos como se mezclan dos sub-arreglos.

Ejemplo 2.- Mezclar los dos sub-arreglos ordenados {6,14,33} y {4,17,21}.

En la prueba de escritorio de las *Figuras 4.9* y *4.10* se muestra la secuencia de la función `mezcla` para combinar dos mitades (ya ordenadas) de un arreglo.

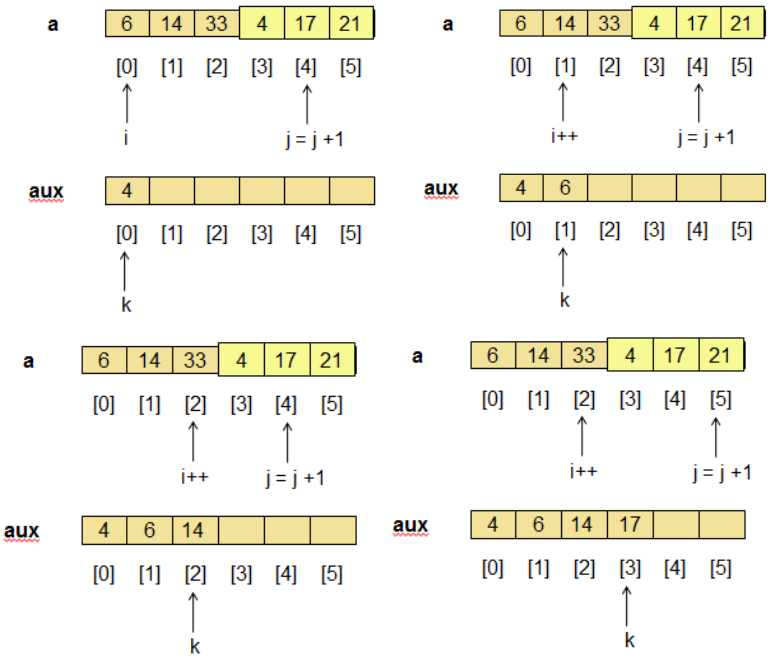
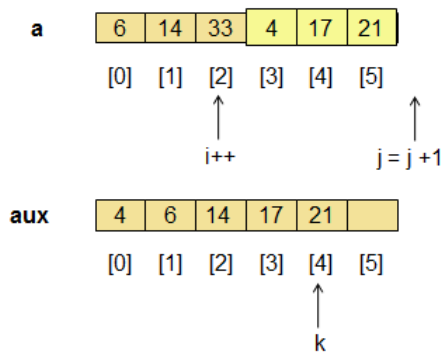


Figura 4.9: Mezcla de {6,14,33} con {4,17,21} (parte I)

```
void mezcla(int *a, int ini, int med, int fin) {
    int *aux;
    aux = new int[fin - ini + 1];
    int i = ini;      // Índice subvector izquierdo
    int j = med + 1; // Índice subvector derecho
    int k = 0;       // Índice del vector aux

    // Mientras ninguno de los indices llegue a su fin se hacen
    // comparaciones. El elemento más pequeño se copia a "aux"
    while (i <= med && j <= fin) {
        if (a[i] < a[j]) {
            aux[k] = a[i];
            i++;
        }
        else {
            aux[k] = a[j];
            j++;
        }
        k++;
    }
    .
    .
    .
```




```
void mezcla(int *a, int ini, int med, int fin) {
    ...
}
    // Uno de los dos sub-vectores ya ha sido copiado del todo,
    //  falta copiar el otro sub-vector
    while (i <= med) {
        aux[k] = a[i];
        i++;
        k++;
    }

    while (j <= fin) {
        aux[k] = a[j];
        j++;
        k++;
    }
```

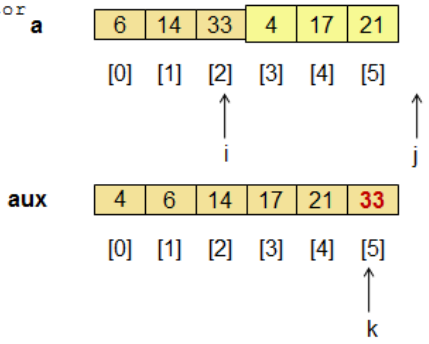


Figura 4.10: Mezcla de {6,14,33} con {4,17,21} (parte II)

La prueba de escritorio de la función `mezcla` de las *Figuras 4.9 y 4.10* muestra que se elige el elemento menor de cada sub arreglo, y se incrementa el índice correspondiente al sub-arreglo del elemento elegido, hasta que uno de los índices rebase el final del sub arreglo, entonces se procede a copiar los demás elementos que falten del otro sub-arreglo.

En mergesort, el mayor esfuerzo se hace durante el proceso de regreso de la recursión, ya que es cuando se lleva a cabo la mezcla. Al dividir el problema no es necesario hacer muchos cálculos, ya que solo hay que partir los sub-arreglos por la mitad.

Para que el mergesort sea eficiente, el tamaño de los dos sub-vectores debe ser igual o muy parecido, ya que en el caso contrario, cuando un sub-vector es más pequeño que el otro, el algoritmo es de orden $\mathbf{O}(n^2)$, es decir, bastante ineficiente.

IV.7.1 Tiempo de ejecución de mergesort

Describiremos el funcionamiento general del mergesort en base a los siguientes tres pasos a ejecutar:

1. Dividir recursivamente la primera mitad del arreglo.
2. Dividir recursivamente la segunda mitad del arreglo.
3. Mezclar los dos subarreglos para ordenarlos.

Como se puede apreciar, ignoramos los casos base.

Sea C[el arreglo a ordenar, A el subarreglo izquierdo y B el subarreglo derecho, *i* el índice en A y *j* el índice en B, como se muestra en la *Figura 4.11*.

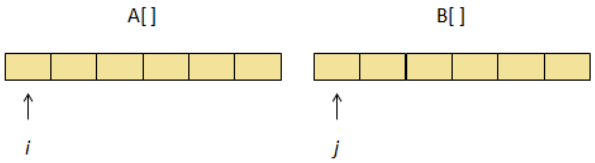


Figura 4.11: Subarreglos que resultan de la división en el mergesort

El siguiente pseudocódigo representa un algoritmo que mezcla estos dos arreglos en un solo arreglo C ordenado de tamaño *n*.

```
Procedimiento Mezcla (↓↑A Entero Vector[1..n/2], ↓↑B Entero Vector[1..n/2], ↑C ∈ Entero Vector[1..n] )
Variables
  i = 1
  j = 1
Acciones
  Para k← 1 ... n Hacer
   Si A[i] < B[j] Entonces
      C[k] = A[i]
      i = i + 1
   sino
     C[k] = B[j]
     J = j + 1
   Fin Si
  Fin Para
```

Para simplificar el cálculo de operaciones elementales, supondremos que cada línea es una operación, de la siguiente manera:

```
Procedimiento Mezcla (↓↑A Entero Vector[1..n/2], ↓↑B Entero Vector[1..n/2], ↑C ∈ Entero Vector[1..n] )
Variables
  i = 1                    1
  j = 1                    1
Acciones
  Para k← 1 ... n Hacer    1
   Si A[i] < B[j] Entonces 1
      C[k] = A[i]          1
      i = i + 1            1
   sino
     C[k] = B[j]           1
     J = j + 1             1
   Fin Si
  Fin Para
```

Entonces, para obtener el número total de operaciones, sumamos las dos líneas de inicialización, más las dos primeras líneas que siempre se ejecutan y, agregamos alguna de las otras dos posibilidades de la condición (según sea el caso) en las que se agregan dos más en ambos casos. Por lo tanto, podemos decir que el tiempo de ejecución del mergesort sobre un arreglo con *n* elementos es la suma del tiempo de ejecución de la recursión más el de la mezcla, es decir $2T(n/2) + 4n + 2$.

$T(n) = 2T(n/2) + 4n + 2$

Considerando que $n \geq 1$, se cumple que:

$T(n) \leq 2T(n/2) + 6n$

En la *Figura 4.12* podemos observar que la profundidad de la recursión en el mergesort para $n = 8$ es 3.

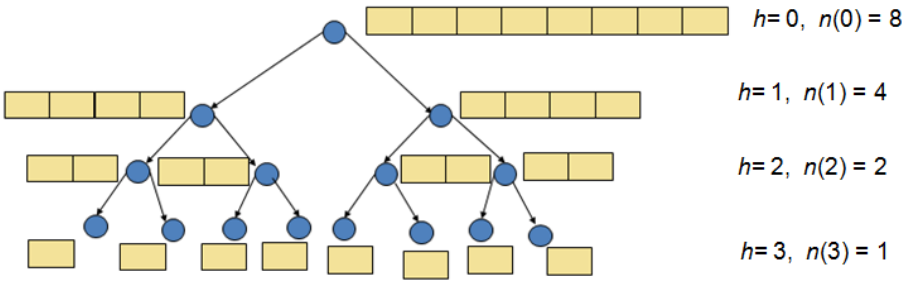


Figura 4.12: Llamadas recursivas en mergesort

Sean h_{max} la profundidad máxima que resulta de dividir un arreglo a la mitad hasta obtener subarreglos de tamaño 1, el nivel de profundidad dentro del árbol que se produce con ésta división, y $n(h)$ el número de elementos del arreglo en la profundidad h. La profundidad máxima h_{max} se alcanza cuando $n(h)=1$. Ahora, el número total de elementos a ordenar es $n(0)$ y como $n(h) = n(0)/2^h$, tenemos que $n(hmax)= n(0)/2^{hmax}$. De lo anterior obtenemos $n(0)/2^{hmax}=1$, $n(0) = 2^{hmax}$ y $h_{max} = \log_2 n(0)$.

Ejemplo: si $n(0)=3$, se tiene $h_{max} = \log_2(3) \cong 1.58$. Esto significa que la recursión tendrá una profundidad máxima de 2 en algunas ramas y de 1 en otras. Concluimos que la profundidad máxima será

$$h_{max} = \lceil \log_2(n(0)) \rceil \leq log_2(n(0)) + 1$$

Previamente habíamos encontrado que $T(n) \leq 2T(n/2) + 6n$, de donde

$T(n) \leq 2T(n/2) + 6n = 2(2T(n/2/2) + 6n/2) + 6n = 4T(n/4) + 6n + 6n = 4T(n/4) +12n$

$= 4(2T(n/8) + 6n/4) + 12n = 8T(n/8) + 18n = \ldots = 2^h T(n/2^h) + 6hn = \ldots$

Entonces

$T(n) \leq 2^h T(n/2^h) + 6hn$

Para que esto último quede en función del caso base $T(1)=6$ (solo se realizan las seis operaciones porque ya no hay recursión), substituimos h por $h_{max} = \log_2(n)$ en la desigualdad anterior.

$$T(n) = 2^{log_2(n)} T\left(\frac{n}{2^{log_2(n)}}\right) + 6log_2(n)n = nT(1) + 6nlog_2(n) = 6n + 6n\, log_2(n)$$

Entonces, el tiempo de ejecución del mergesort;

$T(n) = 6n + 6n\log_2(n)$ es $\mathbf{O}(\ n\log_2(n)\ )$.

Nota.- A petición de algunos alumnos se hace la siguiente aclaración:

$$b^x = n \quad (1)$$

Por definición de logaritmo:

$$x = log_b n \quad (2)$$

Sustituyendo la *ec. 2* en la *ec. 1*:

$$b^{log_b n} = n$$

Por lo tanto:

$$2^{log_2 n} = n$$

IV.8 El método Quicksort

El método de ordenamiento rápido, llamado *quicksort* está basado en la técnica de divide y vencerás y es "probablemente el algoritmo de ordenamiento más utilizado, pues es muy fácil de implementar, trabaja bien en casi todas las situaciones y consume en general menos recursos (memoria y tiempo) que otros métodos" [Guerequeta y Vallecillo, 2000]. A diferencia del *mergesort*, el mayor esfuerzo se hace al construir los subcasos y no en combinar las soluciones. En el primer paso, el algoritmo selecciona como *pivote* uno de los elementos del arreglo a ordenar. El segundo paso es separar los elementos del arreglo a ambos lados del pivote, es decir, se desplazan los elementos de tal manera que los que sean mayores que el pivote queden a su derecha, y los demás queden a su izquierda. Cada uno de los subarreglos a los lados del pivote se ordena posteriormente mediante llamadas recursivas al algoritmo. El resultado final es un arreglo completamente ordenado. En la *Figura 4.13* se ilustra la forma en la que se hace la partición: I_1 corresponde al pivote de la primera partición, I_2 son los respectivos pivotes de la parte izquierda y derecha de la segunda partición.

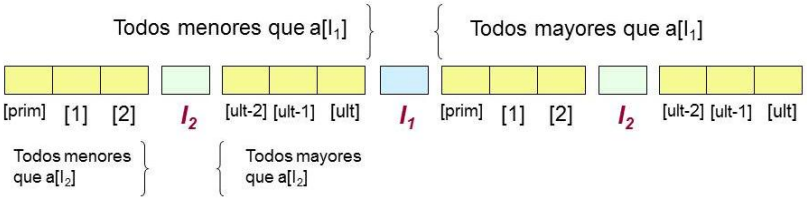


Figura 4.13: Forma en la que se hacen las particiones del quicksort

A continuación el código en C/C++ de la función `quick_sort`. Recibe como parámetros el apuntador al arreglo a ordenar y los índices del inicio y del final del arreglo.

```
void quick_sort(int *a, int prim, int ult) {

  if(prim>=ult) return;

  int part = particion(a, prim, ult);

  quick_sort(a, prim, part - 1);

  quick_sort(a, part+1, ult);
}
```

Devuelve el índice de la partición
todos los de la izq. de a[part] son menores que a[part]
todos los de la der. de a[part] son mayores que a[part]

La función `particion(a,prim,ult)` regresa el índice de la partición y hace que todos los elementos a la izquierda del elemento en la partición sean menores que `a[part]` y todos los elementos de la derecha sean mayores que `a[part]`. A continuación, en la *Figura 4.14* se ilustra un ejemplo de cómo funciona `particion` cuando se le invoca para un arreglo con 6 elementos, en donde `prim` = 0 y `ult` = 5.

```
int particion( int *a, int prim, int ult ){
  // permuta los elementos de a de manera que
  // a[i] <= a[j] si i<j,
  // a[i] >  a[j] si j<i
  // y devuelve la posición j en donde se colocó a a[prim]
  int i = prim;
  int j = ult+1;
  int p = a[prim];
  do{ i++;
  }while( (a[i]<=p) && (i<=ult) );
  do{ j--;
  }while( (a[j]>=p) );
  while( i<j ){
    Intercambia(a,i,j);
    do{i++;
    }while( a[i]<= p );
    do{j--;
    }while( a[j]>= p );
 };
 Intercambia(a,prim,j);
 return j;
}
```

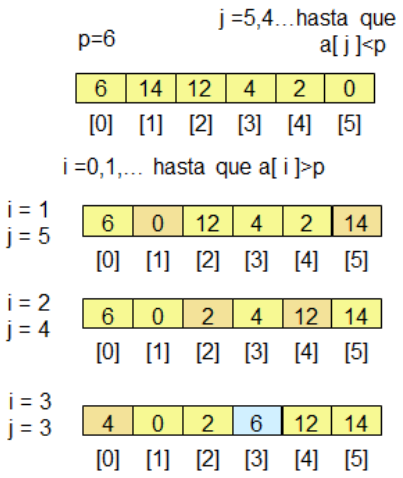


Figura 4.14: La función `particion`

En el *quicksort* el mayor esfuerzo se hace en la partición. Durante el proceso de regreso de la recursión no es necesario hacer cálculos, ya que los sub-vectores ya están ordenados.

"La partición en el *quicksort* es costosa, pero una vez ordenados los dos sub-vectores la combinación es inmediata, sin embargo, la división que realiza el método de ordenación por *Mezcla* consiste simplemente en considerar la mitad de los elementos, mientras que su

proceso de combinación es el que lleva asociado todo el esfuerzo" [Guerequeta y Vallecillo, 2000].

En los algoritmos clásicos de ordenamiento, el mejor caso (el más rápido) se presenta cuando los elementos del arreglo están completamente ordenados y el peor caso (el más lento) cuando están completamente desordenados. Sin embargo, *quicksort* es un algoritmo especial, en el que el peor y el mejor caso no dependen de qué tan ordenado está el arreglo, sino del elemento que se elige como pivote. Estudiaremos esto a detalle en la siguiente sección.

IV.8.1 Tiempo de ejecución del Quicksort

El peor caso en el quicksort se presenta cuando el pivote se elige de tal forma que los sub-arreglos de la derecha y de la izquierda están extremadamente desequilibrados, por ejemplo, que uno de los lados ya está ordenado (un elemento), lo que implica una llamada recursiva a un caso de tamaño cero, mientras que en el otro lado se tiene un caso cuyo tamaño solo se reduce en un elemento. Esto se da cuando el pivote elegido es el mayor o el menor de los elementos en el arreglo. Suponiendo que esto sucede en cada llamada recursiva, el tiempo de ejecución será el resultado de sumar el tiempo de la recursión que ordena el subarreglo, cuyo tamaño se reduce en uno, más el tiempo de la llamada al caso de tamaño cero, más el tiempo de hacer la partición, es decir

$$T(n) = T(n\text{-}1) + T(0) + T_{\text{partición}}$$

El tiempo para ejecutar la partición está en función del tamaño n del arreglo. Supondremos que el costo de cada partición es una constante c multiplicada por el tamaño de la partición. Además, $T(0) = 0$, por lo tanto, para el peor caso:

$$\begin{aligned} T(n) &= T(n\text{-}1) + cn \\ &= T(n\text{-}2) + c(n\text{-}1) + cn \\ &= \dots \end{aligned}$$

$$= \sum_{i=0}^{n} c(n-i) = 0 + c + 2c + 3c + \cdots + (n-1)c + nc$$

$$= cn + c(n-1) + c(n-2) + \cdots + c(1) = \sum_{i=1}^{n} c(i)$$

$$T(n) = c\sum_{i=1}^{n} i = c\left(\frac{n(n+1)}{2}\right) \in O(n^2)$$

Por lo tanto, en el peor caso tenemos que el tiempo de ejecución del quicksort es $O(n^2)$.

La demostración por inducción matemática de que el tiempo de ejecución de quicksort en el mejor caso y en el caso promedio es O(*n* log(*n*)) se puede consultar en [Brassard y Bradley, 2008]. En esta sección daremos la explicación intuitiva de [Cormen et al., 2009] para mostrar que el tiempo de ejecución promedio de quicksort es **O**(*n* log(*n*)).

La elección del pivote hace que el subarreglo de la izquierda contenga a todos los elementos menores a éste y que el subarreglo de la derecha contenga a los que son mayores. Esta partición no es regular, es decir, un subarreglo es más pequeño que el otro. En la *Figura 4.15* suponemos, por ejemplo, que la elección del pivote es tal que el 10% de los elementos del arreglo queda en el sub-arreglo izquierdo, mientras que el 90% restante queda en el sub-arreglo derecho. También podemos apreciar que en la *Figura 4.15* las ramas del subárbol izquierdo llegan rápidamente a sus hojas, ya que la cantidad de elementos a ordenar es más pequeña que la del subárbol derecho.

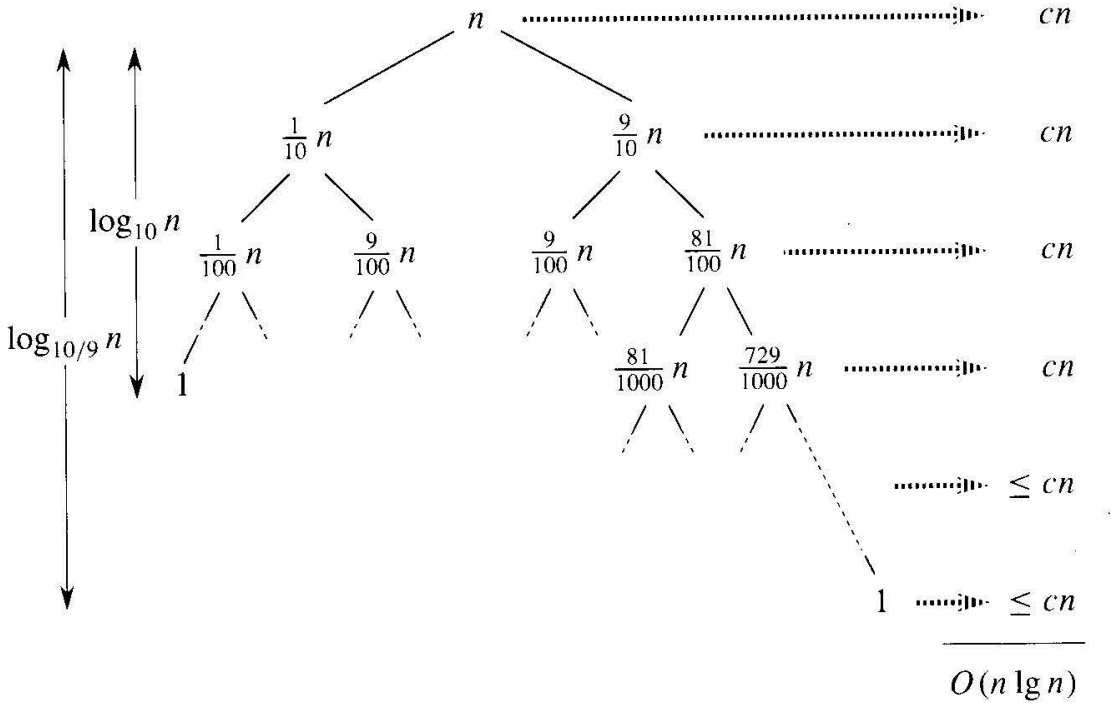


Figura 4.15: Árbol de recursión del quicksort [Cormen et al., 2009]

El tiempo de ejecución para realizar la operación de partición en cada nivel del árbol es *cn*. Para estimar el tiempo total *T*(*n*) es necesario calcular cuál es la profundidad de la recursión. La recursión termina cuando el número de elementos a ordenar es 1.

Como el sub-arreglo derecho es más grande que el izquierdo, entonces la profundidad máxima se alcanza en la hoja de la extrema derecha del árbol. Ahora bien, sea *h* la

profundidad, podemos observar que para h =1 el tamaño del subárbol derecho es $\frac{9}{10}n$, para h =2 el tamaño del subárbol derecho es $(\frac{9}{10})^2 n$, para h =3 es $(\frac{9}{10})^3 n$ y, en general, el tamaño del subárbol derecho es:

$$(\frac{9}{10})^h n$$

La hoja de la extrema derecha del árbol se alcanza cuando $(\frac{9}{10})^h n = 1$, entonces:

$$log_{\frac{9}{10}}\left(\left(\frac{9}{10}\right)^h n\right) = log_{\frac{9}{10}}(1)$$

$$h + log_{\frac{9}{10}}(n) = 0$$

$$h = -log_{\frac{9}{10}}(n) = log_{\frac{10}{9}}(n)$$

Como la profundidad h crece logarítmicamente podemos decir que es Θ(log n). De la *Figura 4.15* podemos concluir que el tiempo de ejecución en cada nivel de recursión es *cn* y que el tiempo total se obtiene multiplicando *cn* por la profundidad del árbol de recursión, la cual crece logarítmicamente. Así, T(n) es $\cong log_{\frac{10}{9}}(n)cn \in \mathbf{O}(nlog(n))$.

IV.9 El método de búsqueda binaria

El algoritmo de búsqueda binaria es un ejemplo claro de la técnica divide y vencerás y es el que normalmente utilizamos para hacer búsquedas en el diccionario o en un directorio telefónico. La búsqueda binaria también es un algoritmo de simplificación, porque el problema se va reduciendo con cada paso, con lo que se obtiene un tiempo de ejecución muy bueno (normalmente de orden logarítmico). La búsqueda binaria es un método más eficiente que la búsqueda secuencial, sin embargo, se requiere que los elementos dentro del arreglo en donde se hará la búsqueda se encuentren ordenados, ya sea de manera ascendente o descendente.

La búsqueda binaria puede tener dos enfoques: recursivo y no recursivo. El recursivo tiene la ventaja de ser simple y claro, y por lo tanto fácil de depurar y mantener. Sin embargo tiene las desventajas que poseen este tipo de algoritmos: su tiempo de ejecución es mayor que el de los no recursivos, aunque del mismo orden **O**(log(*n*)), y además tienen mayor *complejidad espacial*, debido a que requieren más memoria (para guardar la pila de recursión). A continuación se presenta el enfoque recursivo:

- Sea x el número buscado.
    - si (a[mitad] = x) entonces se encontró el elemento buscado.

- si (a[mitad] > x) hacer la búsqueda binaria en la mitad inferior del arreglo.
- si (a[mitad] < x) hacer la búsqueda binaria en la mitad superior del arreglo.

Cuyo pseudocódigo es el siguiente:

```
Función BusquedaBinariaRecursiva ( a ∈Vector[1...n] ∈ Entero; prim; ult; x; ∈ Entero) ⟶ Lógico)
  Variables
    mitad ∈ Entero
  Acciones
    Si ( prim > ult) Entonces regresa ( Falso)
    Si ( prim = ult) Entonces regresa ( a[prim] = x )

    mitad ⟵ (prim + ult) / 2
    Si (x = a[mitad]) Entonces regresa Verdadero

    Si (x  < a[mitad]) Entonces
       regresa BusquedaBinariaRecursiva( a, prim, mitad-1, x  )
    Si no
        regresa BusquedaBinariaRecursiva( a, mitad+1, ult, x  )
    Fin Si
```

El código en C/C++ del algoritmo anterior es:

```
bool BusquedaBinariaRecursiva(int *a, int prim, int ult, int x){
 int mitad;
 if( prim>ult) return (false);
 if( prim==ult) return (a[prim]== x);

 mitad=(prim+ult)/2;
 if( x==a[mitad] ) return (true);

 if( x<a[mitad] )
    return BusquedaBinariaRecursiva(a,prim,mitad-1,x);
 else
    return BusquedaBinariaRecursiva(a,mitad+1,ult,x);

}
```

Cualquier algoritmo recursivo se puede modificar para eliminar la recursión, que se logra reemplazando las llamadas recursivas por un ciclo. A continuación se presenta el pseudocódigo para el enfoque no recursivo de la búsqueda binaria.

```
Función BusquedaBinariaNoRecursiva (a ∈Vector[1..n]∈ Entero, x ∈ Entero) →Entero
  Variables
    prim, ult, medio ∈ 1...n
    encontrado ∈ Lógico
  Acciones
    Encontrado ← Falso
    prim ← 1
    ult ← n
    Mientras ((¬encontrado) OR (prim ≤ ult)) Hacer
       medio ← (prim + ult) / 2
       Si (x = a[medio]) Entonces
          encontrado ← Verdadero
       Si no
          Si (x > a[medio]) Entonces
            prim ← medio + 1
          Si no
            ult ← medio – 1
          Fin Si
       Fin Si
    Fin Mientras
    Si ¬encontrado Entonces
        regresa 0
    Si no
        regresa medio
    Fin Si
```

En este caso, el algoritmo regresa el índice en donde se encuentra el elemento, si es que éste existe, en lugar de regresar Verdadero como en el caso recursivo.

El código en C/C++ del algoritmo anterior es el siguiente.

```
// regresa el índice del arreglo donde se encuentra x,
// regresa -1 si no encuentra x
int BusquedaBinariaNoRecursiva (int x, int *a, int n){
    int prim=0, ult= n-1, medio;
    bool encontrado=false;
    while ((!encontrado) || (prim <= ult)){
      medio = (prim + ult)/ 2;
      if (x == a[medio])
          encontrado = true;
      else{
          if(x > a[medio])
             prim = medio + 1;
          else
             ult = medio - 1;
      };
    };
    if (!encontrado)
        return -1;
    return medio;
};
```

IV.9.1 Tiempo de ejecución de la búsqueda binaria no recursiva

En la función `BusquedaBinariaNoRecursiva`, el peor de los casos se presenta cuando el número buscado no se encuentra, es decir, cuando `prim > ult`. Con cada iteración, el número de elementos que todavía pueden coincidir con el número buscado se reduce aproximadamente a la mitad. Las iteraciones terminan cuando el número de elementos que todavía pueden coincidir es 1. Sea *k* el número máximo de iteraciones. Se debe cumplir que

$$n\,(\tfrac{1}{2})^k \leq 1$$

$$(\frac{1}{2})^k \leq \frac{1}{n}$$

$$\frac{1}{2^k} \leq \frac{1}{n}$$

$$2^k \geq n$$

$$k \geq log_2(n)$$

donde *n* es el tamaño del arreglo.

Entonces, cuando *k* cumple con lo anterior, el algoritmo seguro termina aunque puede terminar antes si encuentra el dato. Por esto, $k \in O(\log(n))$. En la *Figura 4.16* se observa

que el número máximo de iteraciones que se realizan con el programa de búsqueda binaria crece logarítmicamente con el tamaño del arreglo.

n No. de elementos del arreglo	k No. de iteraciones	$\underline{\log_2 n}$ suelo del logaritmo
1	1	0
2	2	1
3	2	1
4	3	2
5	3	2
6	3	2
7	3	2
8	4	3
9	4	3
10	4	3
11	4	3
12	4	3
13	4	3
14	4	3
15	4	3
16	5	4

Figura 4.16: Crecimiento del número de iteraciones en función del tamaño del arreglo en la búsqueda binaria

En la *Figura 4.16* se observa que se realiza un máximo de una iteración de búsqueda binaria cuando el tamaño del arreglo es 1. Cuando el arreglo contiene 2 o 3 elementos se ejecutan un máximo de 2 iteraciones. Cuando el tamaño del arreglo va de 4 a 7 elementos, se llevan a cabo un máximo de 3 iteraciones. Generalizando, el número de iteraciones es el suelo($\log_2 n$) +1, es decir, el número de iteraciones k es del orden **O**(log n). Entonces, el orden de complejidad de la búsqueda binaria es **O**(log n).

Además de los razonamientos anteriores, el orden de complejidad de la búsqueda binaria se puede obtener matemáticamente resolviendo la ecuación de recurrencia del ejemplo 2 de la sección II.8. La ecuación de recurrencia para la búsqueda binaria es la siguiente:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ T(n/2) + c_2 & n > 1 \end{cases}$$

Tiempo de la partición

Y su solución:

$T(n) = T(1) + \log_2 n$ para $n > 1$
$= 1 + \log_2 n$

$T(n)$ es **O**(log n).

El orden de complejidad del algoritmo de búsqueda binaria es logarítmico, mientras que el de búsqueda secuencial es lineal. Por esto podemos decir que la búsqueda binaria es una mejor opción con respecto a la secuencial cuando el tamaño del conjunto de entrada es grande. Sin embargo, si el tamaño del conjunto de los datos de entrada es pequeño, la búsqueda secuencial tiene la ventaja de tener una implementación muy sencilla, y la diferencia entre los dos tipos de búsqueda es mínima, como puede observarse en la *Figura 4.17*.

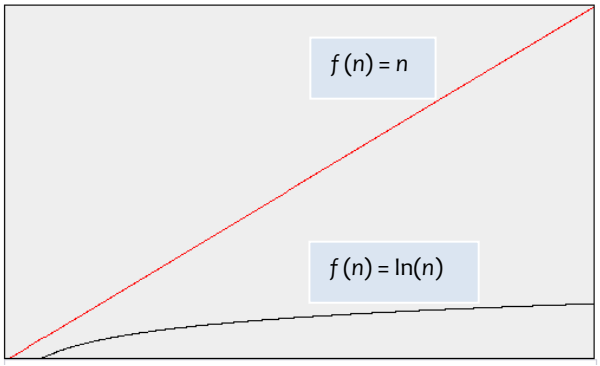


Figura 4.17: $f(n) = n$ vs. $f(n) = \ln(n)$

IV.10 Ejercicios

1.- Implementa el algoritmo divide y vencerás que resuelve el problema de las torres de Hannoi.

2.- Implementa el algoritmo divide y vencerás que resuelve el problema del torneo de tenis.

3.- La búsqueda binaria, ¿Usa la estrategia divide y vencerás?

IV.11 Resumen del capítulo

La experiencia ha demostrado que existen diversos patrones en los que el diseñador se puede basar para resolver ciertos grupos de problemas. Las cinco técnicas de diseño más conocidas, son: divide y vencerás, algoritmos voraces, programación dinámica, backtracking y, ramificación y poda. En el capítulo IV hemos estudiamos el paradigma *divide y vencerás*, el cual consiste en resolver un problema a partir de la solución de subproblemas del mismo tipo, pero de menor tamaño, y después combinar los resultados parciales para obtener la solución del problema original. Hay muchos ejemplos de aplicación de esta técnica. En este capítulo vimos como ejemplos: el problema de las torres de Hanoi, el del torneo de tenis, los métodos de ordenamiento mergesort y quicksort, y finalmente la búsqueda binaria.

Para aplicar correctamente la técnica de divide y vencerás, es importante que los subproblemas sean independientes, es decir, que no exista traslape entre ellos. De lo contrario, el tiempo de ejecución de estos algoritmos será exponencial. Para aquellos problemas en los que hay solapamiento, existe otra técnica de diseño más apropiada, llamada programación dinámica, la cual permite eliminar el problema de la complejidad exponencial debida a la repetición de cálculos. Esta técnica se aborda en el capítulo VI. En el siguiente capítulo, estudiaremos el paradigma de los algoritmos voraces.

Capítulo V Algoritmos voraces

V.1 Definición del paradigma de los algoritmos voraces

Los algoritmos *voraces*, también llamados *ávidos* o *glotones*, se aplican principalmente a problemas de optimización, es decir, problemas en los que hay que maximizar o minimizar algo. Estos algoritmos toman decisiones basándose en la información que tienen disponible de forma inmediata, sin tomar en cuenta los efectos que pudieran tener estas decisiones en el futuro. Su nombre original proviene del término inglés *greedy* y se debe a su comportamiento: en cada etapa "toman lo que pueden" sin analizar las consecuencias, es decir, son glotones por naturaleza.

Los algoritmos *voraces* son fáciles de inventar, porque no requieren de métodos sofisticados, como la evaluación de alternativas o métodos de seguimiento que permitan deshacer las decisiones anteriores. En general, los algoritmos voraces intentan optimizar una *función objetivo*. La *función objetivo* proporciona el valor de una solución al problema, por ejemplo: la cantidad de monedas que se utilizarán para dar cambio, la longitud de la ruta construida, el tiempo necesario para procesar todas las tareas de la planificación. Para hallar el valor de la función objetivo, los algoritmos voraces construyen la solución de la siguiente manera:

- Se define un conjunto o lista de candidatos, por ejemplo: el conjunto de monedas disponibles, las aristas de un grafo que se pueden utilizar para construir una ruta, el conjunto de tareas que hay que planificar, etc.
- En cada paso se considera al candidato indicado por una *función de selección voraz*, que indica cuál es el candidato más prometedor de entre los que aún no se han considerado, es decir, los que aún no han sido seleccionados ni rechazados.
- Conforme avanza el algoritmo, los elementos del conjunto inicial de candidatos pasan a formar parte de uno de dos conjuntos: el de los candidatos que ya se

consideraron y que forman parte de la solución o el conjunto de los candidatos que ya se consideraron y se rechazaron.

- Mediante una *función de factibilidad* se determina si es posible o no completar un conjunto de candidatos que constituya una solución al problema. Es decir, se analiza si existe un candidato compatible con la solución parcial construida hasta el momento.

Para poder utilizar esta técnica, el problema debe poder dividirse en sub-etapas en las que hay varias opciones. La decisión para continuar a la siguiente etapa se toma eligiendo la opción con el mejor resultado parcial.

A continuación presentamos el pseudocódigo de los algoritmos voraces:

```
función voraz ( ↓datos : conjunto) →S : conjunto

  Variables
    candidatos : conjunto
    S : conjunto
    x : elemento
  Acciones
    S ← Ø  // la solución se irá construyendo en el conjunto S
    candidatos ← datos
    Mientras candidatos ≠ Ø AND  NOT(solución(S)) hacer
      x ← SeleccionVoraz( candidatos )
      candidatos ← candidatos – {x}
      Si (factible(S ∪ {x}) entonces S← S ∪ {x}
      Fin Si
    Fin Mientras

    Si solución(S) entonces
      regresar ( S )
    sino
        regresar ( Ø )  // No encontró la solución

Fin voraz
```

En este pseudocódigo, S es el conjunto de elementos con la solución la cual, al inicio, es un conjunto vacío. Mientras todavía existan elementos en el conjunto de candidatos y no se haya llegado a la solución, se realiza una *selección voraz,* es decir, dependiendo si se trata de maximizar o minimizar, se toma el elemento x del conjunto de candidatos cuya evaluación de la función objetivo sea la mayor o la menor, respectivamente. Una vez que se seleccionó el elemento, se procede a la prueba de *factibilidad*. Si la unión de los conjuntos {x} y S completa la solución o hace que la solución parcial sea sea viable, entonces x se

agrega a S, de lo contrario se descarta y, en cualquier caso, x queda excluido del conjunto de candidatos. Este ciclo se realiza hasta que se han agotado todos los elementos del conjunto de candidatos o bien, hasta que se encuentra una solución, es decir, cuando la lista de elementos en S constituye una solución al problema planteado. Los algoritmos voraces asumen que con este procedimiento se minimiza o maximiza (según el caso) la función objetivo.

En este pseudocódigo se puede apreciar que los algoritmos voraces seleccionan el "mejor bocado" para comer, sin pensar en las consecuencias del futuro y sin cambiar de opinión, es decir, una vez que el elemento es considerado como parte de la solución éste queda ahí definitivamente, y cuando un elemento se descarta ya no se vuelve a considerar.

A continuación estudiaremos el funcionamiento del algoritmo voraz en algunos ejemplos.

V.2 El problema del cambio resuelto con un algoritmo voraz

Dado un monto *n* y un conjunto de `denominaciones` de billetes, se desea elegir la mínima cantidad de billetes cuya suma sea *n*.

Para construir la solución con un algoritmo voraz tomaremos cada vez el billete de mayor denominación, siempre y cuando la suma acumulada no sea mayor a *n*. El algoritmo falla cuando se presenta el caso en el que la suma de los billetes no puede igualar a *n*, en este caso el algoritmo ávido no encuentra una solución.

Nótese que en este caso en particular no existe restricción en cuanto al número de billetes que se pueden tomar, es decir, siempre es posible elegir *k* billetes de una determinada denominación. Por lo tanto no es necesario tomar en cuenta si un billete ya se consideró o no.

El pseudocódigo para resolver el problema del cambio con un algoritmo ávido es el siguiente:

```
función cambio ( ↓monto ∈ Entero, ↑ numBilletes ∈ Vector[1...5] de Entero)
  Constantes
    denominaciones ←{ 100, 20, 10, 5, 1 }
  Variables
    suma, iDen ∈ Entero
    numBilletes←{ 0, 0, 0, 0, 0 }
  Acciones
    suma ← 0
    Mientras suma < monto
      iDen ← SeleccionVoraz( monto – suma )
      Si iDen > 5 entonces
          Mensaje "No se encontró solución. Cambio incompleto"
          regresar ( numBilletes )
       Fin Si
       numBilletes[ iDen ] ← numBilletes[ iDen ] + 1
       suma ← suma + denominaciones[ iDen ]
     Fin Mientras
     regresar ( numBilletes )
Fin Función
```

La función `cambio` regresa un vector con la lista de los billetes que forman la solución. Se hace una *selección voraz* eligiendo el billete de mayor denominación posible que falta para completar el cambio sin pasarse. Si no se encuentra un billete adecuado, entonces el algoritmo no encontró la solución y regresa el conjunto vacío. Cuando sí se encuentra un billete compatible con la posible solución, se agrega la denominación del billete seleccionado al vector solución, se suma la cantidad del billete a lo que ya se tenía en suma y se repite la operación para la cantidad que todavía falta cubrir.

El ciclo termina cuando la suma acumulada de los billetes es igual a la cantidad que se desea dar de cambio o cuando ya no se puede encontrar un billete que complete la cantidad requerida.

La función que verifica que la solución sea factible se encuentra dentro de la función de selección voraz. A continuación se presenta el algoritmo de *selección voraz*. En este caso en particular, tomar el billete es factible mientras que su denominación sea menor que la cantidad que se desea completar.

```
Funcion SeleccionVoraz (↓resto ∈ Entero )⟶ ∈ Entero
  Constantes
    denominaciones ⟵{ 100, 25, 10, 5, 1 }
  Variables
    i ∈ Entero
  Acciones
    i⟵1
    Mientras ( i ≤ 5 ) AND ( denominaciones[i] > resto ) Hacer
        i⟵ i + 1
    Fin Mientras
    regresar i
Fin SeleccionVoraz
```

El código en C/C++ del algoritmo anterior es el siguiente:

```
#define NUM_DEN 5

int denominaciones[ NUM_DEN ] = { 100, 20, 10, 5, 1}; // arreglo global
int *numBilletes[NUM_DEN];
numBilletes = new int[NUM_DEN];
for(int i=0; i< NUM_DEN; i++) numBilletes[i]=0;

int SeleccionVoraz ( int  resto ){
  int i = 0;
  while (( i < NUM_DEN) && ( denominaciones[i] > resto ) )
    i++;
  return i;
};

void cambio( int monto, int *numBilletes ){
  int suma=0;
  int iDen=0;
  while( suma < monto){
    iDen = SeleccionVoraz( monto-suma );
    if( indice >= NUM_DEN ){
      cout << "No se encontró solución. Cambio incompleto";
      return;
    };
    numBilletes[iDen] = numBilletes[iDen] + 1;
    suma = suma + denominaciones[iDen];
  };
  return;
}
```

Contamos con un conjunto de 5 billetes de diferente denominación y con la cantidad de cambio que es necesario cubrir con los billetes. El procedimiento de selección voraz busca el billete de mayor denominación dentro del conjunto de denominaciones, suponiendo que siempre podrá tomar un billete de una denominación existente sin que los billetes se agoten.

La condición para elegir un billete es que la denominación del billete no sea mayor a la cantidad que falta por completar (`resto`). Este algoritmo es voraz ya que pretende completar la cantidad pedida mediante billetes de la más alta denominación posible, con la idea de que así se logrará una solución con un mínimo número de billetes, sin embargo esto no es cierto en todos los casos.

Los algoritmos voraces construyen la solución en etapas sucesivas, tratando siempre de tomar la decisión óptima para cada etapa. Sin embargo, la búsqueda de óptimos locales no tiene por qué conducir siempre a un óptimo global. A continuación presentamos un caso en el que falla en algoritmo del cambio.

Supongamos que el sistema monetario está compuesto por los siguientes billetes:

denominaciones←{11,5,1}

Consideremos que se desea dar de cambio: monto←15

El algoritmo voraz tomará primero el billete mayor, por lo que la solución que obtendrá es:

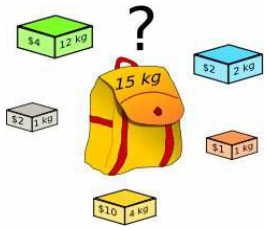
15 = 11 + 1 + 1+ 1+ 1

Sin embargo, la solución óptima es:

15 = 5 + 5 + 5

“La estrategia de los algoritmos ávidos consiste en tratar de ganar todas las batallas sin pensar que, como bien saben los estrategas militares y los jugadores de ajedrez, para ganar la guerra muchas veces es necesario perder alguna batalla.” [Guerqueta y Vallecillo, 2000].

V.3 El problema de la mochila resuelto con un algoritmo voraz



(imagen de Wikipedia)

El problema de la mochila es un problema clásico en el estudio de los algoritmos. Éste consiste en llenar una mochila que soporta un peso máximo W. Se tienen *n* objetos, cada uno de estos objetos tiene un peso w_i y un valor v_i ($i = 1, 2, \ldots, n$).

Los objetos se pueden romper en trozos más pequeños al multiplicarlos por una fracción $0 \leq x_i \leq 1$, de tal manera que el peso del objeto *i* dentro de la mochila sea $x_i w_i$. De esta manera cada objeto *i* contribuye en $x_i w_i$ al peso total de la mochila y en $x_i v_i$ al valor de la carga.

El problema consiste en maximizar el valor de la carga en la mochila, es decir, maximizar:

$$\sum_{i=1}^{n} x_i v_i$$

con la restricción:

$$\sum_{i=1}^{n} x_i w_i \leq W$$

Tanto los pesos como los valores son positivos ($w_i > 0$ y $v_i > 0$) y las fracciones x_i pueden ir de cero a 1 ($0 \leq x_i \leq 1$). Si utilizamos un algoritmo voraz para resolver el problema tendremos que los candidatos son los diferentes objetos y la solución es un vector ($x_1, x_2, \ldots, x_n$) este vector indica qué fracción de cada objeto se debe incluir en la mochila. La *función objetivo* es el valor total de los objetos que están en la mochila. La solución es factible siempre que se cumpla la restricción. Cuando la suma de todos los pesos de los objetos enteros es menor o igual a W, la solución al problema es es trivial (incluir en la mochila todos los objetos), por lo que asumiremos que esta suma sobrepasa W, para que el caso sea interesante. La mochila estará llena si:

$$\sum_{i=1}^{n} x_i w_i = W$$

Que será el caso para la solución óptima. El pseudocódigo del algoritmo es el siguiente:

```
Función mochila ( ↓pesos ∈ Real Vector[1...n], ↓valores ∈ Real Vector[1...n], ↓W∈ Entero,
                  ↓ ↑ x ∈ Real Vector[1...n])
 Constantes
    Ø
 Variables
  i, suma ∈ Entero
 Acciones
  suma ← 0
 Para i←1 hasta n
  x[i] = 0
 Fin Para

  Mientras suma < W
   i ← SeleccionVoraz( x, pesos, valores )
   Si suma + pesos[i] ≤ W entonces
      x[i]← 1
      suma← suma + pesos[i]
   sino
      x[i]← (W-suma)/ pesos[i]
     suma ← W
   Fin Si
  Fin Mientras
  regresar ( x )
Fin Función mochila
```

Se reciben como datos los vectores con los pesos, con los valores de cada objeto y también el peso máximo W que soporta la mochila. La función mochila regresa como solución un vector con los coeficientes por los que se multiplicará el peso de cada objeto para obtener el peso total W. Primero se inicializa el vector solución x con cero. Posteriormente se hace una *selección voraz* para elegir al mejor objeto, siempre y cuando la suma no sea igual al peso W. Si al acumular en suma el peso del objeto *i* no se supera a W, entonces se incorpora a la suma la totalidad del objeto, es decir, el peso w_i multiplicado por el coeficiente 1, de lo contrario se incorpora a la suma el producto $x_i w_i$. Al observar detenidamente el algoritmo podría pensarse que se puede substituir suma ← W por suma ← suma + x[i]*pesos[i], sin embargo, hay que tomar en cuenta que al multiplicar el peso por el coeficiente x_i muy probablemente se obtendrá un número fraccionario que, al sumarlo con suma, será un valor muy cercano a W pero no exactamente igual (debido a la limitada precisión de las computadoras), lo que probablemente haría que se ejecute el ciclo hasta el infinito. Por lo anterior, forzamos a que suma tome el valor de W

La estrategia de la función de *selección voraz* puede variar. Aquí analizaremos tres. La primera estrategia consiste en elegir el objeto más valioso. La segunda consiste en seleccionar primero los objetos más ligeros y dejar los más pesados para el final. La tercera

estrategia es obtener la relación valor/peso y elegir en el orden de mayor a menor valor/peso. A continuación un ejemplo propuesto por Brassard & Bratley [2008].

Para n = 5 objetos y peso máximo que soporta la mochila W=100 kg:

	i =1	i =2	i =3	i =4	i =5	
Pesos:	10	20	30	40	50	kg
Valores:	20	30	66	40	60	pesos
Valor/peso:	2.0	1.5	2.2	1.0	1.2	pesos/kg

Como primera opción seleccionaremos el objeto más valioso entre los restantes. Esto nos lleva a elegir primero el objeto 3, con peso de 30kg, después el objeto 5 que pesa 50kg, como el peso acumulado es 80kg ya no cabe todo el objeto 4, que es el siguiente de mayor valor, por lo que agregamos solo la mitad del objeto 4: 0.5(40kg)=20kg. Esto nos da un valor de \$66+\$60+\$20 = \$146. En este caso el vector solución es x = {0, 0, 1, 0.5, 1}.

Con la segunda opción seleccionamos primero el objeto más ligero, en este caso, la mochila se llena con los primeros 4 objetos y ya no cabe el 5. En este caso el vector solución es x = {1, 1, 1, 1, 0}. El valor total de la mochila es de \$20+\$30+\$66+\$40 = \$156.

Finalmente, con la tercera opción seleccionamos primero el objeto 3 porque es el que tiene mayor relación valor/peso, en seguida el objeto 1, después el objeto 2. El objeto 5, con valor/peso = 1.2\$/kg ya no cabe, entonces, para obtener un peso total de 100kg multiplicamos el objeto 5 por 0.8, así tenemos W= 30kg+10kg+20kg+ (0.8)50kg =100kg. Con este método de selección, el valor final es de \$66+\$20+\$30+(0.8)\$60= \$164. En este caso el vector solución es x={1, 1, 1, 0, 0.8}.

Podemos concluir que la solución que obtenemos depende del tipo de *selección voraz* que utilicemos, además, la solución obtenida no es necesariamente la óptima. Para este caso particular, con la tercera estrategia si se obtiene la solución óptima.

Teorema: "Si se seleccionan los objetos por orden decreciente de la razón $valor_i / peso_i$, entonces el algoritmo de la mochila encuentra una solución óptima" [Brassard y Bratley, 2008].

Pregunta 1: ¿Cuál sería la mejor función de selección voraz en el caso en el que todos los objetos tuvieran el mismo valor?

Respuesta 1: Se seleccionarían primero los objetos más ligeros, para almacenar la mayor cantidad de objetos en la mochila.

Pregunta 2: ¿Cuál sería la mejor función de selección voraz en el caso en el que todos los objetos tuvieran el mismo peso?

Respuesta 2: habría que elegir primero los objetos más valiosos.

Ejercicio.- Resolver el problema de la mochila con los siguientes datos. Se tienen $n = 5$ objetos, el peso máximo que soporta la mochila es W=600kg y cada objeto tiene los siguientes valores y pesos:

	i =1	i =2	i =3	i =4	i =5	
Pesos:	50	100	150	200	250	kg
Valores:	20	24	55	40	70	$
Valor/peso:	0.4	0.24	0.33	0.2	0.28	$/kg

Solución.-

Como primera opción seleccionaremos el objeto más valioso entre los restantes. Esto nos lleva a elegir primero el objeto 5, con peso de 250kg, después el objeto 3 que pesa 150kg, después el objeto 4 que pesa 200kg, el peso acumulado es 600kg. Esto nos da un valor de $55+$40+$70 = $165. En este caso el vector solución es x = {0, 0, 1, 1, 1}.

Con la segunda opción seleccionamos primero el objeto más ligero, en este caso, la mochila se llena con los primeros 4 objetos y ya no cabe todo el objeto 5, por lo que agregamos solo una parte del objeto 4: 0.4*(250kg). En este caso el vector solución es x = {1, 1, 1, 1, 0.4}. El valor total de la mochila es de $20+$24+$55+$40+0.4($70) = $167.

Finalmente, con la tercera opción seleccionamos primero el objeto 1 porque es el que tiene mayor relación valor/peso, en seguida el objeto 3, después el objeto 5. El objeto 4, con valor/peso = 0.2 $/kg ya no cabe, entonces, para obtener un peso total de 600kg multiplicamos el objeto 4 por 0.25, así tenemos W= 50kg+100kg+150kg+ (0.25)200kg+250kg = 600kg. Con este método de selección, el valor final es de $20+$24+$55+ (0.25)*$40 +$70= $179. En este caso, el vector solución es x = {1, 1, 1, 0.25, 1}. Como podemos apreciar, esta tercera opción es la mejor solución.

A continuación se presenta una implementación en C++ (para DevC++) que utiliza la estrategia de elegir primero el mayor valor/peso.

```
#include <cstdlib>
#include <iostream>

using namespace std;

struct objeto{
   double peso;
   double valor;
   double valorPeso;
   bool tomado;
};

objeto objetos[5];

double solucion[]={0,0,0,0,0};

int SeleccionVoraz( ){
    //estrategia del mayor valor/peso
    double mayor = -1;
    int indice=-1, j=0;



    for(int j=0; j<5; j++){
       if(!objetos[j].tomado && (mayor < objetos[j].valorPeso)){
         mayor = objetos[j].valorPeso;
         indice = j;
       };
    };

    return indice;
};

double valorTotal(){
    double valor=0;

    for( int i=0; i<5; i++)
       valor = valor + objetos[i].valor * solucion[i];

    return valor;
};



int main(int argc, char *argv[]){

    double PesoMax, suma=0;
    int i;

    for( i=0; i<5; i++){
       cout<< "Peso del objeto "<< i << " ? = \n";
       cin >> objetos[i].peso;
       cout<< "Valor del objeto "<< i << " ? = \n";
```

```
        cin >> objetos[i].valor;
        objetos[i].valorPeso = objetos[i].valor/objetos[i].peso;
        objetos[i].tomado = false;
    };

    cout << "¿Peso maximo de la mochila?"; cin >> PesoMax;

    while(suma < PesoMax){
        i= SeleccionVoraz();

        if((suma + objetos[i].peso) < PesoMax ){
            solucion[i] = 1;
            suma = suma + objetos[i].peso;
            objetos[i].tomado = true;
        }
        else{
            solucion[i] = (PesoMax-suma)/ objetos[i].peso;
            suma = PesoMax; // corregir error al multiplicar por fracción
        };
    };

    cout << " el valor máximo obtenido es: " << valorTotal() << endl;
    cout << " y el vector solución es: \n";
    for(i=0; i<5; i++)
        cout<< " " << solucion[i] << ", ";

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

En la *Figura 5.1* se muestra este programa corriendo con los datos de este ejercicio.

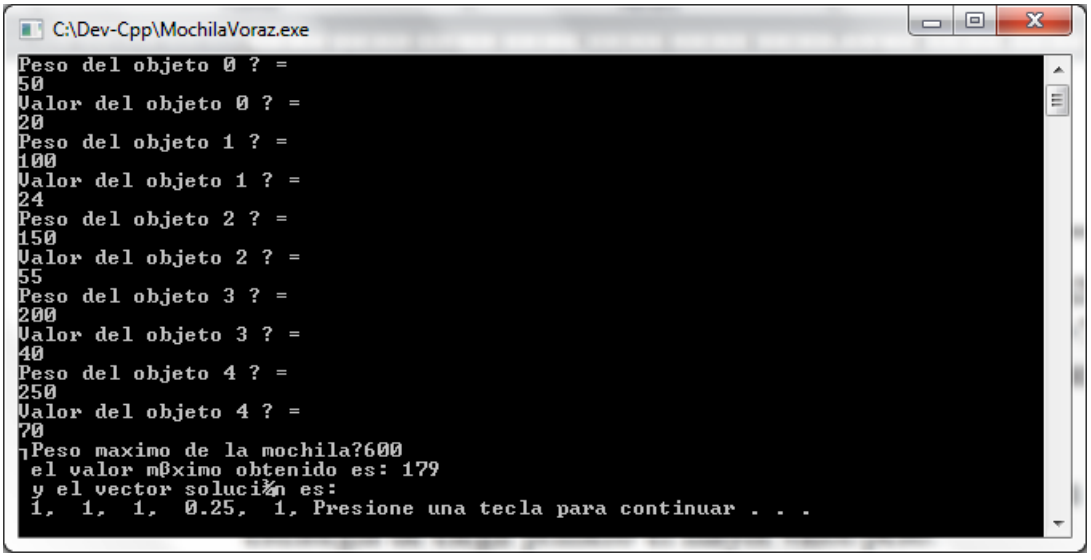


Figura 5.1: Obtención del valor máximo y el vector solución

V.4 El problema de la planificación de las tareas resuelto con un algoritmo voraz

Este es también un problema clásico en el estudio de los algoritmos. Supongamos que disponemos de *n* trabajadores y *n* tareas. Sea $c_{ij} > 0$ *el costo* de asignarle el trabajo *j* al trabajador *i* (puede ser dinero o tiempo). Una asignación de tareas puede expresarse como una asignación de los valores 0 ó 1 a las variables a_{ij}, donde $a_{ij} = 0$ significa que al trabajador *i* no le han asignado la tarea *j* y $a_{ij} = 1$ indica que sí. Una asignación válida es aquella en la que a cada trabajador sólo le corresponde una tarea y cada tarea está asignada a un trabajador. Dada una asignación válida, definimos el *costo total* de dicha asignación como:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} a_i c_{ij}$$

Que es la expresión a minimizar. Para lograrlo podemos seguir una de las siguientes estrategias:

1. Asignar a cada trabajador la mejor tarea posible (por ejemplo, la que hace por menos dinero).

2. Asignar cada tarea al mejor trabajador disponible (por ejemplo, al que cobra menos).

El costo de que el trabajador *i* realice la tarea *j* se indica en en una matriz de costos como la que se muestra en la *Figura 5.2*., en esta figura las tareas y los trabajadores se numeran del 0 a *n*-1, como se enumeran los índices de los arreglos y las matrices en C/C++.

	Tarea 0	Tarea 1	⋯	Tarea *n-1*
Trabajador 0	c_{00}	c_{01}	⋯	$c_{0,n-1}$
Trabajador 1	c_{10}	c_{11}	⋯	$c_{1,n-1}$
	⋮	⋮	⋱	⋮
Trabajador *n-1*	$c_{n-1,0}$	$c_{n-1,1}$	⋯	$c_{n-1,n-1}$

Figura 5.2: Matriz de costos

Suponiendo que conocemos *n,* el número de trabajadores y de tareas, el algoritmo para resolver el problema de la asignación de tareas tiene como entrada la matriz de costos, minimiza el costo total, y da como resultado un vector de asignaciones, tal como se ilustra en la *Figura 5.3*.

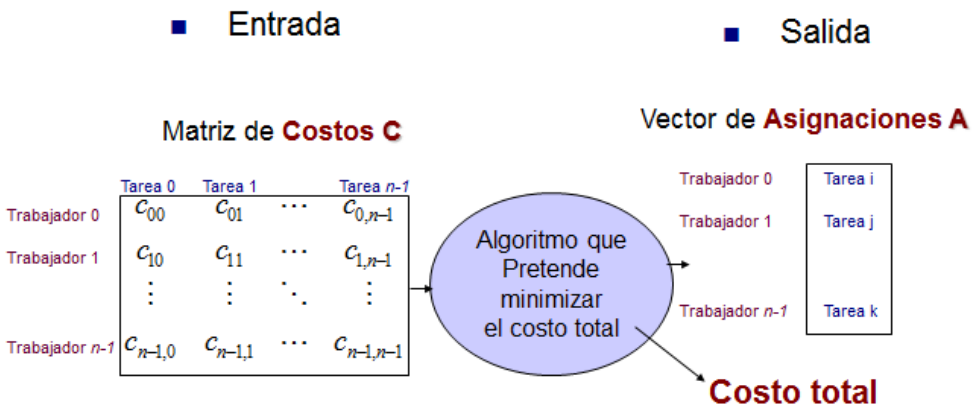


Figura 5.3: Entrada y salida del problema de la asignación de tareas

Este problema se aplica en la vida real a la asignación de máquinas-trabajadores, trabajadores-tareas, inversiones-ingresos, etc.

¿Cómo asigna un algoritmo voraz la mejor tarea posible a cada trabajador?

El algoritmo para resolver este problema se presenta a continuación. C es la matriz de costos, en este caso es de 5x5, y A es el vector de asignación (su tamaño es el número de trabajadores), en el algoritmo que aquí presentamos le llamaremos `tareas`. Primero se asigna la mejor tarea mediante el algoritmo de selección voraz `AsignaTarea`, la cual recibe como parámetro el índice del trabajador al que se le asignará la tarea y regresa el número de tarea que eligió. Posteriormente se despliega el costo mínimo encontrado en base a la tarea asignada a cada trabajador y su costo correspondiente.

```
Función tareasVoraz ( ↓Costos ∈ Real matriz[1...N, 1...N], ↑ costo ∈ Real )
    Constantes
        N=5
    Variables
      tareas ∈ Entero [1...N]
      trabajador, costoTotal ∈ Entero
    Acciones

     Para trabajador←1 hasta n
       tareas[trabajador] ← AsignaTarea(trabajador)
     Fin Para

     costoTotal ← calculaCosto()

   regresa costoTotal

Fin Función TareasVoraz
```

El algoritmo que asigna a cada trabajador la tarea más barata dentro de las que aún no han sido asignadas es el siguiente.

```
Función AsignaTarea ( ↓trabajador ∈ Entero,  ↑ tarea ∈ Entero )
    Constantes
       Maximo ← 1E10
    Variables
      tarea, mejorTarea ∈ Entero
      trabajador ∈ Entero
      min ∈ Real
    Acciones
     Min  ← Maximo
     Para tarea←1 hasta N hacer
       Si (NOT YaAsignada(tarea)) entonces
         Si ( Costo[trabajador][tarea] < min ) entonces
            Min ←  Costo[trabajador][tarea]
            mejorTarea ← tarea
          fin Si
        fin Si
      Fin Para

   regresa mejorTarea

Fin Función AsignaTarea
```

El siguiente algoritmo determina si la tarea ya fue o no asignada previamente a otro trabajador.

```
Función YaAsignada ( ↓tarea ∈ Entero, ↑ yaAsignada ∈ Booleano )
    Constantes

    Variables

    Acciones
     Para trabajador←1 hasta N hacer
       Si (tareas[trabajador] = tarea ) entonces
         Regresa TRUE
       fin Si
     Fin Para

   regresa FALSE

Fin Función YaAsignada
```

La mejor tarea seleccionada, siempre y cuando no esté asignada, es la de menor costo. El algoritmo de la función que determina si una tarea ya se asignó o no es el siguiente.

Una vez asignada una tarea a cada trabajador, solo resta calcular el costo mediante la siguiente función:

```
Función CalculaCosto ( ↑ suma ∈ Real )
    Constantes

    Variables
      suma ∈ Real
      tarea ∈ Entero

    Acciones
     suma← 0
     Para trabajador←1 hasta N hacer
       tarea ← tareas[trabajador]
       suma ← Costo[trabajador][tarea]
     Fin Para

   regresa suma

Fin Función CalculaCosto
```

El vector tareas contiene el número de tarea que se le asignó a cada trabajador, el costo se obtiene de la matriz de costos usando como índices el trabajador con su correspondiente tarea asignada.

V.5 El problema del agente viajero resuelto con un algoritmo voraz

Un agente viajero debe pasar por *n* ciudades diferentes. El objetivo es encontrar una ruta que, comenzando y terminando en una ciudad concreta, pase una sola vez por cada una de las ciudades y minimice la distancia recorrida (o el costo en que se incurre) por el viajante. Se conocen las distancias entre cada una de las ciudades, de tal manera que se puede construir un grafo pesado no dirigido en el que los vértices (o nodos) son las ciudades y el peso de las aristas es la distancia entre los vértices.

En la *Figura 5.4* se aprecian las partes de un grafo pesado no dirigido.

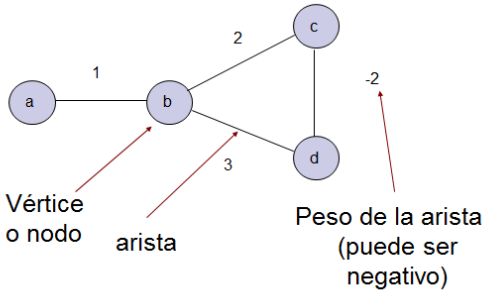


Figura 5.4: Grafos no dirigidos

Recordemos también que un *ciclo hamiltoniano* consiste en un camino que comienza en un nodo, que visita todos los demás nodos exactamente una vez y vuelve al nodo de partida.

El planteamiento formal del problema del agente viajero es el siguiente: "Encontrar, en un grafo no dirigido con pesos en las aristas, un recorrido que sea un ciclo hamiltoniano, donde la suma de los pesos de las aristas sea mínima."

Los vértices son las *ciudades* que debe visitar el agente, quien debe partir de una ciudad inicial y regresar a esta misma. En la *Figura 5.5* se muestran seis ciudades en una gráfica, con sus respectivas coordenadas.

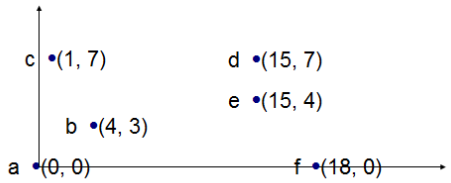


Figura 5.5: Posición de las seis ciudades

A partir de las coordenadas dadas en la *Figura 5.5* se puede generar el grafo como el de la *Figura 5.6*, el cual contiene algunas de las posibles trayectorias entre dos ciudades y sus respectivos pesos (o distancias).

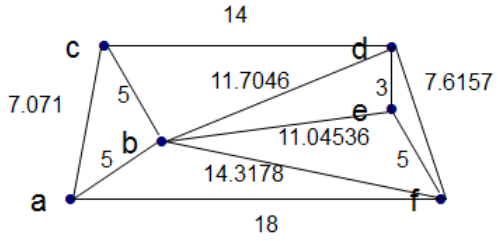


Figura 5.6: Grafo con seis nodos y las distancias entre ellos

Para este caso en particular, una vez que elegimos un nodo, el siguiente paso es elegir entre 5 opciones diferentes. En la siguiente etapa se deberá elegir uno de entre los 4 restantes, dando hasta el momento que se tienen 5*4 posibilidades. De la misma forma se irán eligiendo después de entre 3, 2 y 1 posibilidades en las etapas subsiguientes. El número total de posibilidades es de 5*4*3*2*1 = 120. Generalizando, para *n* ciudades, el número total de posibilidades es de (*n*-1)!, por lo tanto, resolver este problema para una *n* grande es computacionalmente prohibitivo ya que se deberían evaluar todas las posibilidades antes de decidir cuál es la mejor. Un algoritmo así es O(n!).

En la *Figura 5.7* se muestran 4 de los 5! recorridos diferentes que se pueden elegir partiendo del nodo **a**. ¿Cuál es el que elige un algoritmo voraz?

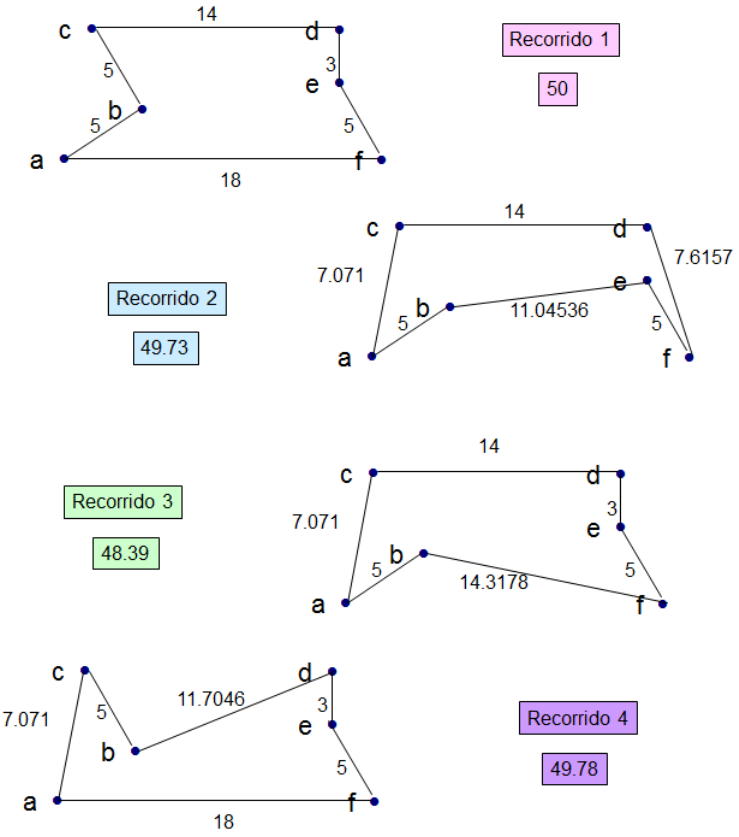


Figura 5.7: Diferentes recorridos partiendo del nodo a

De la *Figura 5.7* es fácil concluir que la distancia mínima entre estos recorridos es la del recorrido 3. Ahora veamos cómo procede un algoritmo ávido para encontrar la solución. En la *Figura 5.8* se muestran los primeros dos pasos. El algoritmo ávido elige ir de **a** a **b** ya que 5 es la distancia mínima. Posteriormente elige ir de **b** a **c** ya que, una vez más, 5 es la menor distancia.

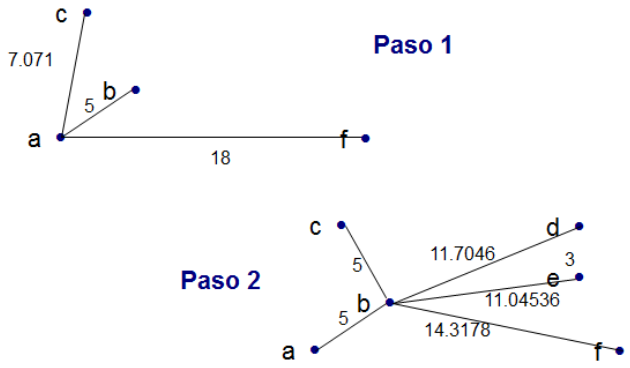


Figura 5.8: Los primeros dos pasos del algoritmo voraz

En la *Figura 5.9* se ilustran los siguientes cuatro pasos. Una vez elegido el camino **a**, **b**, **c**, la mejor opción es seguir al nodo **d**. La distancia menor es ahora de **d** a **e** y ahora solo queda por visitar el nodo **f** y regresar al nodo **a**, tal y como se ilustra en el paso 6.

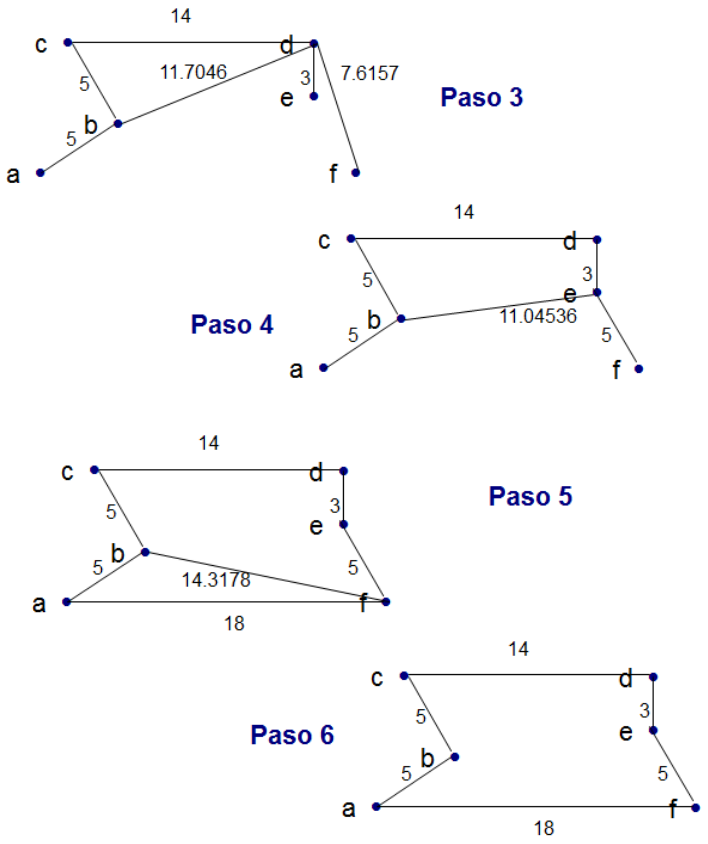


Figura 5.9: Los siguientes cuatro pasos del algoritmo voraz

El recorrido que eligió el algoritmo voraz no es el óptimo, ya que la distancia total es: 5+5+14+3+5+18= 50 50, pero, como habíamos visto en la *Figura 5.7*, existen por lo menos otros 3 recorridos de menor distancia.

Este problema es un ejemplo clásico en el estudio de la optimización computacional, porque es el tipo de problemas que parecen tener una solución sencilla, sin embargo, al llevar la solución a la práctica, ésta se vuelve extremadamente complicada conforme aumenta el tamaño del problema. Se conoce la forma de resolver de manera óptima el problema del agente viajero, pero sólo teóricamente, ya que es inoperable en la práctica.

Utilizamos la matriz de adyacencias `A`, para representar computacionalmente un grafo con *n* nodos en el que cada arista que va del nodo *i* al nodo *j* tiene un peso `A[i,j]`. Así por ejemplo, para representar el grafo de la *Figura 5-10* tenemos que la matriz de adyacencias es la siguiente:

	0	1	2	3
0	0	1	5	2
1	1	0	4	6
2	5	4	0	3
3	2	6	3	0

Tabla 5.1: Matriz de Adyacencias `A` para un grafo de 4 nodos

Nótese que A es una matriz simétrica ya que el peso entre el nodo *i* y el *j* se representa como A[i,j] ó como A[j,i]. Además, A[i,i] = 0 porque no hay nodos conectados con ellos mismos.

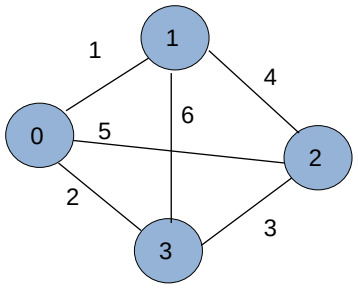


Figura 5.10: Grafo con 4 nodos y 6 aristas

El algoritmo para resolver el problema del agente viajero de la *Figura 5.10* con el paradigma voraz se presenta a continuación. Obtiene una trayectoria que tenga el menor costo (según un algoritmo voraz), se basa en una matriz de adyacencias `A` que contiene los pesos de las aristas entre los nodos que están conectados.

```
Función AgenteViajeroVoraz (↓A ∈ Entero [1...NumNodos, 1...NumNodos],
                            ↑ trayectoria ∈ Entero [1...NumNodos+1] )
    Constantes
      NumNodos 4
    Variables
      nodoActual, nodoSiguiente ∈ Entero
      trayectoria ∈ Entero[1...NumNodos+1]
      nodosAsignados ∈ Booleano[1...NumNodos]

    Acciones
     nodoActual ←0
     Para i←1 hasta NumNodos hacer
       nodosAsignados[nodoActual] ← true
       trayectoria[i] ← nodoActual
       nodoSiguiente← BuscaMejorNodo(nodoActual)
       nodoActual← nodoSiguiente
     Fin Para

     Mensaje "el costo total es"  CalculaCosto()
     trayectoria[NumNodos+1] ←0
   regresa trayectoria

Fin Función AgenteViajeroVoraz
```

El mejor nodo es aquel que tiene la arista de menor peso, siempre y cuando no haya sido ya tomado en cuenta previamente en la trayectoria.

```
Función BuscaMejorNodo (↓nodoActual ∈ Entero, ↑ mejorNodo ∈ Entero)
    Constantes
      Max 1E10
    Variables
      mejorNodo ∈ Entero
     menor ∈ Entero

    Acciones
     menor ← Max
     Para j←1 hasta NumNodos hacer
      Si (j ≠ nodoActual) entonces
          Si (A[ nodoActual][j] < menor  AND  NOT nodosAsignados[j]) entonces
            menor← A[nodoActual][j]
            mejorNodo← j
          fin Si
       fin Si
     Fin Para
  regresa mejorNodo

Fin Función BuscaMejorNodo
```

Finalmente, el siguiente algoritmo que calcula el costo basándose en la trayectoria y en el costo correspondiente en la matriz de adyacencias.

```
Función CalculaCosto(↑suma ∈ Real)
    Constantes
      NumNodos 4
    Variables
      suma ∈ real
      nodoActual, nodoSig ∈ Entero

    Acciones
     suma ← 0
     Para i←1 hasta NumNodos hacer
         nodoActual ← trayectoria[i]
         nodiSig← trayectoria[i+1]
         suma ← suma + A[nodoActual][nodoSig]
     Fin Para
  regresa suma

Fin Función CalculaCosto
```

V.6 Ejercicios

V.6.1 Planteamiento de los ejercicios

1. Haz un programa que se base en el algoritmo voraz presentado en la sección V.4 que resuelve el problema de la asignación de tareas.

2. Haz un programa que utiliza el algoritmo voraz presentado en la sección V.5 para resolver el problema del agente viajero.

3. Resuelve el problema de la asignación de tareas, pero ahora, en lugar de encontrar la mejor tarea para cada trabajador, encuentra el mejor trabajador para cada tarea.

V.6.2 Solución a los ejercicios

Solución el ejercicio 1

Programa principal:

```
#include <iostream.h>
#define N 5

float Costo[N][N], costoTotal; // C es la matriz de costos
int Tareas[N]; // matriz de asignación

// Asignación de tareas
main(){
  int trabajador;

  PedirCostos();// llena la matriz de costos

  // Asignar la mejor tarea
  for( trabajador=0; trabajador<n; trabajador++ ){
     Tareas[trabajador] = AsignaTarea(trabajador);
  }

  costoTotal = calculaCosto();
  cout << "Costo total:" << costoTotal;

  return 0;
}
```

Función para asignar la mejor tarea:

```
#define MAX_FLOAT 1E10

int AsignaTarea( int trabajador ){
  int tarea, mejorTarea;
  float min= MAX_FLOAT; // min con el valor máximo

  for( tarea=0; tarea<n; tarea++ ){
    if( !YaAsignada(tarea) )
      if( Costo[trabajador][tarea] < min ){
        min = Costo[trabajador][tarea];
        mejorTarea = tarea;
      };
  };
  return mejorTarea;
}
```

Función para determinar si la tarea ya fue asignada:

```
bool YaAsignada( int tarea ){
  for( int trabajador=0; trabajador<n; trabajador++ )
    if( tareas[trabajador] == tarea )
       return TRUE;

  return FALSE;
}
```

Función para calcular el costo:

```
float CalculaCosto(){
  float suma=0;
  for( int trabajador=0; trabajador<n; trabajador++ ){
    tarea = Tarea[trabajador];
    suma  = suma + Costo[trabajador][tarea];
  };
  return suma;
}
```

Solución al ejercicio 2

```
#include <cstdlib>
#include <iostream>

#define NUM_NODOS 4
#define MAX_VALUE 1E10

using namespace std;

float A[NUM_NODOS][NUM_NODOS];
int trayectoria[NUM_NODOS+1];
bool nodosAsignados[NUM_NODOS]={0,0,0,0};
int nodoActual, nodoSiguiente;

void CapturaMatrizAdyacencias(){
  for(int i=0; i<NUM_NODOS; i++)
    A[i][i] =0;

  for(int i=0; i<NUM_NODOS; i++){
    for(int j=i+1; j<NUM_NODOS; j++){
      cout<< "A["<<i<<", "<<j<<"]=? ";
      cin>> A[i][j];
      A[j][i]= A[i][j];
    };
  };
  return;
}

void DespliegaMAtrizA(){
  for(int i=0; i<NUM_NODOS; i++){
    for(int j=0; j<NUM_NODOS; j++){
      cout<<" "<< A[i][j];
    };
    cout<<endl;
  };
  return;
}

int BuscaMejorNodo(int nodoActual){
  float menor= MAX_VALUE;
  int mejorNodo;
  for(int j=0;j<NUM_NODOS;j++){
    if(j!=nodoActual){
      if(A[nodoActual][j] < menor && !nodosAsignados[j] ){
        menor = A[nodoActual][j];
        mejorNodo = j;
      };
    };
  };
  return mejorNodo;
}

float calculaCosto(){
  float suma = 0;
```

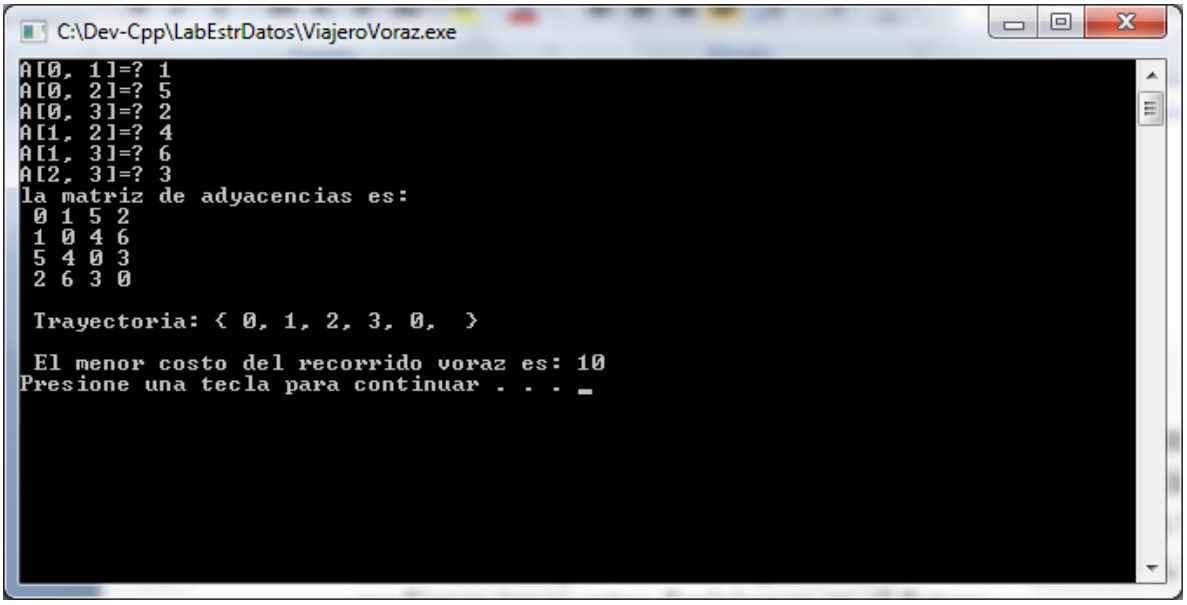
```
  int nodoActual, nodoSig;
  for( int i=0; i< NUM_NODOS; i++){
    nodoActual = trayectoria[i];
    nodoSig = trayectoria[i+1];
    suma = suma + A[nodoActual][nodoSig];
  };
  return suma;
}


int main(int argc, char *argv[]){
  CapturaMatrizAdyacencias();
  cout<< "la matriz de adyacencias es: \n";
  DespliegaMAtrizA();
  nodoActual=0;
  for(int i=0;i<NUM_NODOS;i++){
    nodosAsignados[nodoActual]=1;
    trayectoria[i]=nodoActual;
    nodoSiguiente = BuscaMejorNodo(nodoActual);
    nodoActual = nodoSiguiente;
  };
  trayectoria[NUM_NODOS+1]=0;
  cout << "\n Trayectoria: { ";
  for(int i=0; i< NUM_NODOS+1;i++)
    cout << trayectoria[i]<<", ";
  cout<<" }"<<endl;

  cout<<"\n El menor costo del recorrido voraz es: "<< calculaCosto()
        << endl;

  system("PAUSE");
  return EXIT_SUCCESS;
}
```

Al ejecutar este programa obtenemos lo siguiente:



V.7 Resumen del capítulo

En el capítulo V estudiamos los algoritmos voraces. Estos algoritmos se aplican principalmente a problemas de optimización, es decir, problemas en los que hay que maximizar o minimizar algo. Los algoritmos voraces toman decisiones basándose en la información que tienen disponible de forma inmediata, sin tomar en cuenta los efectos que pudieran tener estas decisiones en el futuro.

En este capítulo aplicamos la técnica de los algoritmos voraces para resolver varios ejemplos: el problema del cambio, el problema de la mochila, el problema de la planificación de tareas y el problema del agente viajero. A continuación, en el capítulo VI estudiaremos la técnica de programación dinámica.

Capítulo VI Programación dinámica

VI.1 Definición del paradigma de la programación dinámica

La *programación dinámica* se utiliza para mejorar el rendimiento de algunos algoritmos. En ciertos casos, cuando se divide un problema en varios subproblemas, resulta ser que éstos no son independientes entre sí. Es decir, como se aprecia en la *Figura 6.1*, cuando se resuelve un subproblema traslapado con otros, se repite una parte de la solución que ya se encontró al resolver otro subproblema.

Cuando se pretende subdividir un problema con la técnica divide y vencerás y los subproblemas no son independientes entre sí, es muy probable que el tiempo para hallar la solución crezca de forma exponencial con el tamaño de la entrada. Es decir, si no hay manera de dividir el problema en un pequeño número de subproblemas independientes entre sí, tendremos que resolver el subproblema varias veces, lo que quizá produzca un algoritmo de tiempo exponencial.

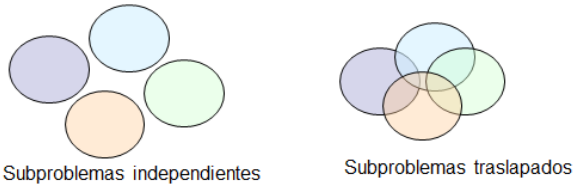


Figura 6.1: Posibles resultados de la división de un problema

Es obvio que solucionar el mismo problema varias veces no es eficiente. La programación dinámica consiste en conservar la solución a cada subproblema que ya se ha resuelto en una tabla (puede ser un arreglo o una matriz), para tomarla cuando ésta se requiera. Esto reduce el tiempo necesario para obtener el resultado final, pues evita repetir algunos de los cálculos. Entonces, una característica del paradigma de la programación

dinámica es que se utilizan tablas como una estructura auxiliar para evitar el solapamiento, es decir, evitar calcular varias veces un mismo resultado.

A continuación planteamos los pasos para el diseño de un algoritmo de programación dinámica:

1.- Plantear un algoritmo de solución recursivo.

2.- Organizar las soluciones parciales en una tabla.

3.- Modificar el algoritmo recursivo de manera que antes de hacer el llamado recursivo se consulte la tabla y, en caso de hallarse la solución en ésta, ya no se hace el llamado recursivo.

"En la programación dinámica normalmente se empieza por los subcasos más pequeños, y por tanto más sencillos. Combinando sus soluciones, obtenemos las respuestas para subcasos de tamaños cada vez mayores, hasta que finalmente llegamos a la solución del caso original" [Brassard & Bratley, 2008].

VI.2 Cálculo de los números de Fibonacci con programación dinámica

La expresión recursiva de la función Fibonacci es la siguiente:

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

A continuación se muestra el pseudocódigo de la función recursiva de Fibonacci, la cual tiene un tiempo de ejecución $T(n)$ de orden exponencial. Los llamados a FibonacciRecursivo (n-1) + FibonacciRecursivo (n-2) hacen que el algoritmo sea muy poco eficiente.

```
Procedimiento FibonacciRecursivo (↓n ∈ Entero)→∈ Entero
 inicio
  Si n <=1 Entonces
    regresar 1
  Si no
    regresar (FibonacciRecursivo (n-1) + FibonacciRecursivo (n-2) )
  Fin Si
 Fin FibRec
```

Sin embargo, si generamos una tabla de programación dinámica, como la de la *Figura 6.2*, para ir guardando los resultados parciales, entonces el algoritmo será mucho más eficiente, pues en lugar de hacer dos llamados recursivos solo sacarán dos resultados de la tabla cuando estos ya han sido calculados.

Figura 6.2: Tabla de programación dinámica

El algoritmo dinámico es de orden **O**(*n*) y es el siguiente:

```
Tabla [1.....] ∈ Entero = {0, 0, 0, ..., 0}
Procedimiento FibonacciDinamico(↓n ∈ Entero)→∈ Entero
 inicio

  Si n =0 OR n=1  Entonces
    regresar 1
  Fin Si

  Si Tabla[n] = 0
    Tabla[n] = FibonacciDinamico(n -1) + FibonacciDinamico(n -2)
  Fin Si

  regresar Tabla[n]

 Fin FibDinam
```

Si bien en este algoritmo hay recursión, ésta solo se lleva a cabo cuando no hay datos en la tabla. Cada vez que se requiere un nuevo dato en la tabla se hace la recursión, sin embargo ésta no es muy profunda, ya que toma los datos que ya se encuentran calculados en la tabla sin necesidad de llegar cada vez al caso base.

Mientras que en los algoritmos ávidos una vez que se toma una decisión se descartan para siempre las demás opciones, la programación dinámica va *guardando la historia* y

construyendo la solución a partir de las soluciones óptimas de problemas más pequeños resueltos previamente.

En la programación dinámica el problema debe poder dividirse en varias etapas, y la decisión en una etapa se toma después de haber considerado los resultados de otras etapas más simples.

VI.3 El problema de apuestas con programación dinámica

Se requiere saber la probabilidad de que uno de dos equipo A y B gane *n* partidos en la serie mundial de beisbol, suponiendo que los equipos son igualmente competentes, de modo que cada uno de ellos tiene el 50% de probabilidad de ganar un partido.

Estudiaremos el caso particular donde *n* = 4 partidos. Llamaremos matriz P a la tabla de programación dinámica de la *Figura 6.3*, la cual representa la probabilidad de que gane el equipo A.

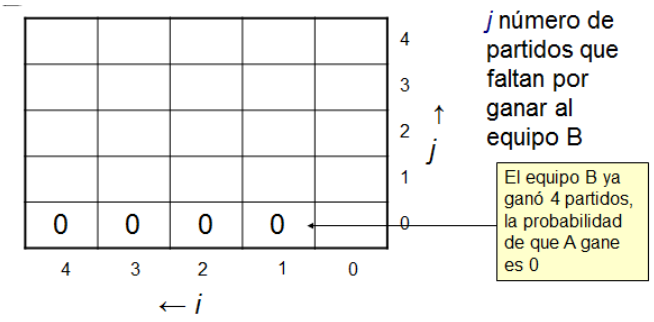


Figura 6.3: Matriz P de probabilidad de que gane el equipo A

P(i,j) es la probabilidad de que el equipo A gane 4 partidos dado que ya ganó (4-i) y que el equipo B ya ganó (4-j).

j es el número de partidos que faltan por ganar al equipo B.

i es el número de partidos que le faltan por ganar al equipo A

En el primer renglón de abajo para arriba de la matriz P de la *Figura 6.3 j*=0. Esto significa que el equipo B ya ganó 4 partidos, por lo tanto la probabilidad de que A gane cuatro partidos es 0.

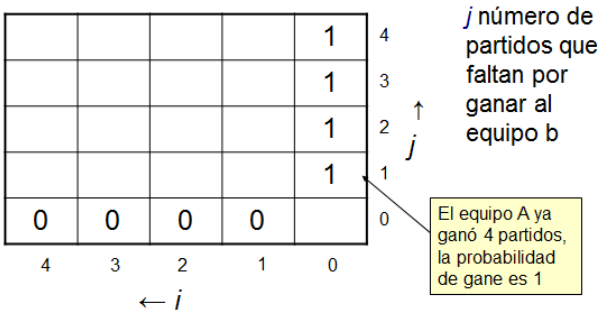


Figura 6.4: Llenado de la última columna de la Matriz P

En la *Figura 6.4*, en la última columna de la matriz P *i*=0, esto significa que el equipo A ya ganó 4 partidos, por lo tanto la probabilidad de que A gane es 1. Si A perdió 3 partidos y ganó otros 3, la probabilidad de que gane A es el promedio de la probabilidad de que A gane 4 partidos P [0,1]=1 y la probabilidad de que A pierda 4 partidos P[0,0] = 0, es decir, (1+0)/2 = 1/2. De la misma manera, si A perdió 2 partidos y ganó 3, la probabilidad de que gane A es el promedio: (1/2+1)/2 = ¾, para calcular la probabilidad si A perdió 0 partidos y B no ha ganado, la probabilidad de que gane A los 4 partidos es el promedio: (7/8+1)/2= 15/16. En la *Figura 6.5* se observa que la probabilidad de que gane el equipo A es (P[i][j-1] + P[i-1][j])/2.

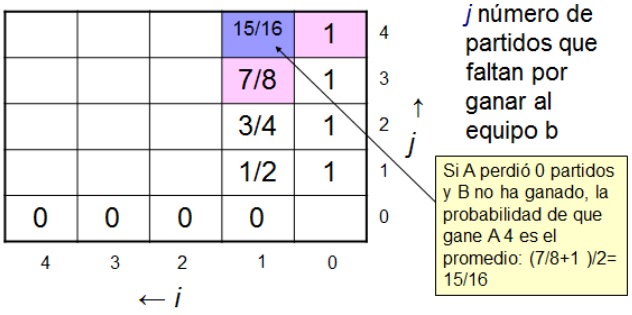


Figura 6.5: La probabilidad de que gane el equipo A es el promedio de P[i][j-1] y P[i-1][j]

En la *Figura 6.6* podemos observar la matriz P completa. El resto de la tabla se construyó de la misma forma que la segunda columna de P en la *Figura 6.5*.

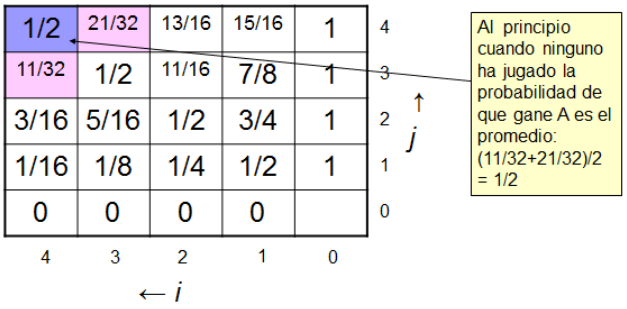


Figura 6.6: Matriz P de probabilidad de que gane el equipo A

Al principio, cuando aún no han jugado, es decir, para i =4 y j =4, la probabilidad de que gane A es el promedio: (11/32+21/32)/2 = 1/2.

A continuación se presenta el código en C/C++ para llenar la matriz P.

```
// La matriz se inicializa con -1 en cada posicion
float P[5][5] = {{-1,-1,-1,-1,-1},
                 {-1,-1,-1,-1,-1},
                 {-1,-1,-1,-1,-1},
                 {-1,-1,-1,-1,-1},
                 {-1,-1,-1,-1,-1}
                };

float Prob(int i, int j){
  if(i==0 && j>0)
    return 1;

  if(i>0 && j==0)
    return 0;

  if(P[i][j]==-1)
    P[i][j]=(Prob(i-1,j)+Prob(i,j-1))/2;

  return P[i][j];
}
```

Nótese que el procedimiento se llama varias veces con los mismos valores para i y j, sin embargo, la recursión solo se hace cuando no se cuenta con el resultado, esto reduce enormemente la complejidad del algoritmo. En el peor de los casos, cuando la tabla está vacía, la complejidad es **O**(i j), lo que implica que es cuadrática. Conforme se va llenando la tabla el tiempo de ejecución se va reduciendo hasta **O**(1). Hay que tomar en cuenta que en la programación dinámica se reduce la complejidad temporal a cambio de un aumento en la complejidad espacial, ya que las tablas que guardan la información requieren memoria.

Nota: la matriz que guarda los resultados parciales (y así evitar recursiones repetitivas) puede ser alojada conforme se va necesitando para evitar desperdiciar memoria que probablemente nunca será usada.

Puede observarse que el código anterior es **O**(n^2). [Aho et al., 1998] hacen un análisis matemático de la función recursiva (sin programación dinámica) del problema original y encuentran que la complejidad es **O**($\frac{2^n}{\sqrt{n}}$). Ésta función está dada por:

$$P(i,j) = 1 \text{ si } i = 0 \text{ y } j > 0$$

$$= 0 \text{ si } i > 0 \text{ y } j = 0$$

$$= (P(i\text{-}1, j) + P(i, j\text{-}1))/2$$

El orden obtenido con la programación dinámica es **O**(n^2), que es mucho mejor que el resultado obtenido con la forma recursiva **O**($\frac{2^n}{\sqrt{n}}$).

En suma, cuando se obtienen subproblemas que no son independientes entre sí, entonces una solución recursiva no resulta eficiente debido a la repetición de cálculos, la cual con frecuencia crece de forma exponencial.

Cuando el tiempo de ejecución de un algoritmo recursivo es exponencial, podría, en algunos casos, encontrarse una solución más eficiente mediante la programación dinámica. La programación dinámica consiste en resolver los subproblemas una sola vez, guardando sus soluciones en una tabla para utilizarlas posteriormente.

VI.4 El problema del cambio con programación dinámica

Dada una cantidad *n*, y un conjunto de monedas de cierta denominación, por ejemplo, d_1, d_2 y d_3 se desea elegir la mínima cantidad de monedas cuya suma sea *n*.

Este problema ya se resolvió en el capítulo V con los algoritmos voraces, sin embargo vimos que estos algoritmos no siempre encuentran la solución óptima. Ahora resolveremos este mismo problema construyendo una tabla de programación dinámica. La matriz C contiene un renglón por cada denominación diferente y una columna para cada cantidad de monedas, en este caso, desde 0 hasta 8 monedas.

Cada elemento de la matriz C[*i, j*] es el mínimo número de monedas que se necesitan para pagar una cantidad *j* cuando solo se utilizan monedas de la denominación d_i y denominaciones menores. Por ejemplo, como se aprecia en la *figura. 6.7*, para pagar 1 peso solo se puede usar una moneda de 1 (d_1=1) y para pagar desde 2 hasta 8 pesos se debe se debe pagar con la misma cantidad de monedas de 1, por lo tanto las cinco primeras columnas y todo el primer renglón de C son las que se indican en la *Figura 6.7*.

Denominación/ Cantidad	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1				
$d_3 = 6$	0	1	2	3	1				

Figura 6.7: para pagar la cantidad c, $0 \leq c \leq 8$ con monedas de 1, se requieren c monedas

C[2, 4] = 1 ya que con una moneda de 4 (d_2=4) se cubre la cantidad de 4, lo mismo sucede para C[3,4].

Si la cantidad *j* es más pequeña que la denominación d_i entonces, C[*i, j*]= C[*i*-1, *j*], pero si $j > d_i$, entonces habrá que elegir la menor de dos opciones:

1. Usar al menos una moneda de la denominación d_i. Una vez que hayamos entregado la primera moneda de esta denominación, quedarán por pagar $j - d_i$ unidades y esta cantidad puede cubrirse con C[*i, j*- d_i]. El total de monedas será entonces 1+ C[*i, j*- d_i].
2. En lugar de la opción anterior pagar con C[*i*-1, *j*] monedas.

Como debemos elegir la mínima cantidad de monedas, entonces C[*i, j*] queda determinado por:

$$C[i, j] = \min(C[i-1, j], 1+ C[i, j- d_i])$$

La *Figura 6.8* muestra la tabla completa la cual se llenó con este criterio.

Denominación/ Cantidad	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	4	2
$d_3 = 6$	0	1	2	3	1	2	1	2	2

Figura 6.8: Tabla de programación dinámica para el problema del cambio

Así, por ejemplo, si queremos determinar cuántas monedas se necesitan cuando se cuenta con las 3 denominaciones para pagar la cantidad de 7, se debe elegir entre pagar con monedas de 1 y 4, es decir, el dato en C[2,7] o con una moneda de 6 y las necesarias de menor denominación para completar el 7, es decir, 1+ C[3,1], entonces hay que elegir:

$$C[3,7] = \min(C[3-1, 7], 1+ C[3, 7- 6])$$

$$= \min(C[2,7], 1+ C[3,1]) = \min (4, 1+1)$$

$$= \min(4, 2) = 2$$

Entonces, para determinar el número mínimo de monedas que se usa para completar la cantidad *j*, solo será necesario determinar cuáles son las denominaciones con las que se cuenta y extraer el dato de C[*i, j*].

A continuación se presenta el algoritmo de una función que resuelve el problema del cambio mediante programación dinámica.

```
función cambioProgDinamica ( ↓monto ∈ Entero, ↑ numBilletes ∈ Entero)
 Constantes
   MontoMax ← 8
   NumDen ← 3
   denominaciones ←{ 1,4,6 }

 Variables
   suma, iDen ∈ Entero
   numBilletes←0
   C[NumDen][ MontoMax]

 Acciones

   Para j← 0… MontoMax Hacer
     C[0,j] ← j
   Fin Para

   Para i← 1… NumDen Hacer
      Para j← 0… MontoMax Hacer
         Si j < denominaciones[i] entonces
             C[i,j] ← C[i-1,j]
          sino
             C[i,j] ← min( C[i-1,j],  1+C[1, j-denominaciones[i]] )
          fin Si
       Fin Para
   Fin Para
   Regresar(denominaciones)
Fin Función cambioProgDinamica
```

El siguiente código en C++ implementa el algoritmo anterior usando el programa principal en lugar de una función:

```
#define MONTO_MAX 8

int C[ NUM_DEN ][ MONTO_MAX+1 ];
int denominaciones[ NUM_DEN ] = { 1, 4, 6 };
int numBilletes[ NUM_DEN ];

void imprimirC(){
  for( int i=0; i<NUM_DEN; i++ ){
    cout << denominaciones[i] << " - ";
    for( int j=0; j<=MONTO_MAX; j++ )
      //cout<< "C["<<i<<"]["<<j<<"]="<< C[i][j] << " ";
      cout << " " << C[i][j];
    cout<<endl;
  };
  cout<<endl;
}

int main( int argc, char *argv[] ){
  for( int j=0; j<=MONTO_MAX; j++ )
    C[0][j] = j;

  for( int i=1; i<NUM_DEN; i++ )
    for( int j=0; j<=MONTO_MAX; j++ )
      if (j< denominaciones[i])
        C[i][j] = C[i-1][j];
      else
        C[i][j] = min( C[i-1][j], 1 + C[i][j-denominaciones[i]] );

  imprimirC();

  int monto, suma=0, menor, iDen;
  cout << "introduce un monto (entre 1 y 8) ";
  cin>>monto;

  cout << "Hay 3 denominaciones, hasta cual vas a usar?";
  cin>>iDen;
  cout<<" el cambio se da con " << C[iDen-1][monto]<< " monedas"<<endl;

  system("PAUSE");
  return EXIT_SUCCESS;
}
```

VI.5 El problema de la mochila con programación dinámica

Como vimos en el ejemplo 2 de la sección VI.1, el problema consiste en llenar una mochila que soporta un peso máximo W de manera que contenga un valor máximo. Se tienen *n* objetos, cada uno de estos objetos tiene un peso w_i y un valor v_i ($i = 1, 2, \ldots, n$).

La *función objetivo* es el valor total de los objetos que están en la mochila. Una solución es factible siempre que se cumpla la restricción del peso máximo que soporta la mochila. Cuando la suma de todos los pesos de los objetos a elegir es menor o igual a W, el problema es inexistente. Asumiendo que la suma de los pesos sobrepasa W (para que el caso sea interesante) la solución óptima estará restringida por:

$$\sum_{i=1}^{n} x_i w_i = W$$

x_i toma los valores 0 cuando el objeto *i* no se incluye, y 1 cuando se incluye.

Cuando los objetos se pueden romper en trozos más pequeños al multiplicarlos por una fracción x_i, de tal manera que el peso del objeto *i* sea $x_i w_i$, se puede usar un algoritmo voraz. Sin embargo, el problema se complica cuando los objetos no se pueden romper. Brassard y Brattley (2006) ponen el ejemplo del caso en el que el peso máximo que soporta la mochila es 10 y se tienen 3 objetos con los siguientes pesos y valores: $w_1 = 6$ $v_1 = 8$, $w_2 = 5$ $v_2 = 5$, y $w_3 = 5$ $v_3 = 5$. Para este caso, el algoritmo voraz elige el primer objeto, ya que es el que tiene mayor valor por peso, sin embargo ya no puede colocar otro más porque no se puede partir, entonces la solución no es óptima.

Construir una tabla de programación dinámica ayuda a encontrar una mejor solución cuando los objetos no se pueden partir y tampoco se repiten. En la *Figura 6.9* tenemos una matriz M en la que hay un renglón por cada objeto, ordenados del más ligero al más pesado, y una columna por cada peso máximo que soporta la mochila, en este ejemplo va de 0 a 11 unidades de peso. En el primer renglón se indica el valor máximo que se obtiene al incluir solamente el objeto 1 con $w_1 = 1$ y valor $v_1 = 1$. En la primera columna el valor es de cero, ya que no se incluye algún objeto para un peso máximo de 0. En la segunda columna, el valor máximo es 1, ya que se incluye el objeto 1 con $w_1 = 1$ y $v_1 = 1$. En las columnas del primer renglón, solo se puede incluir el objeto 1.

Límite de peso/ Objetos	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1$ $v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2$ $v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5$ $v_3 = 18$	0	1	6	7	7							
$w_4 = 6$ $v_4 = 22$	0	1	6	7	7							
$w_5 = 7$ $v_5 = 28$	0	1	6	7	7							

Figura 6.9: Matriz M[*i*,*j*] con el valor máximo para completar el peso *j*

En el segundo renglón, tercera columna, la mejor opción es tomar el objeto 2, $w_2 = 2$ y valor $v_2 = 6$, para llenar un peso máximo de 2. Los otros renglones también tienen el valor 6,

ya que los demás objetos no caben si el peso máximo es 2. En la cuarta y quinta columnas, como solo se puede tomar un objeto de cada uno y los demás objetos no caben, entonces el valor máximo es 7, que es el resultado de tomar los objetos $w_1 = 1$ con valor $v_1 = 1$ y $w_2 = 2$ con valor $v_2 = 6$. El segundo renglón de la matriz es 7 a partir de la cuarta columna, ya que no se puede aumentar el valor cuando se tienen solo los primeros dos objetos.A continuación explicaremos el método general para obtener el resto de los resultados. Nótese que cuando se toma un objeto *i* éste ya no podrá incluirse nuevamente, es decir, solo hay uno de cada uno.

Tenemos dos opciones para llenar la celda M[*i*,*j*]. Llamémosle opción a) a dejar los objetos de menor peso que ya se tenían, es decir M[*i*-1, *j*] y dejar fuera al objeto *i*. Y sea la opción b) substituir estos objetos con el objeto del renglón *i*, para esta opción habría que llenar el peso que falta para completar el peso máximo *j*. Es aquí donde se reducen los cálculos, ya que el peso que falta por completar es $j - w_i$. Este subproblema ya está resuelto en alguna de las columnas anteriores. Por ejemplo, si elegimos j = 6 y tomamos el objeto $w_3 = 5$ con $v_3 = 18$, entonces, como ya tenemos 5 y el peso máximo es 6, ahora debemos conocer el caso $j = 1$ con $i = 2$, el cual ya está calculado y tiene valor máximo 1, por lo que para obtener el valor de la opción b) debemos sumar a $v_3 = 18$ un 1. Por último, hay que elegir la que tenga mayor valor de las opciones a) ó b). Ver la *Figura 6.10.*

Límite de peso/ Objetos	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1$ $v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2$ $v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5$ $v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6$ $v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7$ $v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	40

Figura 6.10: Matriz M[*i*,*j*] con el valor máximo para completar el peso *j*

a).- Obtener el valor cuando se toman los objetos anteriores

M[i,j]=M[*i*-1, *j*] = M[3-1, 6] = M[2,6] = 7

b).- Obtener el valor cuando se toma el nuevo objeto (el del renglón *i*) más el valor de los objetos que faltan para completar el peso *j*, esto ya está calculado en el renglón *i*-1, coulmna $j - w_i$.

M[i,j]=v_i + M[*i*-1, $j - w_i$]= v_3 + M[2,6-5] =18 + M[2,1] = 18+1=19

La fórmula general se obtiene calculando el máximo valor de los casos a) y b):

max(M[*i*-1, *j*], v_i + M[*i*-1, $j - w_i$])

para este caso en particular: max(7, 19) = 19

Otro ejemplo. Si elegimos j = 7 y tomamos el objeto $w_4 = 6$ con $v_4 = 22$, entonces, como ya tenemos 6 y el peso máximo es 7, ahora debemos buscar el caso $j = 1$, el cual ya esta calculado previamente y tiene valor máximo 1. Para obtener el valor de la opción b) debemos sumar a $v_4 = 22$ un 1. Por último, hay que elegir la que tenga mayor valor de las opciones a) y b).

a).- M[i,j]= M[i-1, j] = M[4-1, 7] = M[3,7] = 24

b).- M[i,j]= v_i + M[i-1, $j - w_i$]= v_4 + M[3,7-6] =22 + M[3,1] = 22+1=23

Ahora debemos determinar:

max(M[i-1, j], v_i + M[i-1, $j - w_i$]) = max (24, 23) = 24

Entonces, aunque en M[4,7] caben los objetos 1 y 4, decidimos dejar los objetos 2 y 3, ya que estos producen un mayor valor.

Similarmente, para el último elemento M[5,11] tenemos:

a).- M[i,j]= M[i-1, j] = M[5-1, 11] = M[4,11] = 40

b).- M[i,j]= v_i + M[i-1, $j - w_i$]= v_5 + M[4,11-7] =28 + M[4,4] = 28+7=35

M[5,11] = max(M[i-1, j], v_i + M[i-1, $j - w_i$]) = max (40, 35) = 40

La matriz M permite determinar cuál es el valor máximo que se puede tener en la mochila y también cuales son los objetos que componen esta carga con valor máximo. El procedimiento para identificar estos objetos es el siguiente: en nuestro ejemplo tenemos el valor máximo en M[5,11] = 40 y en M[4,11] = 40. Analizando M[5,11] vemos que M[5,11] $\neq v_5$ + M[4, 11 – w_5], es decir, 40≠28+M[4,4] = 35, lo que significa que no se agregó el objeto 5, sino que se eligió la opción de quedarse con los objetos para M[4,11], entonces analizaremos esta opción. Como M[4,11] = v_4 + M[3, 11 – w_4] = 22+M[3,5] =40, entonces la carga óptima se obtuvo agregando el objeto 4 a los objetos elegidos en M[3,5]. Para determinar éstos objetos, M[3,5]= v_3 + M[2, 5 – w_3]=18+M[2,0]=18, de aquí observamos que solo se agregó el objeto 3 y ningún otro más. En conclusión, el valor máximo se obtuvo agregando a la mochila los objetos 3 y 4, los cuales constituyen la carga óptima.

A continuación presentamos el algoritmo de la función que calcula la matriz M:

```
función MochilaProgDinamica ( ↓objetoMax ∈ Entero, ↓peso ∈ Entero, ↑ maxValor ∈ Entero)
 Constantes
   NumObjetos ← 5
   PesoMax ← 11
   W←{1, 2, 5, 6, 7}
   V ←{1, 6, 18, 22, 28}

 Variables
   suma, iDen ∈ Entero
   numBilletes←0
   C[NumDen][ MontoMax]

 Acciones

   Para i← 0… NumObjetos-1 Hacer
     M[ i,0 ] ← 0
   Fin Para
   Para j← 1… PesoMax Hacer
     M[ 0, j ] ← 1
   Fin Para

   Para i← 1… NumObjetos-1 Hacer
      Para j← 1… PesoMax Hacer
         Si W[i] > j entonces
              M[ i j ] ← M[ i-1, j ]
          sino
              M[i][j] ← max( M[ i-1, j ], V[i] + M[ i-1, j-W[i] ] )
          fin Si
       Fin Para
   Fin Para
   Regresar( M[objetoMax,peso] )
Fin Función MochilaProgDinamica
```

El código en C++ que genera la matriz M es el siguiente:

```
#include <cstdlib>
#include <iostream>

#define NUM_OBJETOS 5
#define PESO_MAX  11

using namespace std;

int M[NUM_OBJETOS][PESO_MAX+1];
int W[] = {1, 2, 5, 6, 7};      // Vector de pesos de los objetos
int V[] = {1, 6, 18, 22, 28};   // Vector de valores de los objetos

void imprimeMatriz(){
     for(int j=0; j<=PESO_MAX; j++)
         cout<< j << "   ";
     cout<< endl << endl;
     for(int i=0; i<NUM_OBJETOS; i++){
        for(int j=0; j<=PESO_MAX; j++){
           cout<< M[i][j] << "   ";
        };
        cout << endl;
     };
     return;
}

int main(int argc, char *argv[]){
    for(int i=0; i<NUM_OBJETOS; i++)
       M[i][0] = 0;

    for(int j=1; j<=PESO_MAX; j++)
       M[0][j]= 1;

    for(int i=1; i<NUM_OBJETOS; i++){
        for(int j=1; j<=PESO_MAX; j++){
           if( W[i] > j)
             M[i][j] = M[i-1][j];
           else
             M[i][j] = max( M[i-1][j], V[i] + M[i-1][j-W[i]] );
        };
    };

    imprimeMatriz();


    system("PAUSE");
    return EXIT_SUCCESS;
}
```

VI.6 Los coeficientes binomiales con programación dinámica

El teorema de Newton dice que el resultado de un binomio elevado a la *n*-ésima potencia tiene la siguiente forma.

$$(1+x)^n = 1 + \binom{n}{1}x + \binom{n}{2}x^2 + \cdots + \binom{n}{n-1}x^{n-1} + x^n$$

Donde las combinaciones se calculan de la siguiente manera:

$$\binom{n}{r} = \frac{n!}{r!\,(n-r)!}$$

A los valores $\binom{n}{r}$ se les conoce como *los coeficientes binomiales*, que pueden calcularse cada vez con la formula anterior o bien construir el *triángulo de Pascal*. Este famoso triángulo es una tabla de programación dinámica, en la que los cálculos anteriores sirven para encontrar los valores siguientes. En la *Figura 6.11* se muestra la estructura del triángulo de Pascal.

n/r	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

Figura 6.11: Triángulo de Pascal

Si llamamos TP a la tabla de la *Figura 6.11*, tenemos que:

TP[*n*,*r*] = TP[*n*-1,*r*-1] + TP[*n*-1,*r*] para 0< *r* < *n*

TP[*n*,*r*] = 1 para *r* = 0 y para *r*= *n*

TP[*n*,*r*] = 0 para todos los demás casos.

Cuando se obtienen los datos de la tabla TP evitamos hacer el cálculo de $\binom{n}{r}$ en cada ocasión. Para $0 \leq r \leq n$ tendríamos el siguiente pseudocódigo:

```
TP [0...n] [0...r] ∈ Entero
Funcion CoefBinomial(↓n ∈ Entero, ↓r ∈ Entero)⟶∈ Entero
 Inicio
  Si r<0 OR  r>n Entonces
    regresar 0
  Si r =0 OR r=n  Entonces
    regresar 1

  Si TP[n][r] =0
     TP[n][r] =CoefBinomial( n-1, r-1 ) + CoefBinomial( n-1, r)
     regresar TP[n][r]

 Fin CoefBinomial
```

VI.7 Ejercicios

1.- Haz un programa que llame a la función `CambioProgDinámica` de la sección VI.4 para que, dado un monto (entre 0 y 8) y una denominación máxima a tomar (1, 4 ó 6), se obtenga el número mínimo de monedas de cambio. (Puedes usar como base el código que presentamos para hacer la matriz de programación dinámica de esta sección).

2.- Haz un programa que llame a la función `MochilaProgDinámica` de la sección VI.5 para que, dado un peso (entre 0 y 11) y un objeto máximo a tomar (del objeto 1 al 5), se obtenga el valor máximo. (Puedes usar como base el código que presentamos para hacer la matriz de programación dinámica de esta sección).

3.- Implementa el código en C/C++ para construir el triángulo de Pascal.

VI.8 Resumen del capítulo

En este capítulo hemos estudiado el paradigma de diseño llamado *programación dinámica*, la cual se utiliza para mejorar el rendimiento de algunos algoritmos. Este paradigma es útil cuando, al dividir un problema en varios subproblemas, éstos no son independientes entre sí, es decir, se traslapan. Si estos subproblemas se resuelven de forma independiente y, usamos el paradigma divide y vencerás, las partes traslapadas se resuelven varias veces, lo que es ineficiente. Con la programación dinámica se utilizan tablas como una estructura auxiliar para evitar el solapamiento, es decir, evitar calcular varias veces un mismo resultado. Los ejemplos que resolvimos en este capítulo para ilustrar la técnica de la programación dinámica fueron: el cálculo de los números de Fibonacci, el problema de las apuestas, el problema del cambio, el problema de la mochila, y el problema de los coeficientes binomiales. En ocasiones, ni los algoritmos voraces, ni la programación dinámica son viables para resolver un problema, entonces se utilizan técnicas que, si bien, no son tan eficientes como las anteriores, tienen mucho poder, ya que buscan en un espacio muy amplio de soluciones posibles, tal es el caso del backtracking (búsqueda con retroceso) y de la ramificación y poda. En el siguiente capítulo se aborda la técnica del backtracking.

Capítulo VII Árboles de búsqueda: backtracking y ramificación y poda.

VII.1 Introducción

Cuando las técnicas divide y vencerás, algoritmos voraces y programación dinámica no son aplicables para resolver un problema, entonces no queda más remedio que utilizar la fuerza bruta. A estos algoritmos también se les conoce como *exhaustivos* y son aquellos que analizan todo el espacio de búsqueda para encontrar una o todas las soluciones, es decir, agotan todas las posibilidades hasta hallar la solución. Estos algoritmos garantizan que pueden encontrar una solución óptima sin embargo, son bastante ineficientes y, cuando el conjunto de soluciones posibles es muy grande, resultan no viables debido a que su tiempo de ejecución es enorme. El *backtracking* (búsqueda con retroceso) se aplica a problemas que requieren de una búsqueda exhaustiva dentro del conjunto de todas las soluciones potenciales. El espacio a explorar se estructura de tal forma que se puedan ir descartando bloques de soluciones que posiblemente no son satisfactorias. Los algoritmos de backtracking optimizan el proceso de búsqueda de tal forma que se puede hallar una solución de una forma más rápida que con los algoritmos de la fuerza bruta. *Ramificación y poda*, es una variante del backtracking y tiene el objetivo de evitar el crecimiento exponencial que se presenta cuando se aplica una búsqueda exhaustiva.

VII.2 Definición del paradigma de backtracking

El método de backtracking es una técnica de resolución de problemas que realiza una búsqueda exhaustiva, sistemática y organizada sobre el espacio de búsqueda del problema y es aplicable a problemas de optimización, juegos y búsqueda entre otros. Se analiza un conjunto conocido de soluciones posibles para buscar en éste la solución óptima. La técnica

de backtracking proporciona una manera sistemática de generar todas las posibles soluciones siempre que dichas soluciones sean susceptibles de resolverse en etapas.

¿Qué tipo de problemas se resuelven con algoritmos backtracking?

El backtracking es útil para problemas de optimización en los que no existe ninguna teoría que pueda aplicarse para encontrar el óptimo. También en algunos juegos. Por ejemplo el ajedrez, el gato y los laberintos. Se usa también en los análisis gramaticales (también llamados *parsers*) y en procesos de calendarización (*scheduling*).

VII.3 Características de los algoritmos de backtracking

- Funciona en problemas cuyas soluciones se pueden construir por etapas.
- La solución maximiza, minimiza o satisface una *función criterio*.
- Sea S el conjunto de todas las soluciones posibles de un problema, y S_i una de éstas soluciones. Una solución parcial se expresa mediante una *n*-tupla (x_1, x_2,…, x_n). El backtracking trabaja tratando de extender una solución parcial continuamente, de tal forma que cada $x_i \in S_i$ representa la decisión que se tomó en la *i*-ésima etapa, dentro de un conjunto finito de alternativas.
- El espacio de posibles soluciones S se estructura como un árbol de exploración, en el que en cada nivel se toma la decisión de la etapa correspondiente, y se le llama *nodo estado* a cualquier nodo del árbol de exploración.

En cada etapa de búsqueda en el algoritmo backtracking, el recorrido del árbol de búsqueda se realiza en profundidad. Si una extensión de la solución parcial actual no es posible, se retrocede para intentar por otro camino, de la misma manera en la que se procede en un laberinto cuando se llega a un callejón sin salida.

Es común que los árboles de exploración se definan de tal forma que los nodos solución solo se encuentran en las hojas. En la *Figura 7.1* se muestra como ejemplo el árbol de exploración para el juego del gato.

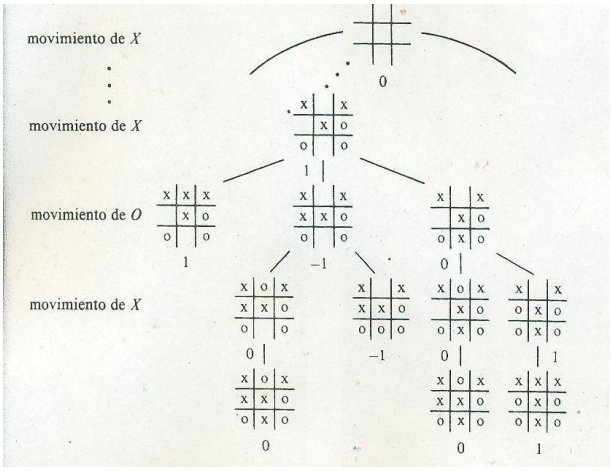


Figura 7.1: Árbol de exploración para el juego del gato [Aho et al., 1998]

El siguiente algoritmo es un esquema general de los algoritmos de retroceso:

```
Función Retroceso( etapa )
  Inicializar_Opciones();
  Hacer
    opcion←SeleccionarOpcion
    Si aceptable(opcion) Entonces
     Guardar (opcion)
     Si incompleta(solucion) Entonces
       exito ←Retroceso( siguiente( etapa ) ) // recursión
       Si ( exito = FALSO ) Entonces
         retirar( opcion )
       Fin Si
     Si no  // solución completa u hoja del árbol
       exito ← VERDADERO
     Fin Si
    Fin Si
  mientras (exito = FALSO ) AND ( opcion != UltimaOpcion)

  regresar exito

Fin Retroceso
```

Para una determinada etapa, se elige una de las opciones dentro del árbol de búsqueda y se analiza para determinar si esta opción es aceptable como parte de la solución parcial. En caso de que no sea aceptable, se procede a elegir otra opción dentro de la misma etapa hasta encontrar una opción aceptable o bien agotar todas las opciones. Si se encuentra una solución aceptable, ésta se guarda y se profundiza en el árbol de búsqueda aplicando el

mismo procedimiento Retroceso a la siguiente etapa. En caso de que Retroceso devuelva como resultado que no hubo éxito, entonces se retira la opción que se eligió y se intenta con otra, de lo contrario, cuando sí hubo éxito entonces Retroceso termina y regresa el control al procedimiento que lo llamó. Al final del procesamiento Retroceso termina cuando se encontró un caso exitoso o cuando se agotaron todas las opciones.

En general, la velocidad de las computadoras no es práctica para una búsqueda exhaustiva de más de 10^8 elementos. Así es que hay que considerar que aplicar retroceso puede requerir un tiempo tan grande que sea prohibitivo.

VII.4 El problema de la mochila con backtracking

Como vimos en los ejemplos de los capítulos anteriores, el problema consiste en llenar una mochila con el máximo valor, con la limitante de que solo soporta un peso máximo W. Se tienen *n* objetos, cada uno de estos objetos *i* tiene un peso w_i y un valor v_i ($i = 1, 2, \ldots, n$). La solución es factible siempre que se cumpla la restricción:

$$\sum_{i=1}^{n} x_i w_i \leq W$$

x_i toma los valores 0 cuando el peso w_i no se incluye, y 1 cuando w_i se incluye.

También mencionamos que cuando los objetos se pueden romper en trozos más pequeños al multiplicarlos por una fracción x_i, de tal manera que el peso del objeto *i* en la mochila sea $x_i w_i$. Se puede usar un algoritmo voraz, pero que el problema se complica cuando los objetos no se pueden romper. Una de las opciones para resolver el problema es construir una tabla de programación dinámica. En esta sección analizamos la solución del problema utilizando el algoritmo de retroceso. Tomaremos el ejemplo de [Brassard y Bratley, 2008]. Sean 4 objetos con los siguientes pesos y valores:

$w_1 = 2 \quad v_1 = 3$

$w_2 = 3 \quad v_2 = 5$

$w_3 = 4 \quad v_3 = 6$

$w_4 = 5 \quad v_3 = 10$

Modificaremos el problema de tal forma que se pueda tomar cualquier cantidad de objetos del mismo tipo. Por ejemplo, si el peso máximo es W = 8, se pueden tomar 4 objetos con $w_1 = 2$, o también dos con $w_1 = 2$ y uno con $w_3 = 4$, o uno con $w_2 = 3$ y otro con $w_3 = 4$, o bien uno con $w_2 = 3$ y otro con $w_4 = 5$. Las diferentes combinaciones generan un

árbol de búsqueda. En el nodo raíz todavía no se elige ningún objeto. En el primer nivel hay 4 ramas, que representan las cuatro opciones del principio. Una vez que se toma la primera decisión, la cantidad de nodos que se pueden visitar está en función de la restricción del peso y de los nodos que ya se visitaron anteriormente. Por ejemplo, si ya se visitó el nodo con la opción de tomar un objeto con $w_2 = 3$ y otro con $w_4 = 5$, ya no será necesario considerar el caso en el que hay que visitar el nodo que toma al objeto con $w_4 = 5$ y luego al objeto con $w_2 = 3$ puesto que en este problema no importa el orden. De esta forma se reduce el árbol de búsqueda considerablemente. En la *Figura 7.2* se muestra el árbol de búsqueda que se genera para este problema con W=8.

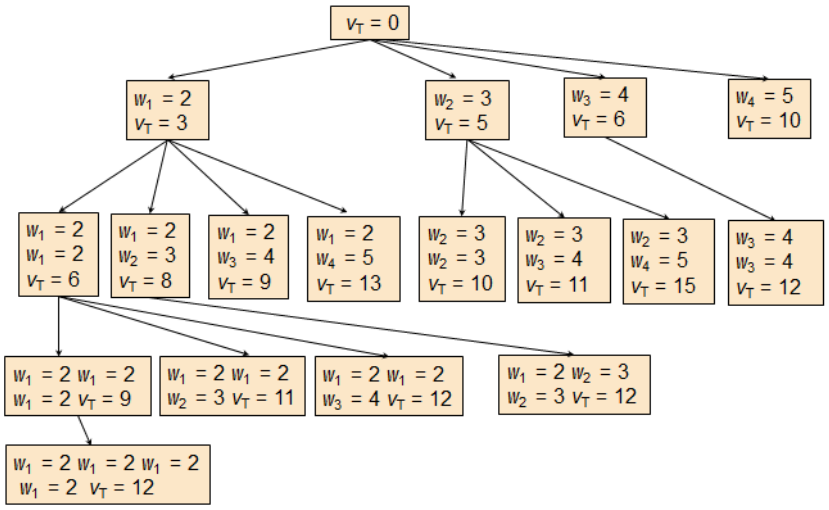


Figura 7.2: árbol de búsqueda para el problema de la mochila

A continuación se presenta un pseudocódigo del algoritmo que maximiza el valor de la mochila recorriendo el árbol de búsqueda anterior, al que llamaremos `MochilaBackTrack`. Este algoritmo calcula el valor de la mejor carga que se puede construir empleando elementos de los tipos *i*, donde cada tipo corresponde a un peso w_i con un valor v_i y los tipos elegibles van de *i* hasta *n*, que es el número total de tipos. Además, el peso no debe sobrepasar el *resto* de la capacidad disponible en la mochila.

```
Función MochilaBackTrack (↓i ∈ Entero, ↓resto ∈ Entero)⟶∈ Entero
  valor ← 0
  para k← i hasta n hacer
    Si w[k] ≤ resto Entonces
      // compara el valor de incluir una pieza de tipo k con el mejor caso de incluir las piezas previas
       valor ←max(valor, v[k] + MochilaBackTrack(k, resto- w[k]) )
    finSi
  finPara
  devolver valor
Fin MochilaBackTrack
```

El algoritmo comienza con un llamado a `MochilaBackTrack(1,8)` y cada llamada recursiva a `MochilaBackTrack(k,W- w[k])` significa extender la profundidad del recorrido del árbol al nivel inmediatamente inferior. El ciclo (**para** *k*← *i* **hasta** *n* **hacer**) examina todas las posibilidades en el nivel dado.

VII.5 Definición del paradigma de ramificación y poda

Ramificación y poda es una variante del *algoritmo de búsqueda con retroceso*. En el caso del algoritmo Retroceso no existen reglas fijas para la búsqueda de soluciones, sino que en la mayoría de los casos, se usa un proceso de prueba y error en cada etapa para ir construyendo una solución parcial. Para muchos problemas, esta prueba en cada etapa crece exponencialmente. Ramificación y poda pretende evitar este crecimiento exponencial.

El algoritmo de ramificación y poda, consiste en realizar el recorrido del árbol de exploración en cierto orden hasta encontrar la primera solución, o bien, recorrer todo el árbol excepto las *zonas podadas* para obtener todas las soluciones o la solución óptima deseada. Para ello es necesario determinar una *función de poda*, también llamada *prueba de factibilidad*, la cual se obtiene a partir de la función criterio. Esta función permite identificar cuándo una tupla parcial nunca conducirá a una solución satisfactoria, por lo que no tiene caso seguir buscando por ese camino. Por ejemplo, cuando se desea *maximizar* los pasos son los siguientes:

- Establecer una cota inferior a la posible solución (una cota superior cuando se trata de *minimizar*).
- Si al seguir una rama del árbol se encuentra una solución parcial inferior a la cota, se descarta toda esa rama.

El algoritmo ramificación y poda puede seguir diferentes estrategias, por ejemplo, recorrer el árbol que representa el espacio de soluciones en profundidad o recorrerlo a lo ancho, o utilizar funciones de costo para seleccionar el nodo que en principio parece más

prometedor. A continuación se presenta un ejemplo siguiendo la estrategia del costo mínimo.

VII.6 El problema de la planificación de tareas con ramificación y poda

Como vimos en la sección V.1 en este problema se dispone de *n* trabajadores y *n* tareas. Se tiene definido un *costo* $c_{ij} > 0$ que indica lo que cuesta asignarle el trabajo *j* al trabajador *i* (puede ser dinero o tiempo). El costo de que el trabajador *i* realice la tarea *j* se indica mediante una *matriz de costos*. En la *Figura 7.3* se ilustra una matriz de costos con cuatro trabajadores: a, b, c y d a los cuales se les debe asignar una de cuatro tareas.

Trabajador/ Tarea	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

Figura 7.3: Matriz de Costos para 4 trabajadores y 4 tareas [Brassard y Bratley, 2008]

Podemos obtener una cota superior al problema con el costo de asignar cualquier tarea a cualquier trabajador. Por ejemplo, el trabajador *a* la tarea 1, a *b* la tarea 2, a *c* la 3 y a *d* la 4, cuyo costo sería 11+15+19+28 = 73. Si tomáramos otra diagonal, haciendo *a* = 4, *b* = 3, *c* = 2, *d* = 1, el costo sería 40+13+17+17= 87, así que 73 es la mejor cota superior de estas dos, ya que es la menor.

Para establecer una cota inferior, tomamos el menor costo de cada tarea, independientemente de quien la ejecute, así tenemos que 11+12+13+22 = 58 es la cota inferior. Otra opción es sumar el costo más pequeño de cada renglón, asumiendo que cada trabajador tiene que hacer una tarea. En este caso 11+13+11+14 = 49 es otra cota inferior, sin embargo, 58 es una cota inferior más útil, porque es superior a 49 y así se acorta el intervalo de búsqueda. Entonces tenemos que la respuesta se encuentra entre 58 y 73.

Elaboraremos árboles con soluciones parciales, en las que a cada nodo del árbol se le asigna una cota en función de las soluciones que se pueden obtener por ese camino "haciendo trampa", es decir, eligiendo el menor costo sin importar que se repita a un trabajador. La forma de obtener la cota es la siguiente.

En el nodo raíz se tienen 4 opciones, que son:

1. Asignar al trabajador *a* la primera tarea (a←1). En este caso, la primera tarea tiene un costo de 11. La cota inferior del resto de las asignaciones se obtiene de la suma de los costos más bajos posibles de las tareas restantes sin tomar ya en cuenta el renglón del trabajador 1, quien ya tiene asignada una tarea. Entonces, el costo mínimo para la tarea 2 es el del trabajador *d* = 14. El menor costo para la tarea 3 es el de *b* = 13, y finalmente, el menor costo de la tarea 4 es el de *b* = 22. Por lo tanto, el costo mínimo aproximado, si se asigna la primera tarea al trabajador *a*, es 11+14+13+22 = 60.
2. Asignar al trabajador *a* la segunda tarea (a←2). En este caso, la segunda tarea tiene un costo de 12. Como la tarea 1 la pueden realizar *b*, *c*, o *d*, el costo mínimo es el del trabajador *c*, que es 11 (ya no se le puede asignar al trabajador *a*). La tarea 3 también podría asignarse a *b*, *c*, o *d*, y entre ellos el que tiene menor costo es *b*, que es de 13. Y finalmente, para la tarea 4 el menor costo es 22. Por lo tanto, el costo mínimo aproximado, si se asigna la segunda tarea al trabajador *a*, es 11+12+13+22 = 58.
3. Asignar al trabajador *a* la tercera tarea (a←3). En este caso, la tercera tarea tiene un costo de 18. Como la tarea 1 la pueden realizar *b*, *c*, o *d*, el costo mínimo es el del trabajador *c*, que es de 11. La tarea 2 también podría asignarse a *b*, *c*, o *d*, y entre ellos el que tiene menor costo es *d*, con 14. Y finalmente, el menor costo de la tarea 4, eligiendo entre uno de los trabajadores *b*, *c*, o *d*, es 22. Por lo tanto, el costo mínimo aproximado, si se asigna la tercera tarea al trabajador *a*, es 11+14+18+22 = 65.
4. Asignar al trabajador *a* la cuarta tarea (a←4). En este caso, la cuarta tarea tiene un costo de 40. Como la tarea 1 la pueden realizar *b*, *c*, o *d*, el costo mínimo es el del trabajador *c*, con un costo de 11. La tarea 2 también podría asignarse a *b*, *c*, o *d*, y entre ellos el que tiene menor costo es *d*, con costo 14. Y finalmente, el menor costo de la tarea 3, eligiendo entre uno de los trabajadores *b*, *c*, o *d*, es 13. Por lo tanto, el costo mínimo aproximado, si se asigna la cuarta tarea al trabajador *a*, es 11+14+13+40 = 78.

Lo anterior se ilustra en la *Figura 7.4*.

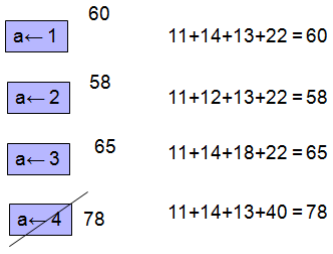


Figura 7.4: Cotas para distintas opciones de asignar tarea al trabajador *a*

Como ya tenemos una cota superior = 73, el cuarto nodo del árbol (a←4) queda "podado" y por lo tanto se descarta. Ahora resta hacer lo mismo para el siguiente nivel del

árbol pero solo tomando en cuenta los 3 primeros nodos. En la *Figura 7.5* se ilustra el cálculo de cotas para el segundo nivel del árbol, en el nodo (a←2), que es el nodo más prometedor porque tiene la cota más pequeña. Los cálculos se hicieron de forma muy similar, por ejemplo, si a *a* se le asignó la tarea 2 y a *b* la 1, entonces ya se tienen los costos 12 y 14, de tal forma que el costo mínimo será asignar a *c* la tarea 3, con costo 19, y a *c* la tarea 4 con costo 23, de ahí obtenemos 12+14+19+23 = 68.

De igual manera, si a *a* se le asignó la tarea 2 y a *b* la 3, entonces ya se tienen los costos 12 y 13, de tal forma que la tarea 1 solo se le puede asignar a *c* o a *d* y el costo mínimo será asignar a *c* la tarea 1, con costo 11, y a *c* la tarea 4 con costo 23, de ahí obtenemos 12+13+11+23 = 59.

Si a *a* se le asignó la tarea 2 y a *b* la 4, entonces ya se tienen los costos 12 y 22, de tal forma que la tarea 1 solo se le puede asignar a *c* o a *d* y el costo mínimo será asignar a *c* la tarea 1, con costo 11, y a *c* la tarea 3 con costo 19, de ahí obtenemos 12+22+11+19 = 64. Como se observa en la *Figura 8.3.* La rama con cota superior de 68 se poda porque hay una rama en el nivel anterior con cota de 65.

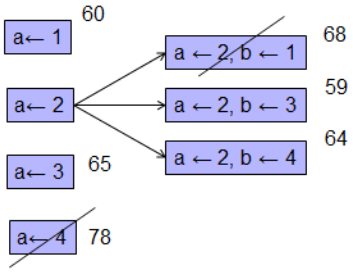


Figura 7.5: Cotas para distintas opciones de *b* cuando se asignó la tarea 2 al trabajador *a*

En la *Figura 7.6* tenemos toda la rama completa para el caso del nodo (a←2) y encontramos que en esta rama, el costo mínimo es 64 y esta se convierte en nuestra nueva cota superior. Gracias a esta cota podemos podar el nodo (a←3). El (a←2, b←4) tampoco tiene caso explorarlo, ya que ni haciendo trampa mejoran la cota 64, que es una solución real.

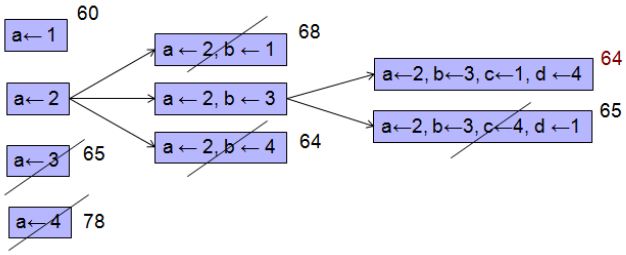


Figura 7.6: Encontramos una nueva cota superior = 64

En la *Figura 7.7* se observa toda la rama completa para el caso del nodo (a←2) y también para el nodo (a←1), que es el segundo más prometedor. En ambos casos se eligió el nodo del siguiente nivel con la cota más prometedora.

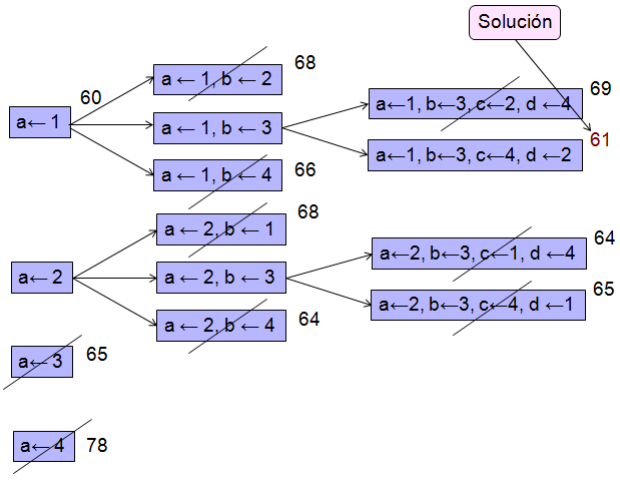


Figura 7.7: Ramas de los nodos más prometedores

Como la solución *a*←1, *b*←3, *c*←4, *d*←2 tiene un costo de 61 y este es un valor inferior a la cota de los otros nodos que faltan por explorar, entonces ya no tiene caso seguir explorándolos. Nótese que aunque al principio el nodo *a*←2 era el más prometedor, resultó que el costo mínimo se encontró explorando el nodo *a*←1.

VII.7 Ejercicios

1.- Haz un programa que se base en el algoritmo de la sección VII.4 para resolver el problema de la mochila con backtracking.

2.- Haz un programa que resuelva el problema de la asignación de tareas descrito en la sección VII.6 utilizando ramificación y poda.

3.- Resuelve por ramificación y poda el siguiente problema de asignación de tareas

Trabajador/ Tarea	1	2	3	4
a	29	19	17	12
b	32	30	26	28
c	3	21	7	9
d	18	13	10	15

Solución.- Para establecer una cota superior tenemos la suma de las dos diagonales:

29+30+7+15 = 81
18+21+26+12 = 77 ⟵ mejor solución real (cota superior)

Para establecer una cota inferior tenemos:
El menor costo de cada tarea: 3+13+7+9 =32
El costo más pequeño por persona: 12+26+3+10 = 52 ⟵ mejor cota inferior

Entonces tenemos que la solución está acotada por el intervalo (51, 77)

Estableciendo una cota para el nodo a←1 : 29+13+7+9 = 58

Estableciendo una cota para el nodo a←2 : 19+3+7+9 = 38

Estableciendo una cota para el nodo a←3 : 17+3+13+9 = 42

Estableciendo una cota para el nodo a←4 : 12+3+13+7 = 35

Como todas las cotas están por debajo de la cota superior, entonces hay que explorar todos los nodos.

Explorando el nodo a←1 b←2 : 29+30+7+9 =75

Explorando el nodo a←1 b←3 : 29+26+13+9 = 72

Explorando el nodo a←1 b←4 : 29+28+3+9 = 69

Todas las cotas son más altas que la del nodo más prometedor a←4 (35). Lo mismo sucede cuando se exploran los nodos a←2 y a←3, detallaremos aquí la exploración del nodo a←4:

Explorando a←4 b←1: 12+32+13+7=64

Explorando a←4 b←2: 12+30+3+7 =52

Explorando a←4 b←3: 12+26+3+13=54

El nodo a←4 b←1 lo podemos podar pues su cota es muy alta, explorando a←4 b←3 obtenemos las cotas 54 y 77 así que exploraremos el nodo a←4 b←2:

a←4 b←2 c←1 : 12+30+3+7 = 52

a←4 b←2 c←3 : 12+30+7+18 = 67

La mejor solución encontrada es a←4 b←2 c←1 : 12+30+3+7 = 52

VII.8 Resumen del capítulo

Backtracking es una técnica de diseño que emplea algoritmos exhaustivos, es decir, algoritmos que analizan todo el espacio de búsqueda para encontrar todas las soluciones posibles. Aunque estos algoritmos garantizan que pueden encontrar la solución óptima, son muy ineficientes, ya que, cuando el conjunto de soluciones posibles es muy grande, su tiempo de ejecución crece tanto que es imposible llevarlo a la práctica. En el backtracking se generan sistemáticamente todas las soluciones posibles siempre y cuando estas soluciones se puedan resolver por etapas. Es útil en problemas de optimización en los que no existe ninguna teoría que pueda aplicarse para encontrar el óptimo. Para ilustrar el backtracking, analizamos el problema de la mochila.

El paradigma de ramificación y poda, es una variante del backtracking. Tiene el objetivo de evitar el crecimiento exponencial que se presenta cuando se aplica una búsqueda exhaustiva. Consiste en realizar el recorrido del árbol de exploración en cierto orden hasta encontrar la primera solución, o bien, recorrer todo el árbol excepto las *zonas podadas* para obtener todas las soluciones o la solución óptima deseada. Para ello es necesario determinar una *función de poda*, que identifique cuando una tupla parcial nunca conducirá a una solución satisfactoria. Cuando se va a *maximizar*, es necesario establecer una cota inferior (cota superior para *minimizar*) a la solución posible. Si al seguir una rama del árbol se encuentra una solución parcial inferior a la cota, se descarta toda esa rama. Para ilustrar el paradigma de ramificación y poda, analizamos el problema de la planificación de tareas.

Para terminar este libro, hemos dedicado el último capítulo a proporcionar una introducción a los problemas NP, este tema forma parte de un curso más avanzado del análisis de algoritmos.

Capítulo VIII Introducción a problemas NP

VIII.1 Introducción

Las ciencias de la computación son relativamente recientes, todavía queda mucho por hacer y descubrir en cuanto a técnicas algorítmicas. A los algoritmos que resuelven un problema en tiempo polinomial $\mathbf{O}(n)$, $\mathbf{O}(n^2)$,… se les dice *polinomiales*. Además, son *deterministas* aquellos algoritmos que cada vez que se ejecutan con una misma entrada obtienen la misma solución. Se dice que los algortimos *polinomiales* son eficientes. Brassard & Brattley (2008) definen la eficiencia de un algoritmo de la siguiente manera: "un algoritmo es eficiente si existe un polinomio $p(n)$ tal que el algoritmo puede resolver cualquier caso de tamaño n en un tiempo que está en $\mathbf{O}(p(n))$". Sin embargo, hay problemas cuyos algoritmos de solución tienen tiempos de ejecución factoriales $\mathbf{O}(n!)$ y exponenciales $\mathbf{O}(x^n)$ (x >1). Estos algoritmos son ineficientes porque, cuando crece el tamaño de sus entradas solo un poco más, requieren de tiempos absurdamente grandes para encontrar la solución. Sin embargo, es importante resolver algunos de estos problemas ya que se presentan en la vida real y es útil encontrar una solución aunque solo sea "razonablemente" buena y no la mejor posible.

Un ejemplo típico es encontrar la ruta más corta para visitar varias ciudades (el problema del agente viajero), o el problema de la asignación cuando intervienen 3 o más dimensiones. Por ejemplo, la asignación de salones, horarios, profesores y asignaturas tiene 4 dimensiones. Otros problemas típicos son, el de la mochila, la búsqueda del camino simple más largo de un grafo y la verificación de la existencia de ciclos hamiltonianos en un grafo. Hasta la fecha no se ha encontrado un algoritmo que resuelva esta clase de problemas en tiempo polinomial. Los mejores algoritmos para resolver estos problemas crecen exponencialmente con el tamaño de la entrada y por esto se les cataloga como problemas difíciles. Los algoritmos que toman un tiempo exponencial son imprácticos, pues requerirán más tiempo del disponible, excepto para tamaños de entrada muy pequeños.

Identificar la clase de complejidad de problemas como los anteriormente mencionados es útil, porque nos permite encontrar una buena estrategia para enfrentarlo. En la siguiente sección, estudiaremos algunos conceptos que sirven para entender porqué un problema está dentro de alguna clase de complejidad.

VIII.2 Definición de algunos conceptos importantes

Para poder clasificar la complejidad de los problemas de acuerdo a su complejidad, es necesario definir los siguientes conceptos:

Máquina de Turing.- Es un modelo teórico de una computadora que ideó el matemático Alan Turing.

Consta de los siguientes elementos:

- Una *cinta* de longitud infinita, dividida en celdas, donde cada celda puede contener solo un símbolo.
- Un *diccionario de símbolos* predefinido del cual se toman los símbolos para las celdas.
- Un *control* que tiene la capacidad de examinar el símbolo de una celda y tomar una decisión. La decisión depende tanto del símbolo como del estado en el que se encuentra el control en ese momento. Entonces, el control esta siempre en algún estado de *n* estados posibles.

El control comienza con un estado inicial y apuntando a una celda de la cinta. A partir de esta situación, el control debe ejercer tres acciones diferentes:

1. Cambiar de estado, o permanecer en el actual.
2. Escribir un nuevo símbolo en la celda a la que apunta, o no.
3. Desplazarse hacia la celda de la derecha o hacia la celda de la izquierda, (solo una celda a la vez) o quedarse en la celda actual.

El objetivo de la máquina de Turing es ejecutar el programa que esté codificado en la cinta. El resultado obtenido después del procesamiento queda plasmado en los símbolos de la cinta. Es necesario definir un "estado inicial" y una celda de inicio para el control. Además definir también un "estado final" (o varios) para indicar al control cuando el proceso ha terminado.

Se dice que "un problema tiene solución algorítmica (es computable) cuando puede ser representado por medio de una máquina de Turing que llega, en algún momento, a un estado final" [Levine, 1996].

Un problema es *indecidible* cuando no es computable, es decir, cuando no tiene solución algorítmica. En teoría, ni un genio para diseñar algoritmos puede encontrar alguna solución a este tipo de problemas.

Máquina de Turing probabilística.- Es una máquina de Turing con una instrucción adicional: "escribir" donde el valor de la escritura se determina aleatoriamente con una distribución de probabilidad en el alfabeto de la máquina. Como consecuencia, una máquina de Turing probabilística puede (a diferencia de una máquina de Turing) tener un resultado estocástico: Dada una entrada y un programa para la máquina, puede ejecutar el programa en tiempos variables de ejecución o puede no parar; más aún, puede aceptar una entrada en una ejecución y no aceptarla en la siguiente.

Algoritmo determinista.- Es aquel que si se le introduce varias veces una misma entrada, pasa por los mismos estados y produce exactamente la misma salida cada vez.

Algoritmo no determinista.- Dentro de la ciencia de la computación, existe un caso especial de algoritmos que emplean el modelo de la "máquina de Turing probabilística", a éstos se les llama no deterministas porque para una misma entrada, la salida no siempre es la misma. Otra forma de definirlo es: un algoritmo que supone una solución y luego comprueba si es válida. Por ejemplo, un algoritmo no determinista para el problema del agente viajero supone un recorrido correcto, y luego comprueba si su costo es menor que una constante dada. Los algoritmos no deterministas no existen ni existirán porque no es posible generar una distribución de probabilidad completamente plana, es decir, todas las computadoras tienen un sesgo que podría impedir generar la solución que se requiere. Sin embargo, aunque no exista, el algoritmo no determinista nos servirá más adelante para definir los problemas NP.

En una *máquina determinista* se ejecuta una instrucción cada vez y, a partir de una instrucción específica se ejecuta la siguiente, la cual es única, mientras que en una *máquina no determinista* existen varias opciones para la siguiente instrucción, la máquina puede elegir cualquiera de éstas y si una de las opciones conduce a la solución, entonces elegirá la opción marcada como la correcta. Aparentemente éste es un modelo ridículo, ya que es imposible construir una computadora no determinista, sin embargo la máquina no determinista tiene una importante utilidad teórica. Es más, este modelo no es tan poderoso como uno podría pensar, ya que, por ejemplo, los problemas indecidibles siguen siendo indecidibles incluso para las máquinas no deterministas.

Problemas de optimización.- Buscan una solución con la cual se obtenga el valor mínimo (problema de minimización) o bien el valor máximo (problema de maximización) de una función objetivo. Los problemas de optimización se pueden transformar a problemas de decisión.

Problemas de decisión.- Son aquellos cuya solución es un "si" o un "no".

VIII.3 Clases de complejidad

Podemos clasificar los problemas en diversos tipos, por ejemplo, problemas de ordenamiento, de búsqueda, de optimización, de decisión, etc. Los problemas de decisión dan como resultado final un *Si* o un *No*. Este tipo de problemas son interesantes porque permiten comprobar de manera eficiente la validez de una posible solución a un problema complejo. Por ejemplo, en el caso de los ciclos hamiltonianos, es difícil hallar todos los caminos que formen un ciclo hamiltoniano, sin embargo, transformando el problema a un problema de decisión, es sencillo determinar si una cierta sucesión de nodos define o no a un ciclo hamiltoniano. En general, para muchos problemas resulta mucho más sencillo verificar la validez de una supuesta solución que encontrar una. Casi todos los problemas se pueden transformar en un problema de decisión. De este hecho se desprende que en la teoría de la complejidad sea de especial interés el estudio de los problemas de decisión.

La complejidad de los problemas se clasifica en diversas clases. En estas notas nos limitaremos a definir los P, NP, NP-completos y NP-Difíciles.

VIII.3.1 Problemas NP

Problemas P (Polinomial): Los problemas P se pueden resolver mediante un algoritmo con complejidad polinomial y se dice que son *tratables*.

Problemas NP (NP: Non deterministic Polynomial-time Problem).- Los problemas NP son aquellos problemas que se pueden transformar (reducir) a un problema de decisión que se resuelve mediante un algoritmo no determinista en un tiempo polinomial. Sin embargo, los algoritmos no deterministas son una abstracción matemática que no se puede utilizar directamente en la práctica, es decir, no se pueden programar y ejecutar en una computadora real ya que las computadoras son deterministas. No obstante, todos los problemas NP tienen la propiedad de que se puede verificar de manera eficiente (complejidad polinomial) si una posible solución es correcta o no. Por ejemplo, el problema de los ciclos hamiltonianos es NP, porque se puede verificar si una ruta en particular es hamiltoniana o no, en tiempo polinomial.

En otras palabras, los problemas NP son problemas de decisión. Un problema es de clase NP cuando el algoritmo para decidir si una posible solución es satisfactoria o no, se ejecuta en tiempo polinomial. Sin embargo, dicha solución está dada por una máquina no determinista, es decir, se parte de la base de que de alguna forma se tiene acceso a una posible solución.

También existen problemas que no están en la clase NP, por ejemplo el problema de determinar si en un grafo “NO existe” un ciclo hamiltoniano. Este problema es mucho más difícil, ya que, basta mostrar un solo ciclo para demostrar que un grafo tiene ciclos hamiltonianos, sin embargo, nadie conoce un algoritmo que pueda mostrar, en tiempo

polinomial, que un grafo no tiene ciclos hamiltonianos (habría que hacer una búsqueda exhaustiva de todos los caminos posibles en el grafo).

VIII.3.2 Reducción de un problema a otro

Para entender la definición de los problemas NP-completos es necesario entender primero cómo se puede "reducir" un problema P_1 a otro problema P_2.

Reducción de P_1 a otro problema P_2.- Para hacer esta reducción se requiere de una función que pueda transformar cualquier instancia de P_1 a una instancia de P_2. Después se resuelve P_2 y la solución se transforma a una solución de P_1. Por ejemplo, cuando se introducen datos a una calculadora, normalmente se hace en sistema decimal, la calculadora transforma estos números a sistema binario, hace los cálculos en binario y después transforma la solución a decimal para mostrarla al usuario. Se dice que P_1 se *reduce* a P_2, porque basta con encontrar la solución a P_2 para tener también la de P_1.

*Reducción **polinomial** de P_1 a otro problema P_2.-* Que un problema P_1 se pueda *reducir polinomialmente* a otro problema P_2 significa que todas las tareas asociadas con la reducción (esto excluye las tareas asociadas con la solución del problema) se pueden ejecutar en un tiempo polinomial.

Decimos que un problema P_2 es tanto o más complejo que un problema P_1 cuando las soluciones que obtenemos para P_2 son también soluciones para P_1 mediante una reducción polinomial de P_1 a P_2. Si, por el contrario, las soluciones de P_2 no siempre pueden transformarse en soluciones de P_1 entonces P_2 es menos complejo que P_1. En este caso no existe un algoritmo polinomial para reducir P_1 a P_2.

Por ejemplo, el problema de los ciclos hamiltonianos (P_1) se puede reducir al problema del agente viajero (P_2) como se muestra en la *Figura 8.1*.

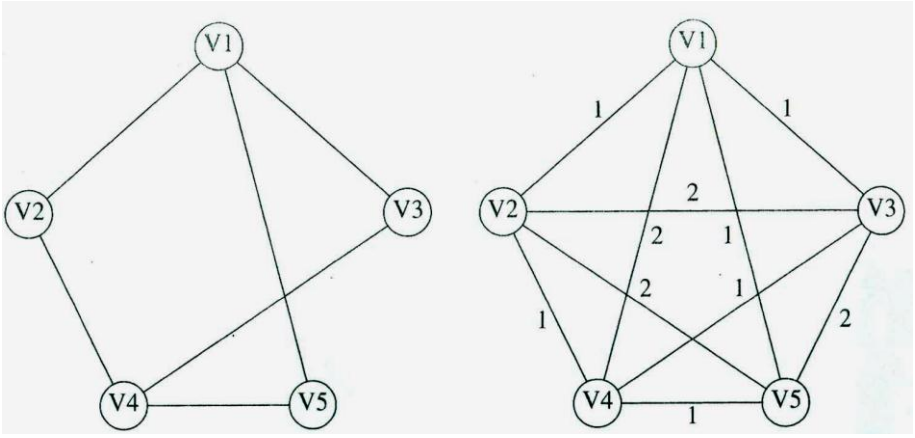


Figura 8.1: Reducción del problema de ciclos hamiltonianos al problema del agente viajero [Weiss, 2007].

Los vértices que tienen conexión entre sí en el problema P_1 se transforman a P_2 con aristas de peso 1 y la ausencia de conexión se transforma con aristas de peso 2. La pregunta que se hace en P_2 es la siguiente: ¿existe un ciclo que visite todos los vértices (ciclo hamiltoniano) y que tenga un costo total ≤ 5? Para este caso particular podemos apreciar que, si queremos un camino que pase por todos los vértices en el grafo de la derecha, el costo será por lo menos de 6. Entonces la respuesta al problema P_2 es, no existe un ciclo que visite todos los vértices y que tenga un costo total ≤ 5. Transformando esta respuesta a P_1 tenemos que no hay un ciclo hamiltoniano en P_1. Esto se puede confirmar en el grafo de la derecha.

VIII.3.3 Problemas NP-completos

De entre todos los problemas que se sabe están en NP, hay un subconjunto conocido como el de los problemas NP-completos, el cual contiene a los problemas NP más difíciles. La NP-Completitud es una propiedad de los problemas de decisión.

Definimos que "*un problema NP-completo tiene la propiedad de que* ***cualquier*** *problema NP puede ser reducido a él en un tiempo polinomial*" [Weiss, 2007] (incluyendo a los mismos problemas NP-completos ya que también son NP). Entonces, los problemas NP-completos pueden verse como una "subrutina" para resolver los problemas NP en la que se agrega un tiempo polinomial para hacer el llamado a la subrutina y para interpretar la solución. Esto implica que si *algún* problema NP-completo tuviera solución en tiempo polinomial entonces *todo* problema en NP tiene también solución en tiempo polinomial. Se piensa que esto es improbable. Podemos concluir que los problemas NP-completos son al menos tan difíciles como los NP pero se conjetura que son más difíciles.

La pregunta que surge ahora es: ¿Cómo se encuentra un problema NP-completo? El primer problema que se identificó, en la década de los 70´s, como NP-completo fue el problema SAT (Satisfactibilidad lógica), el cual consiste en determinar si existe al menos un conjunto de valores en las variables de una expresión lógica que produzcan como resultado Verdadero. Entonces cualquier problema en NP puede reducirse a un SAT. Algunos problemas NP-completos conocidos son: el SAT, el problema del agente viajero, el problema de la mochila y el problema del coloreado de grafos.

VIII.3.4 Problemas NP-Difíciles

La NP-Completitud es una propiedad de los problemas de *decisión*, mientras que la NP-dificultad es una propiedad que no solo se asocia a problemas de decisión sino también a problemas de *optimización*.

Un problema NP-difícil es al menos tan difícil como los problemas NP-completos. Se puede probar que un problema es NP-difícil, mostrando que al menos un problema que se sabe es NP-completo puede reducirse a él. En palabras más sencillas, que los NP se reduzcan a un NP-completo significa que los NP-completos sirven para resolver cualquier

problema en NP y que, los problemas NP-completos se reduzcan a un NP-difícil significa que los NP-difíciles sirven para resolver cualquier problema NP-completo.

Por otra parte, como la definición de NP-difícil dice que un NP-completo se reduce a un NP-difícil, los problemas NP-completos son también NP-Difíciles, ya que cualquier problema NP-completo se puede reducir a otro NP-completo.

Cuando se prueba que, la *versión de decisión* de un problema de optimización pertenece a los NP-completos, entonces la versión de optimización pertenece a los NP-difíciles, porque siempre se puede resolver el problema de decisión cuando se ha resuelto la versión de optimización, es decir, el problema de decisión se reduce al problema de optimización.

Por ejemplo, la versión de decisión del problema del agente viajero, en la que se decide si existe un recorrido que tiene un costo menor que *k*, es NP-completo. Para dar una respuesta afirmativa basta con obtener un ejemplo de un recorrido cuyo costo sea menor a *k*. La versión de optimización de este mismo problema, que consiste en determinar cuál es el recorrido menos costoso, es un problema NP-difícil ya que, para hallar la solución, es necesario demostrar que ningún recorrido es menos costoso que cierta cantidad.

Pongamos otro ejemplo. El problema del coloreado de gráficas en su versión de optimización consiste en determinar cuál es el mínimo número de colores *k* con que se pueden pintar los vértices de un grafo *G* de tal forma que ningún par de vértices consecutivos compartan el mismo color. En su versión de decisión el problema es: dado un grafo *G* y un entero *k*, ¿puede pintarse *G* con *k* colores? Obviamente el primero es más complejo ya que se debe conseguir una evidencia de que ninguna coloración de *G* con *k*-1 colores es válida. Para el problema de decisión basta con obtener un ejemplo de coloración válida de *G* con *k* colores.

VIII.4 Resumen del capítulo

Un problema de decisión es de clase NP, cuando el algoritmo para decidir si una solución, proporcionada por un algoritmo no determinista, es satisfactoria o no, se ejecuta en tiempo polinomial. Dentro de los problemas NP se encuentran los NP-completos, los cuales son los más difíciles de los problemas NP. Si transformamos polinomialmente un problema NP a un NP-completo y resolvemos este último, entonces también obtendremos la solución al problema NP. Cualquier problema NP-completo se puede transformar polinomialmente a otro NP-completo, por ejemplo, el de la mochila, o el de los ciclos hamiltonianos al del agente viajero. La NP-completitud está asociada a los problemas de decisión.

Los problemas de optimización son más difíciles que los de decisión, cuando se estudia la propiedad de la NP-dificultad se consideran también los problemas de optimización.

Cualquier problema NP-completo se puede transformar polinomialmente a un problema NP-difícil. Si pudiéramos resolver el NP-difícil obtendríamos también la solución al NP-completo.

Algunos de los problemas que hemos visto a lo largo de este libro, están identificados como problemas que tienen un algoritmo de solución cuyo tiempo de ejecución crece exponencialmente con el tamaño del problema. Estos algoritmos se vuelven prohibitivamente costosos conforme va creciendo el tamaño de los datos de entrada, tal es el caso del problema del agente viajero y el de la mochila. Estos problemas se presentan en la práctica y es muy útil poder disponer de una buena solución. Estos problemas son de tipo NP-completo. Es útil saber identificar cuándo un problema es NP-completo, porque así sabremos que no existe, hasta ahora, un algoritmo que lo resuelva en su versión de optimización en tiempo polinomial. Por lo que será más conveniente conformarse con una buena solución, que se obtenga en un tiempo razonable, aunque ésta no sea la mejor posible (la óptima).

Bibliografía

- Abellanas M., Lodares D., *Algoritmos y teoría de grafos*, 1era ed, Macrobit RA-MA, México, 1990.
- Aho A., Hopcrof J. y Ullman. *Estructuras de Datos y Algrítmos* Pearson Educación, México, 1998.
- Brassard G. y Bratley P. *Fundamentos de Algorítmia*, Prentice Hall, España, reimpresión 2008.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to algorithms. MIT press. 2009.
- Guerequeta R. y Vallecillo A. *Técnicas de Diseño de Algoritmos.* Servicio de publicaciones de la universidad de Málaga, 2ª Edición, España, 2000. Disponible en http://www.lcc.uma.es/~av/Libro/indice.html. (Ultima consulta, enero 2013).
- Lee R.C.T., Tseng S.S., Chang R.C, Tsai Y.T., *Introducción al diseño y análisis de algoritmos. Un enfoque estratégico*, McGraw Hill, México, 2007.
- Levine G., *Estructuras fundamentales de la computación*. McGraw-Hill, México. 1996.
- López G., Jeder I. y Vega A. *Análisis y Diseño de Algoritmos. Implementación en C y Pascal*. Alfaomega, Argentina, 2009.
- Martí N., Segura C. M. y Verdejo J. A. *Especificación, Derivación y Análisis de Algoritmos*. Pearson-Prentice Hall, Madrid, España, 2006.
- Martí N., Ortega Y. y Verdejo J.A. *Estructuras de datos y métodos algorítmicos.* Pearson- Prentice Hall. Madrid, España, 2004.

- McConnell J., *Analysis of Algorithms. An Active Learning Approach* 2nd ed., Jones and Bartlett Publishers, USA, 2008.

- Parberry I. *Problems on Algorithms.* Prentice Hall, Englewood Cliffs, USA, 1995.

- Peña R. *Diseño de programas: Formalismo y abstracción*. Pearson Educación, 3era ed, Madrid España, 2005.

- Villalpando F., *Análisis asintótico con aplicación de funciones de Landau como método de comprobación de eficiencia en algoritmos computacionales*. E-Gnosis, Revista Digital Científica y tecnológica, vol. 1, art. 15, México, 2003.

- Weiss W. *Estructuras de Datos y Algoritmos en Java*. Addison Wesley iberoamericana, reimpresión de 2009.

- Weiss, M. A. *Data Structures and Algorithm Analysis in Java*, 2nd ed. Pearson-Addison Wesley, 2007.

Introducción al Análisis y al Diseño de Algoritmos
Se terminó de imprimir el 03 de junio de 2014 en
Publidisa Mexicana S. A. de C.V.
Calz. Chabacano No. 69, Planta Alta
Col. Asturias C.P. 06850.
26 ejemplares en papel bond 90 gr.