

# DISEÑO DE ALGORITMOS

APUNTES PARA CLASES

(ESTO NO PRETENDE SER UN LIBRO)

---

JOSÉ CANUMÁN CHACÓN

*Profesor Depto Ingeniería en Computación  
Facultad de Ingeniería  
Universidad de Magallanes  
jcanuman @kataix*

*jose.canuman@umag.cl*

---

# Índice

<b>I Referencias</b>	<b>2</b>
<b>II Introducción</b>	<b>4</b>
<b>1. Pero, ¿Qué es un algoritmo?</b>	<b>4</b>
1.1. Los algoritmos en la historia	5
1.1.1. Antigüedad	5
1.1.2. Edad Media	6
1.1.3. Edad Moderna	6
1.2. Los límites de los algoritmos: P y NP	8
<b>III Análisis</b>	<b>10</b>
<b>2. ¿Cómo medir un algoritmo?</b>	<b>10</b>
<b>3. Orden asintótico</b>	<b>10</b>
3.1. Simplificación	12
<b>4. Análisis: Estimación de la ejecución</b>	<b>13</b>
4.1. Peor caso, mejor caso y promedio	14
4.2. Mínimo	15
4.2.1. Búsqueda secuencial	16
<b>5. Ordenación por comparación</b>	<b>18</b>
5.1. Intercambio	19
5.2. Selección	21
5.3. Inserción	24

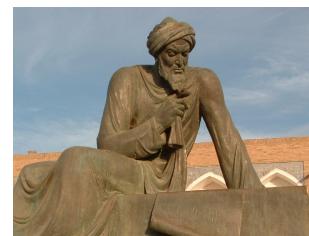
<b>6. Solución de ecuaciones de recurrencias</b>	<b>25</b>
6.1. Recurrencias homogéneas	25
6.2. Ecuaciones no homogéneas	27
6.3. Cambio de variables	27
6.4. Iteraciones	28
<b>7. Cuestiones y problemas</b>	<b>33</b>
<b>IV Diseño</b>	<b>36</b>
<b>8. Divide y Vencerás</b>	<b>37</b>
8.1. Características	37
8.2. Estructura general	37
8.3. Ecuaciones de Recurrencia	37
8.4. Casos Especiales	38
8.4.1. División en Dos Partes Iguales	38
8.4.2. División en Tres Partes Iguales	38
8.5. Teorema Maestro	38
8.6. Ejemplos Clásicos	39
8.6.1. Merge Sort	39
8.6.2. Multiplicación de Matrices de Strassen	39
8.6.3. Quick Sort (caso promedio)	39
8.6.4. Búsqueda Binaria	39
8.7. Ventajas y Desventajas	40
8.8. Búsqueda Binaria	40
8.9. Exponenciación	41
8.10. Mínimo y Máximo	42
8.11. Más problemas clásicos	43
<b>9. Programación Dinámica</b>	<b>43</b>
9.1. Características Principales	44
9.2. Estructura General	44

9.3.	Tipos de Programación Dinámica	44
9.3.1.	Top-Down (Memoización)	44
9.3.2.	Bottom-Up (Tabulación)	44
9.4.	Ventajas y Desventajas	45
9.5.	Cuándo Usar Programación Dinámica	45
9.6.	Fibonacci	45
9.6.1.	¿Se puede hacer mejor?	48
9.7.	Problema de la mochila 0/1	49
9.8.	Cambio de moneda	50
9.9.	Distancia de edición	50
9.10.	Más problemas clásicos	52
<b>10.</b>	<b>Otros: Greedy(Ávidos), Backtracking(Vuelta atrás)</b>	<b>52</b>
10.1.	Greedy	52
10.1.1.	Ejemplos de problemas No Aptos para Greedy	54
10.2.	Backtracking	54
10.3.	Características	55
10.4.	Cuándo usar backtracking	57
<b>11.</b>	<b>Cuestiones y problemas</b>	<b>58</b>
<b>V</b>	<b>Ordenación</b>	<b>59</b>
<b>12.</b>	<b>HeapSort</b>	<b>60</b>
<b>13.</b>	<b>MergeSort(fusión)</b>	<b>62</b>
<b>14.</b>	<b>Quicksort</b>	<b>64</b>
<b>15.</b>	<b>Quickselect</b>	<b>66</b>
15.1.	Resumen ordenación/búsqueda	68
<b>16.</b>	<b>Cuestiones y problemas</b>	<b>68</b>

<b>VI Búsqueda exacta en texto (Exact string matching)</b>	<b>71</b>
<b>17. Búsqueda secuencial</b>	<b>71</b>
<b>18. Autómata</b>	<b>72</b>
<b>19. Karp-Rabin</b>	<b>74</b>
<b>20. Knuth-Morris-Pratt</b>	<b>75</b>
<b>21. Boyer-Moore</b>	<b>77</b>
21.1. Horspool	77
<b>22. Cuestiones y problemas</b>	<b>79</b>
<b>23. Análisis de Complejidad Detallado</b>	<b>80</b>
23.1. Complejidad Temporal	80
23.2. Complejidad Espacial	82
23.3. Relación entre las Heurísticas	82
23.4. Comparación de Algoritmos	83
<b>24. Implementaciones en C</b>	<b>84</b>
24.1. Fuerza Bruta	84
24.2. KMP (Knuth-Morris-Pratt)	84
24.3. Boyer-Moore	85
24.4. Rabin-Karp	86
24.5. Boyer-Moore-Horspool (BMH)	87
24.6. Boyer-Moore-Sunday (BMS)	88
<b>VII Repaso Matemáticas para Ciencias de la Computación</b>	<b>90</b>
24.7. Potencias	90
24.8. Logaritmos	90
24.8.1. Identidades y propiedades	90

24.9. Piso( <i>floor</i> ) y techo( <i>ceiling</i> )	91
24.9.1. Identidades y propiedades	91
24.10. Factorial y coeficiente Binomial	91
24.10.1. Identidades y propiedades	92
24.11. Sumatorias	92
24.11.1. Identidades y propiedades	93
24.12. Series	94
24.12.1. Matrices	94
24.13. Hanoi	96

---



La palabra *algoritmo* proviene del nombre del matemático del siglo 9, Abū'Abd Allāh Muhammad bin Mūsā **al-Khwārizmī**, nacido en la antigua ciudad de Kwarizm , ahora llamada Khiva, en la provincia de Khorezm de Uzbekistan. Además escribió en Bagdad cerca del 820 DC, el famoso libro The Compendious Book on Calculation by Completion and Balancing (al-Kitāb al-Mukhtaṣar fī ḥisāb al-Jabr wal-Muqābalah), que en su título contiene la palabra "álgebra", por eso también es citado como **padre del álgebra**.

# Referencias

PARTE



**Introduction to Algorithms** by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.

**Algorithms** by Robert Sedgewick and Kevin Wayne

**Data Structures and Algorithms** by Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, John Hopcroft, Ullman D. Jeffrey

**The Art of Computer Programming** by Donald Knuth

**Data Structures and Algorithm Analysis in C** by Mark Allen Weiss

**Introduction to Algorithms: A Creative Approach** by Udi Manber

**Handbook of Algorithms and Data Structures** by Gaston H. Gonnet, Ricardo Baeza-Yates

**Fundamentos de algoritmia** by Brassard, G.; Bratley, P.

GeeksforGeeks  
Handbook Baeza-Gonnet  
Ejemplos MA Weiss

De que trata el curso:

Programación y resolución de problemas, con aplicaciones.

Algoritmo: método para resolver un problema y saber medir su eficiencia.

Uso de estructura de datos segun algoritmo.

Pero también:

Reutilizar código.

Entender un problema.

Saber aplicar un algoritmo a un problema.

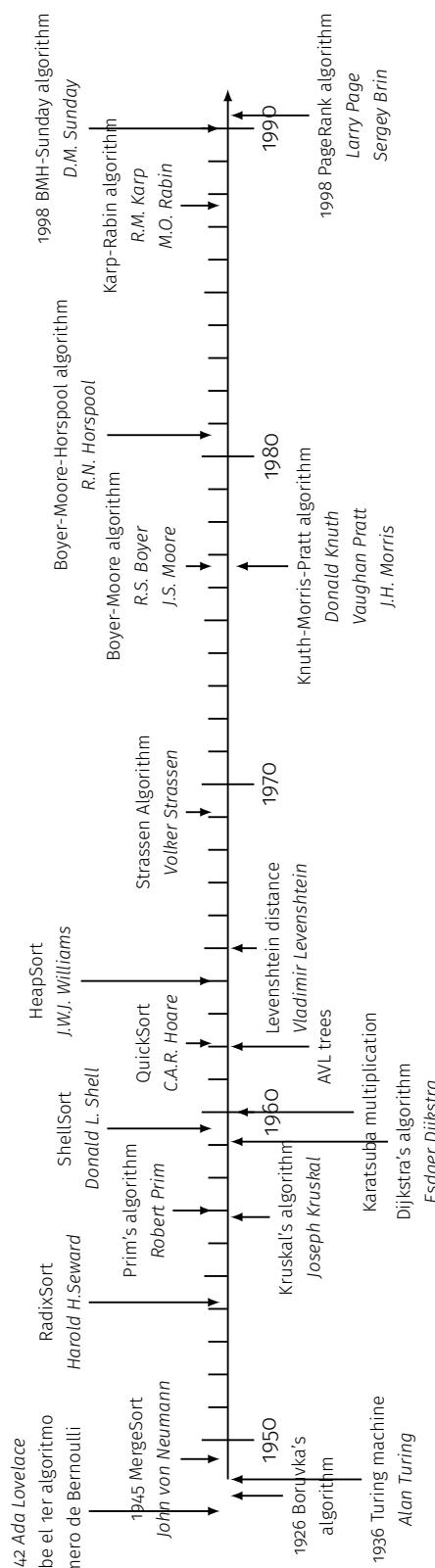
Usar un método de diseño para aplicar un algoritmo.

Usar lenguaje técnico/algorítmico.

Aplicar cultura informática.

Diferenciar los conceptos de la implementación.

1	Metas
	Conocer algoritmos clásicos para resolver problemas no triviales. Hacer análisis de la eficiencia de los algoritmos y aplicar técnicas de diseño de los mismos.
2	Malla (Año 3 4/Semestre 1)
	ING (V): Diseño de algoritmos, Req: Estructuras de datos CIVIL 2003(VII): Diseño de algoritmos, Req: Estructuras de datos CIVIL 2020(VII): Diseño de algoritmos, Req: Estructuras de datos
3	Evaluación
	40 % Pruebas



# Introducción

SECCIÓN 1

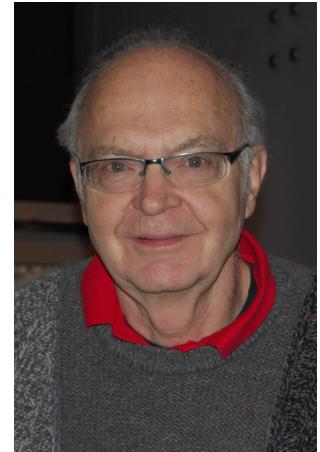
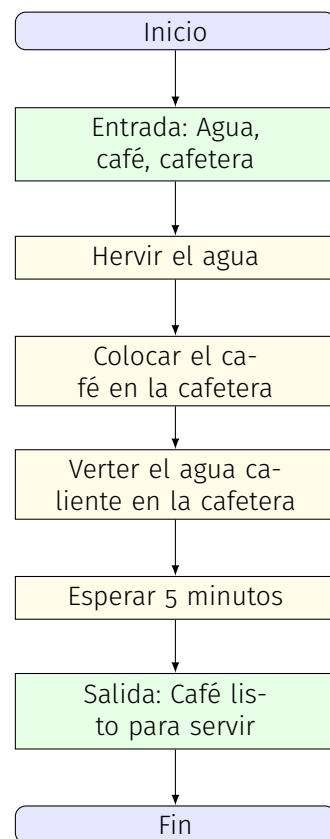
## Pero, ¿Qué es un algoritmo?

La idea intuitiva más general de un algoritmo es un procedimiento que consiste de un conjunto finito de instrucciones que, dada una entrada, nos permite obtener una salida si tal salida existe u obtener nada si no existe para esa entrada en particular. Todo esto a través de la ejecución sistemática de las instrucciones. Se requiere que un algoritmo se detenga en cada entrada, lo que implica que cada instrucción requiere una cantidad finita de tiempo, y cada entrada tiene una longitud finita.

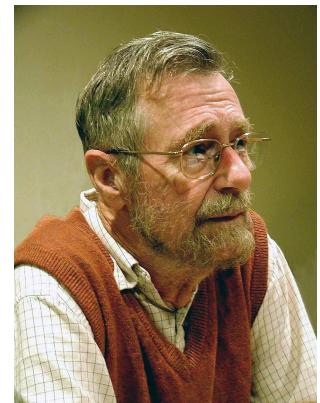
Según **Dijkstra**(1971), un algoritmo se corresponde con una descripción de un patrón de comportamiento, expresado en términos de un conjunto finito de acciones.

Por ejemplo, considera el proceso de preparar café como un algoritmo:

- **Entrada:** Agua, café, cafetera.
- **Instrucciones:**
  1. Hervir agua.
  2. Colocar el café en la cafetera.
  3. Verter agua caliente en la cafetera.
  4. Esperar 5 minutos.
- **Salida:** Café listo para servir.



**Figura 1.** Donald Ervin Knuth



**Figura 2.** Edsger Wybe Dijkstra

**Figura 3.** Descripción y diagrama de flujo para el proceso de preparar café.

También requerimos que la salida ante una entrada sea única, es decir, el algoritmo

es determinista en el sentido de que ejecuta el mismo conjunto de instrucciones ante una entrada en particular.

Existe una definición formal dada por **Knuth**, donde indica que un algoritmo debería tener 5 propiedades:

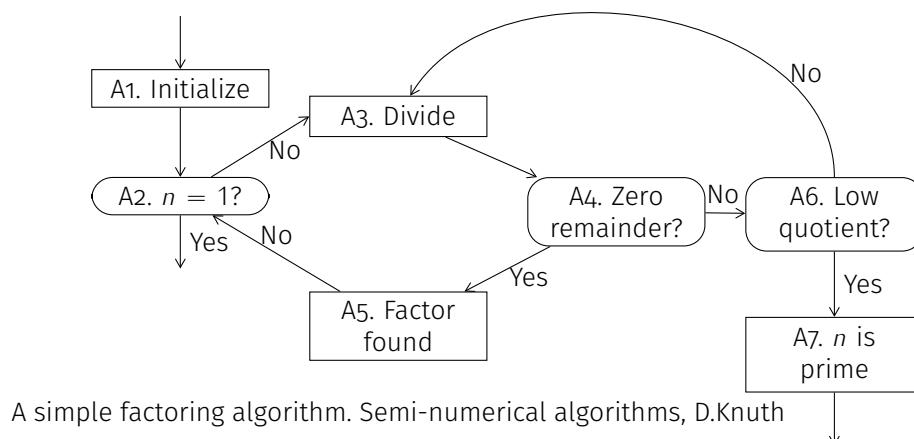
**Carácter finito:** siempre debe terminar después de un número finito de pasos.

**Precisión:** cada paso debe ser preciso, no ambiguo.

**Entrada:** puede tener o mas ingresos.

**Salida:** una o mas salidas, relacionada con la entrada.

**Eficacia:** todas las operaciones deben ser lo suficientemente básicas que terminan exactamente y de longitud finita.



**Figura 4.** Un algoritmo simple de factorización. Algoritmos semi-numéricos, D. Knuth.

#### SUBSECCIÓN 1.1

### Los algoritmos en la historia

Los algoritmos han sido una parte fundamental del desarrollo de las matemáticas y la informática a lo largo de la historia. A continuación, se presenta una breve referencia a su evolución a través de diferentes épocas:

#### 1.1.1. Antigüedad

En la antigüedad, uno de los algoritmos más conocidos es el Algoritmo de Euclides, utilizado para calcular el máximo común divisor (MCD) de dos números. Este algoritmo, que data del siglo III a.C., es un ejemplo temprano de un procedimiento sistemático para resolver un problema matemático.

### 1.1.2. Edad Media

Durante la Edad Media, el matemático persa Al-Juarismi hizo contribuciones significativas al desarrollo de los algoritmos. Su obra, "El libro de la suma y el balanceo" (Al-Kitāb al-Mukhtaṣar fī ḥisāb al-jabr wa-l-muqābala), fue traducida al latín en el siglo XII por Robert de Chester. La traducción, titulada "Algoritmi de numero Indorum", fue fundamental para introducir el álgebra en Europa. Esta obra es una de las razones por las que el término "algoritmo" se deriva del nombre de Al-Juarismi.

Además, la obra introdujo métodos sistemáticos para resolver ecuaciones lineales y cuadráticas, y es considerada una de las bases del álgebra moderna.

### 1.1.3. Edad Moderna

En la Edad Moderna, el trabajo de Alan Turing en las máquinas computacionales sentó las bases de la informática teórica. Turing introdujo el concepto de la máquina de Turing, un modelo abstracto de computación que formaliza la noción de algoritmo y computabilidad. Su trabajo ha influido profundamente en el desarrollo de la teoría de la computación y en la creación de las computadoras modernas.

Estos hitos históricos muestran cómo los algoritmos han evolucionado desde simples procedimientos matemáticos hasta convertirse en el núcleo de la informática moderna.



**Figura 5.** Línea del tiempo: evolución de los algoritmos.

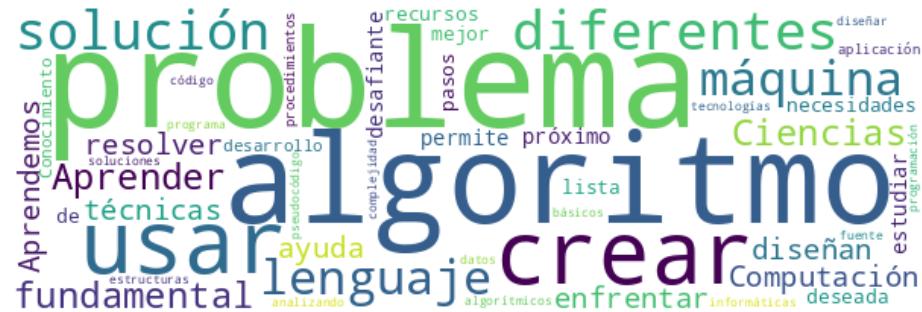
¿Porqué estudiar Algoritmos?

- Los algoritmos son fundamentales en las Ciencias de la Computación.
- Nos ayudan a resolver problemas eficientemente.
- Facilitan el diseño de soluciones óptimas y reutilizables.
- Son la base de áreas avanzadas como Machine Learning, criptografía y optimización.
- Permiten usar mejor los recursos computacionales.

Además de las aplicaciones clásicas, los algoritmos son fundamentales en áreas como:

- **Machine Learning (ML):** Clasificación de datos, predicción y detección de patrones.
- **Optimización logística:** Planificación de rutas de entrega.
- **Ciberseguridad:** Encriptación y análisis de vulnerabilidades.

- **Procesamiento de señales:** Aplicaciones como compresión de audio e imágenes.



## SUBSECCIÓN 1.2

## **Los límites de los algoritmos: P y NP**

## Complejidad Computacional: P, NP y NP-completo

En el ámbito de la teoría de la complejidad computacional, los problemas se clasifican en diferentes clases según la dificultad de resolverlos. Dos de las clases más importantes son *P* y *NP*.

### Problemas *P*

Estos son problemas que pueden ser resueltos en tiempo polinómico por una máquina determinista. En otras palabras, existe un algoritmo que puede resolver cualquier instancia del problema en un tiempo que es una función polinómica del tamaño de la entrada. Un ejemplo clásico de un problema en *P* es la ordenación de una lista de números con el algoritmo QuickSort, que tiene complejidad  $O(n \log n)$ .

### Problemas *NP*

Esta clase incluye problemas para los cuales, si se proporciona una solución, esta puede ser verificada en tiempo polinómico por una máquina determinista. Sin embargo, no se sabe si todos los problemas en *NP* pueden ser resueltos en tiempo polinómico. Un ejemplo de un problema en *NP* es el problema de la satisfacibilidad booleana (SAT).

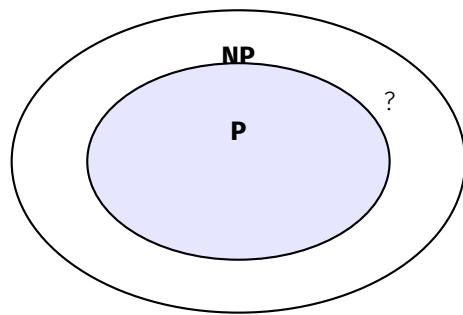
### Problemas *NP*-completos

Estos son los problemas más difíciles dentro de *NP*. Un problema es *NP*-completo si es al menos tan difícil como cualquier otro problema en *NP*, lo que significa que si se encuentra un algoritmo en tiempo polinómico para resolver un problema *NP*-completo, entonces todos los problemas en *NP* pueden ser resueltos en tiempo polinómico. El problema del viajante (TSP) es un ejemplo clásico de un problema *NP*-completo. En este problema, se busca encontrar el camino más corto que visita un conjunto de ciudades exactamente una vez y regresa a la ciudad de origen.

### Importancia en la informática

La distinción entre *P* y *NP* es fundamental en la informática teórica porque aborda la cuestión de si todos los problemas cuya solución puede ser verificada rápidamente también pueden ser resueltos rápidamente. Esta es una de las preguntas abiertas más importantes en la teoría de la computación. En la práctica, muchos problemas reales son *NP*-completos, y encontrar soluciones eficientes para ellos tiene un impacto significativo en campos como la optimización, la logística y la inteligencia artificial.

La relación entre *P* y *NP* sigue siendo una de las preguntas más importantes en la informática teórica. Si alguien prueba que  $P = NP$  o  $P \neq NP$ , cambiaría drásticamente cómo entendemos el mundo de los algoritmos.



**Figura 6.** Relación entre las clases  $P$  y  $NP$ .



# Análisis

SECCIÓN 2

## ¿Cómo medir un algoritmo?

Generalmente un problema computacional puede resolverse mediante diversos algoritmos o programas. El análisis de algoritmos es una actividad muy importante, especialmente en entornos con recursos restringidos. Es necesario para:

- Comparar algoritmos distintos
- Predecir el comportamiento de un algoritmo en circunstancias extremas
- Ajustar los parámetros de un algoritmo para obtener los mejores resultados

El análisis puede ser empírico(experimental) o teórico.

Eficiencia: capacidad de resolver el problema propuesto empleando un bajo consumo de recursos computacionales.

### Costo espacial, costo temporal

En ocasiones el tiempo y memoria son recursos competitivos y un buen algoritmo es aquel que resuelve el problema con un buen compromiso tiempo/memoria.

SECCIÓN 3

## Orden asintótico

Supongamos que tenemos 3 algoritmos que calculan lo mismo.

```
//A1.c
#include <stdio.h>

int main(){
    int n,m;
    scanf("%d",n);
    m=n*n;
    printf("%d",m);
}
```

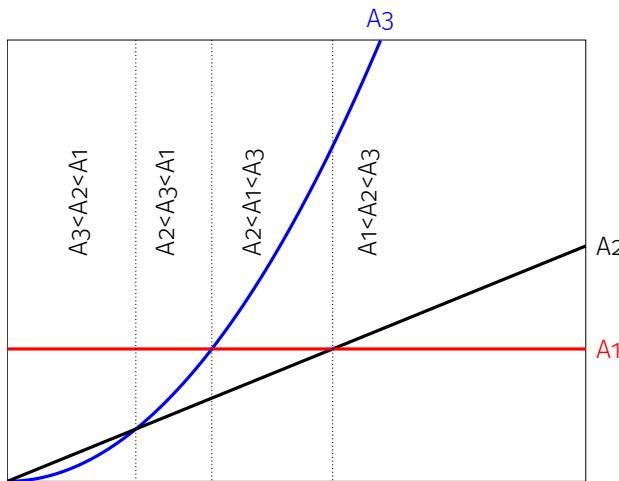
```
//A2.c
#include <stdio.h>

int main(){
    int i,n,m=0;
    scanf("%d",n);
    for(i=1; i <=n; i++)
        m=m+n;
    printf("%d",m);
}
```

```
//A3.c
#include <stdio.h>

int main(){
    int i,j,n,m=0;
    scanf("%d",n);
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            m=m+1;
    printf("%d",m);
}
```

Intuitivamente parece que el mejor programa es A1 y el peor A3. Pero, ¿cuál es el mejor? En general tendremos un comportamiento relativo tal como:



Una buena caracterización del costo computacional debería permitir establecer la calidad de un programa con independencia tanto del hardware como del tamaño de la entrada.

Ventajas de la caracterización asintótica:

- Generalmente los programas sólo son útiles para resolver problemas de gran tamaño (si es pequeño podríamos resolverlos manualmente sin dificultad)
- Al considerar sólo tamaños grandes, se pueden hacer aproximaciones sencillas que simplifican considerablemente el análisis del costo.
- La bondad relativa de distintos programas ya no depende de los valores concretos de los tiempos de ejecución de las distintas operaciones elementales empleadas (siempre que éstos no dependan del tamaño), ni del tamaño concreto de las instancias del problema a resolver.

Para simplificar el análisis del costo, generalmente se utiliza el concepto de *paso*: Un PASO es la ejecución de un segmento de código cuyo tiempo de proceso no depende del tamaño del problema considerado, o bien está acotado por alguna constante.

**Costo computacional de un programa:** Número de PASOS en función del tamaño del problema. Construcciones a las que se asigna un PASO:

- Asignación, operaciones aritméticas o lógicas, comparación, acceso a un elemento de vector o matriz, etc.
- Cualquier secuencia finita de estas operaciones cuya longitud no dependa del tamaño.

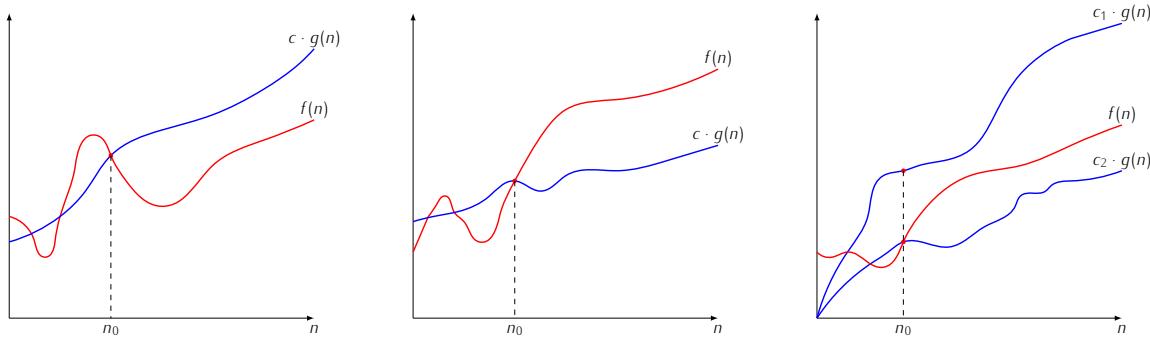
Construcciones a las que no se le puede asignar un PASO, sino un número de PASOS en función del tamaño:

- Asignación de variables estructuradas (ej. vectores) cuyo número de elementos dependa del tamaño
- Ciclos cuyo número de iteraciones dependa del tamaño

La comparación de funciones de costo debe ser insensible a "constantes de implementación" tales como los tiempos concretos de ejecución de operaciones individuales (cuyo costo sea independiente del tamaño). La independencia de las constantes se consigue considerando sólo el comportamiento asintótico de la función de costo (es decir, para tamaños "grandes"). A menudo ocurre que, para una tamaño dada, hay diversas instancias con costos diferentes, por lo que el costo no puede expresarse propiamente como función de la tamaño. En estas ocasiones conviene determinar cotas superiores e inferiores de la función costo; es decir, en el mejor caso y en el peor caso. Hay situaciones en las que incluso las cotas para mejores y peores casos son funciones complicadas. Generalmente bastará determinar funciones simples que acotan superior e inferiormente los costos de todas las instancias para tamaños grandes.

### Notación asintótica

Abstracción de las constantes asociadas al concepto de "PASO", de la noción de tamaños "grandes" y de la de "cotas asintóticas":



$$\mathcal{O}(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}_0^+ \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N}, \forall n \geq n_0, f(n) \leq cg(n)\}$$

Se dice que  $f(n) = \mathcal{O}(g(n))$ .

$$\Omega(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}_0^+ \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N}, \forall n \geq n_0, f(n) \geq cg(n)\}$$

Se dice que  $f(n) = \Omega(g(n))$ .

$$\Theta(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}_0^+ \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N}, \forall n \geq n_0, c_1g(n) \geq f(n) \geq c_2g(n)\}$$

Se dice que  $f(n) = \Theta(g(n))$ .

### SUBSECCIÓN 3.1

## Simplificación

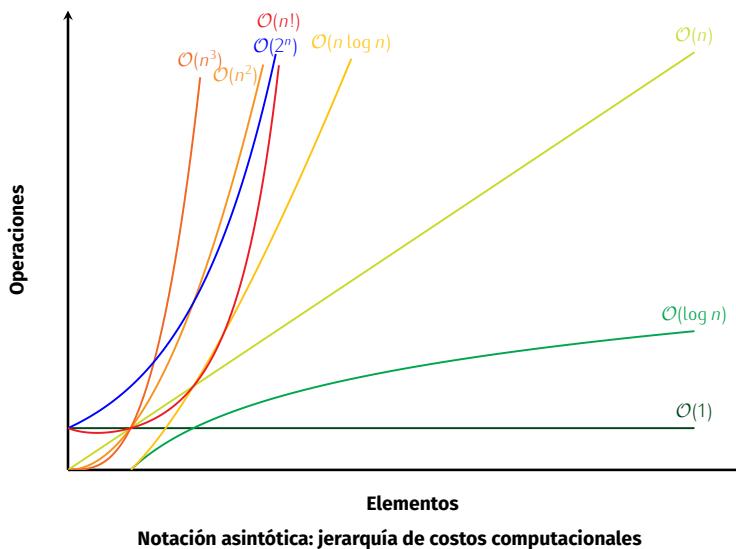
---

$$\mathcal{O}(c) = \mathcal{O}(1), c \in \mathbb{R}^+$$

$$\mathcal{O}(cf) = \mathcal{O}(f), c \in \mathbb{R}^+$$

$$\mathcal{O}(f + g) = \mathcal{O}(\max\{f, g\})$$

$$\mathcal{O}\left(\sum_{i=0}^k c_i n^i\right) = \mathcal{O}(n^k)$$



## SECCIÓN 4

## Análisis: Estimación de la ejecución

---

El tiempo de un algoritmo puede medirse en número de operaciones (comparaciones en otros casos). En forma empírica puede ser contando o midiendo el tiempo de las operaciones.

```
1 count = count+1;
2 sum = sum + count;
```

line	cost	times
1	$c_1$	1
2	$c_2$	1

$$T(n) = c_1 + c_2$$

```
1 if n<0
2   count = 0;
3 else
4   count = 1;
```

line	cost	times
1	$c_1$	1
2	$c_2$	1
4	$c_3$	1

$$T(n) = c_1 + \max(c_2, c_3)$$

```
1 i=1;
2 sum = 0;
3 while(i <= n){
4   i = i+1;
5   sum = sum + i;
6 }
```

line	cost	times
1	$c_1$	1
2	$c_2$	1
3	$c_3$	$n + 1$
4	$c_4$	$n$
5	$c_5$	$n$

$$T(n) = c_1 + c_2 + (n + 1) \cdot c_3 + n \cdot c_4 + n \cdot c_5$$

	line	cost	times
1	i=1;	$c_1$	1
2	sum = 0;	$c_2$	1
3	while(i <= n){	$c_3$	$n + 1$
4	j=1;	$c_4$	$n$
5	while (j <= n){	$c_5$	$n \cdot (n + 1)$
6	sum = sum + i;	$c_6$	$n \cdot n$
7	j=j+1;	$c_7$	$n \cdot n$
8	}	$c_8$	$n$
9	i = i+1;		
10	}		

$$T(n) = c_1 + c_2 + (n + 1) \cdot c_3 + n \cdot c_4 + (n + 1) \cdot c_5 + n^2 \cdot c_6 + n^2 \cdot c_7 + n \cdot c_8$$


---

## SUBSECCIÓN 4.1

**Peor caso, mejor caso y promedio**

A menudo el costo NO es (sólo) función de la tamaño. En los ejemplos vistos hasta ahora, todas las "instancias" de una tamaño dada tenían el mismo costo computacional. Pero esto no es siempre así.

	line	cost	times
1	void	$c_1$	
2	InsertionSort( int A[ ], int N ){	$c_2$	
3	int j, P;	$c_3$	$\sum_{j=1}^n j$
4	int Tmp;	$c_4$	$\sum_{j=1}^n (j - 1)$
5		$c_5$	$n - 1$
6	for( P = 1; P < N; P++ ){	$c_6$	
7	Tmp = A[ P ];	$c_7$	
8	for( j = P; j > 0 && A[ j - 1 ] > Tmp; j-- )	$c_8$	
9	A[ j ] = A[ j - 1 ];	$c_9$	
10	A[ j ] = Tmp;	$c_{10}$	
11	}		
12	}		

$$T(n) = c_1 \cdot n + c_2 \cdot (n - 1) + c_3 \cdot \sum_{j=1}^n j + c_4 \cdot \sum_{j=1}^n (j - 1) + c_5 \cdot (n - 1)$$

$$T(n) = c_1 \cdot n + c_2 \cdot n - c_2 + c_3 \cdot \sum_{j=1}^n j + c_4 \cdot \sum_{j=1}^n j - c_4 \cdot \sum_{j=1}^n 1 + c_5 \cdot n - c_5$$

Supongamos que las asignaciones tienen el mismo costo, esto es:  $c_2 = c_4 = c_5$ ,

$$T(n) = c_1 \cdot n + c_2 \cdot (n - 1) + c_3 \cdot \sum_{j=1}^n j + c_2 \cdot \sum_{j=1}^n (j - 1) + c_2 \cdot (n - 1)$$

$$T(n) = c_1 \cdot n + c_2 \cdot \left( 2 \cdot (n - 1) + \sum_{j=1}^n (j - 1) \right) + c_3 \cdot \sum_{j=1}^n j$$

Es claro que este tiempo no es independiente de las entradas dadas. Luego tenemos:

**mejor caso:** corresponde a cuando se realiza un número mínimo de operaciones. En este caso ocurre cuando el arreglo A ya está ordenado, y en ese caso el ciclo

interno no se ejecuta. Entonces:

$$\begin{aligned} T(n) &= c_1 \cdot n + 2 \cdot c_2 \cdot (n - 1) \\ T(n) &= (c_1 + 2c_2)n - 2c_2 \\ T(n) &= a \cdot n + b \in \mathcal{O}(n) \end{aligned}$$

**peor caso:** corresponde cuando se ejecuta el número máximo de operaciones. En este caso, cuando el ciclo interno se ejecuta todas las veces posibles. En este caso se tiene:

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot \left( 2 \cdot (n - 1) + \sum_{j=1}^n (j - 1) \right) + c_3 \cdot \sum_{j=1}^n j \\ T(n) &= \frac{c_2 + c_3}{2} n^2 + \left( c_1 + \frac{3c_2 + c_3}{2} \right) n - 2c_2 \\ T(n) &= a \cdot n^2 + b \cdot n + c \in \mathcal{O}(n^2) \end{aligned}$$

**casos típicos o promedios:** corresponde cuando se da una instancia aleatoria generada uniformemente entre todas las posibles. Para este caso podemos pensar que cada una de las iteraciones del ciclo interno se ejecuta la mitad de las operaciones posibles, es decir, que el valor actual queda aproximadamente en el medio del arreglo a su izquierda. En este caso se tiene:

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot \left( 2 \cdot (n - 1) + \sum_{j=1}^n \frac{j - 1}{2} \right) + c_3 \cdot \sum_{j=1}^n \frac{j}{2} \\ T(n) &= \frac{c_2 + c_3}{4} n^2 + \left( c_1 + \frac{7c_2 + c_3}{4} \right) n - 2c_2 \\ T(n) &= a \cdot n^2 + b \cdot n + c \in \mathcal{O}(n^2) \end{aligned}$$

```

1  for( i = 1; i <= n; i++ )
2    for( j = 1; j <= i; j++ )
3      for( k = 1; k <= j; k++ )
4        x=x+1;

```

line	cost	times
1	$c_1$	$n + 1$
2	$c_2$	$\sum_{j=1}^n j + 1$
3	$c_3$	$\sum_{j=1}^n \sum_{k=1}^j k + 1$
4	$c_4$	$\sum_{j=1}^n \sum_{k=1}^j k$

$$\begin{aligned} T(n) &= c_1 \cdot (n + 1) + c_2 \cdot \sum_{j=1}^n (j + 1) + c_3 \cdot \sum_{j=1}^n \sum_{k=1}^j (k + 1) + c_4 \cdot \sum_{j=1}^n \sum_{k=1}^j k \\ T(n) &= a \cdot n^3 + b \cdot n^2 + c \cdot n + d \in \mathcal{O}(n^3) \end{aligned}$$

## SUBSECCIÓN 4.2

### Mínimo

---

$$T(n) = n - 1 \in \mathcal{O}(n)$$

**Algoritmo 1** Búsqueda del elemento mínimo

---

```

1: function MINIMO( $V$ )                                 $\triangleright V[0..n - 1]$ 
2:    $m \leftarrow V[0]$ 
3:   for  $i \leftarrow 1$  to  $n - 1$  do
4:     if  $V[i] < m$  then
5:        $m \leftarrow V[i]$ 
6:   return  $m$ 

```

---

**4.2.1. Búsqueda secuencial****Algoritmo 2** Búsqueda secuencial #1

---

```

1: procedure BSEC1( $V, x$ )       $\triangleright V[0..n - 1]$ 
2:    $r \leftarrow n$ 
3:    $i \leftarrow 0$ 
4:   while  $i \leq n - 1 \wedge r = n$  do
5:     if  $x = V[i]$  then
6:        $r \leftarrow i$ 
7:      $i \leftarrow i + 1$ 
8:   return  $r$ 

```

---

**Algoritmo 3** Búsqueda secuencial #2

---

```

1: procedure BSEC2( $V, x$ )       $\triangleright V[0..n - 1]$ 
2:    $r \leftarrow 0$ 
3:   while  $r \leq n - 1 \wedge x \neq V[r]$  do
4:      $r \leftarrow r + 1$ 
5:   return  $r$ 

```

---

$$\begin{aligned} t_{\min}(n) &= 1 \in O(1) \\ t_{\max}(n) &= n \in O(n) \\ \bar{t}(n) &\in O(n) \end{aligned}$$

```

//bseq2.c
#include <stdio.h>

int bseq2(int v[], int x, int n){
    int r=0;

    while((r <=n-1)&&(x != v[r])){
        r = r+1;
    }
    return r;
}

int main(){
    int a[] = {3,6,23,45,7,8,9,2,66};
    int x = 88;

    printf("%d\n",bseq2(a,x,9));
}

```

Compilación:  
gcc bseq2.c -o bseq2  
Ejecución:  
\$ ./bseq2  
9  
\$

Claramente el mejor caso se produce cuando  $r = 0$ . Sólo se realiza una comparación.  $t_{\min}(n) = 1 \in O(1)$ . El peor caso se produce cuando  $r = n - 1$  o  $r = n$ , en ambos casos se realizan  $n$  comparaciones, luego:  $t_{\max}(n) = n \in O(n)$ . En este caso se debe calcular el caso promedio, para ello se debe usar probabilidades.

Sea  $\alpha$  la probabilidad de que el elemento esté, y  $\beta = 1 - \alpha$  la probabilidad que no esté, luego:

$$\alpha = P(r \in [0, n - 1]), \beta = P(r = n)$$

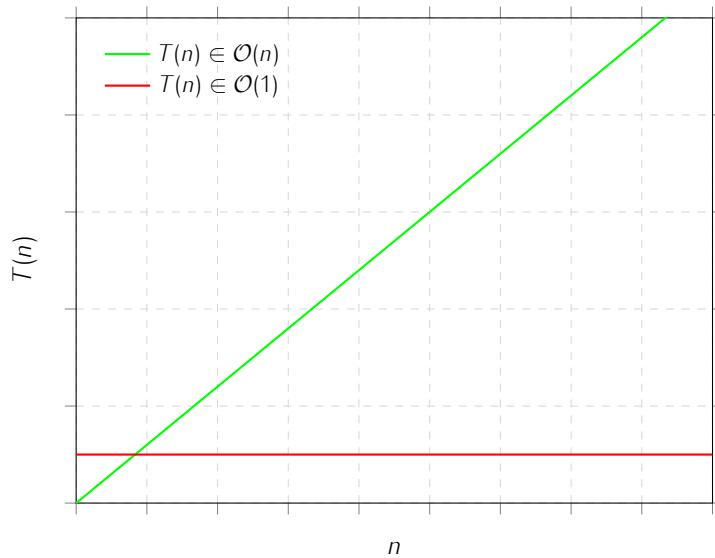
Hipótesis de equiprobabilidad:

$$P(r = 0) = P(r = 1) = \dots = P(r = n - 1) = \frac{\alpha}{n}$$

El número de comparaciones promedio es:

$$\begin{aligned}
 \bar{t}(n) &= \beta n + \sum_{i=0}^{n-1} \frac{\alpha}{n} i \\
 &= (1 - \alpha)n + \frac{\alpha}{n} \sum_{i=0}^{n-1} i \\
 &= (1 - \alpha)n + \frac{\alpha}{n} \left( \frac{n(n-1)}{2} \right) \\
 &= \frac{(2 - \alpha)n - \alpha}{2} \in O(n)
 \end{aligned}$$

ya que  $0 \leq \alpha \leq 1$



SECCIÓN 5

## **Ordenación por comparación**

---

SUBSECCIÓN 5.1

### **Intercambio**

### Bubble Sort

Método sencillo que consiste en revisar cada elemento del conjunto con el siguiente, intercambiándolos de posición si están en orden inverso. Es necesario revisar varias veces toda la lista hasta que ya no se necesiten intercambios. Este método también se llama por intercambio directo.

```

1 void bubbleSort(int v[], int n) {
2     int i, j;
3
4     for (i = 0; i < n - 1; i++) {
5         for (j = 0; j < n - i - 1; j++) {
6             if (v[j] > v[j + 1])
7                 swap(&v[j], &v[j + 1]);
8         }
9     }
10 }
```

### Algoritmo 4 Ordenación por intercambio

---

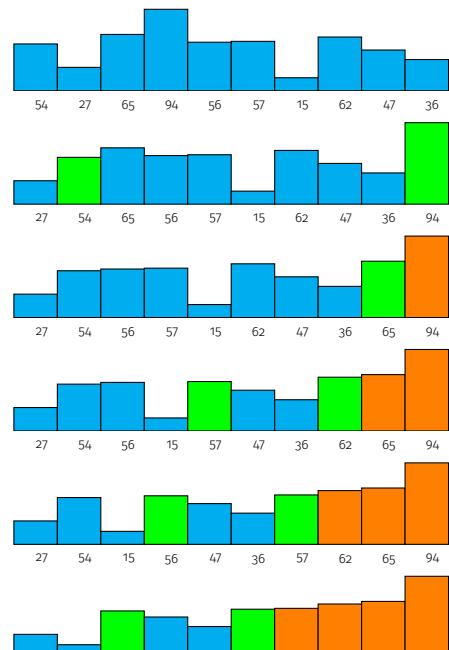
```

1: procedure SWAPSORT( $V$ ) ▷  $V[0..n - 1]$ 
2:   for  $i \leftarrow 1$  to  $n - 1$  do
3:     for  $j \leftarrow n - 1$  to  $i$  do
4:       if  $V[j] < V[j - 1]$  then
5:          $V[j] \leftrightarrow V[j - 1]$  ▷  $swap(V[j], V[j - 1])$ 
```

---

$$T(n) \in O(n^2)$$

0:	54	27	65	94	56	57	15	62	47	36
1:	27	54	65	56	57	15	62	47	36	94
2:	27	54	56	57	15	62	47	36	65	94
3:	27	54	56	15	57	47	36	62	65	94
4:	27	54	15	56	47	36	57	62	65	94
5:	27	15	54	47	36	56	57	62	65	94
6:	15	27	47	36	54	56	57	62	65	94
7:	15	27	36	47	54	56	57	62	65	94
8:	15	27	36	47	54	56	57	62	65	94
9:	15	27	36	47	54	56	57	62	65	94
	15	27	36	47	54	56	57	62	65	94



SUBSECCIÓN 5.2

## **Selección**

### Selection Sort

Este método consiste en seleccionar el menor elemento del conjunto y a continuación intercambiarlo con el elemento que ocupa la primera posición del vector. Hay que repetir esta operación con los  $n - 1$  elementos restantes, luego los  $n - 2$  elementos restantes y así sucesivamente hasta que solo quede un elemento.

```

1 void selectionSort(int v[], int n) {
2     int i, j, min_idx;
3
4     for (i = 0; i < n - 1; i++) {
5         min_idx = i;
6         for (j = i + 1; j < n; j++)
7             if (v[j] < v[min_idx])
8                 min_idx = j;
9
10        swap(&v[min_idx], &v[i]);
11    }
12 }
```

### Algoritmo 5 Ordenación por Selección

---

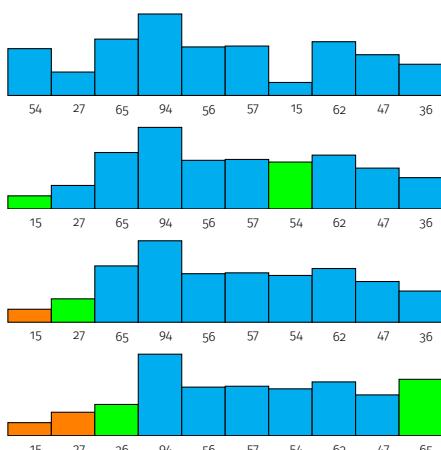
```

1: procedure SELECTIONSORT( $V$ ) ▷  $V[0..n - 1]$ 
2:   for  $i \leftarrow 1$  to  $n - 1$  do
3:      $k \leftarrow i - 1$ 
4:      $m \leftarrow V[i - 1]$ 
5:     for  $j \leftarrow i$  to  $n - 1$  do
6:       if  $V[j] < m$  then
7:          $k \leftarrow j$ 
8:          $m \leftarrow V[j]$ 
9:        $V[k] \leftarrow V[i - 1]$ 
10:       $V[i - 1] \leftarrow m$ 
```

---

$$T(n) \in O(n^2)$$

0:	54	27	65	94	56	57	15	62	47	36
1:	15	27	65	94	56	57	54	62	47	36
2:	15	27	65	94	56	57	54	62	47	36
3:	15	27	36	94	56	57	54	62	47	65
4:	15	27	36	47	56	57	54	62	94	65
5:	15	27	36	47	54	57	56	62	94	65
6:	15	27	36	47	54	56	57	62	94	65
7:	15	27	36	47	54	56	57	62	94	65
8:	15	27	36	47	54	56	57	62	94	65
9:	15	27	36	47	54	56	57	62	65	94
	15	27	36	47	54	56	57	62	65	94



$$t_{min}(n) = n - 1 \in O(n)$$

$$t_{max}(n) = \frac{n(n-1)}{2} \in O(n^2)$$

$$\bar{t}(n) \in O(n^2)$$

SUBSECCIÓN 5.3

## **Inserción**

### Insertion Sort

Método utilizado por los jugadores de cartas(naipes). En cada paso, a partir de  $i = 2$ , el  $i$ -ésimo elemento de la secuencia se procesa y se transfiere a la secuencia destino(que ya está ordenada), insertándolo en el lugar correspondiente.

```

1 void insertionSort(int v[], int n) {
2     int i, key, j;
3
4     for (i = 1; i < n; i++) {
5         key = v[i];
6         j = i - 1;
7
8         while (j >= 0 && v[j] > key) {
9             v[j + 1] = v[j];
10            j = j - 1;
11        }
12        v[j + 1] = key;
13    }
14 }
```

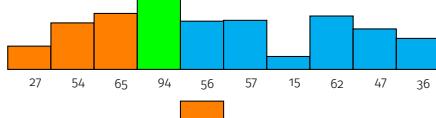
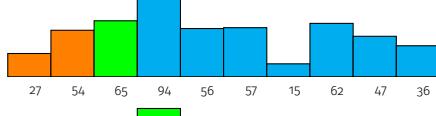
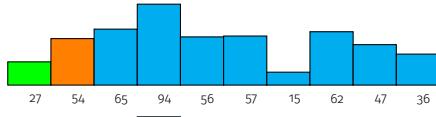
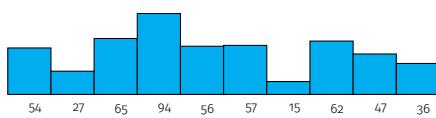
### Algoritmo 6 Ordenación por Inserción

```

1: procedure INSERTIONSORT( $V$ ) ▷  $V[0..n - 1]$ 
2:   for  $i \leftarrow 1$  to  $n - 1$  do
3:      $x \leftarrow V[i]$ 
4:      $j \leftarrow i - 1$ 
5:     while  $j \geq 1 \wedge x < V[j]$  do
6:        $V[j + 1] \leftarrow V[j]$ 
7:        $j \leftarrow j - 1$ 
8:      $V[j + 1] \leftarrow x$ 
```

$$T(n) \in \mathcal{O}(n^2), \in \mathcal{O}(n)$$

0:	54	27	65	94	56	57	15	62	47	36
1:	27	54	65	94	56	57	15	62	47	36
2:	27	54	65	94	56	57	15	62	47	36
3:	27	54	65	94	56	57	15	62	47	36
4:	27	54	56	65	94	57	15	62	47	36
5:	27	54	56	57	65	94	15	62	47	36
6:	15	27	54	56	57	65	94	62	47	36
7:	15	27	54	56	57	62	65	94	47	36
8:	15	27	47	54	56	57	62	65	94	36
9:	15	27	36	47	54	56	57	62	65	94
	15	27	36	47	54	56	57	62	65	94



## SECCIÓN 6

## Solución de ecuaciones de recurrencias

## SUBSECCIÓN 6.1

### Recurrencias homogéneas

Son de la forma:

$$a_0 T(n) + a_1 T(n-1) + a_2 T(n-2) + \dots + a_k T(n-k) = 0$$

que para resolver se deben buscar las soluciones que sean combinaciones de funciones exponenciales.

Para ello se realiza el cambio de variable  $x^k = T(n)$  y se obtiene la *ecuación característica*:

$$a_0 x^k + a_1 x^{k-1} + a_2 x^{k-2} + \dots + a_k = 0$$

Se denota por  $r_1, r_2, \dots, r_k$  a las raíces(reales o complejas) de la ecuación, podemos tener dos casos:

#### Caso #1: Raíces distintas

Solución de la recurrencia:

$$T(n) = c_1 r_1^n + c_2 r_2^n + \dots + c_k r_k^n = \sum_{i=1}^k c_i r_i^n$$

$c_i$  se determinan a partir de las condiciones iniciales.

*Ejemplo 1* Dada la siguiente recurrencia, determine la solución:

$$T(n) = T(n-1) + T(n-2), n \geq 2, T(0) = 0, T(1) = 1.$$

Haciendo  $x^2 = T(n)$ , se obtiene la ecuación característica  $x^2 = x + 1$ , cuyas raíces son:

$$r_1 = \frac{1 + \sqrt{5}}{2}, r_2 = \frac{1 - \sqrt{5}}{2}$$

Luego:

$$T(n) = c_1 \left( \frac{1 + \sqrt{5}}{2} \right)^n + c_2 \left( \frac{1 - \sqrt{5}}{2} \right)^n$$

Constantes  $c_1$  y  $c_2$  usando las condiciones iniciales:

$$\begin{aligned} T(0) &= c_1 \left( \frac{1+\sqrt{5}}{2} \right)^0 + c_2 \left( \frac{1-\sqrt{5}}{2} \right)^0 = c_1 + c_2 = 0 \\ T(1) &= c_1 \left( \frac{1+\sqrt{5}}{2} \right)^1 + c_2 \left( \frac{1-\sqrt{5}}{2} \right)^1 = 1 \end{aligned}$$

De aquí:

$$c_1 = -c_2 = \frac{1}{\sqrt{5}}$$

Sustituyendo:

$$T(n) = \frac{1}{\sqrt{5}} \left( \frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left( \frac{1-\sqrt{5}}{2} \right)^n$$

Sea

$$\phi = \left( \frac{1+\sqrt{5}}{2} \right)$$

Entonces:

$$T(n) = \frac{1}{\sqrt{5}} (\phi^n - \hat{\phi}^n) \quad T(n) \in O(\phi^n)$$

## Caso #2: Raíces iguales

---

Supongamos que se repite  $r_1$  (multiplicidad  $m > 1$ ). La ecuación característica es de la forma:

$$(x - r_1)^m (x - r_2) \dots (x - r_{k-m+1}) = 0$$

Solución de la recurrencia:

$$T(n) = \sum_{i=1}^m c_i n^{i-1} r_1^n + \sum_{i=m+1}^k c_i r_{i-m+1}^n$$

$c_i$  se determinan a partir de las condiciones iniciales.

Ejemplo 2 Encontrar la solución de la siguiente ecuación de recurrencia:

$$T(n) = 5T(n-1) - 8T(n-2) + 4T(n-3), \quad n \geq 2, \quad T(k) = k, \quad \text{para } k = 0, 1, 2$$

Ecuación característica:

$$x^3 - 5x^2 + 8x - 4 = 0 \quad o \quad (x-2)^2(x-1) = 0$$

Entonces:

$$T(n) = c_1 2^n + c_2 n 2^n + c_3 1.$$

$$c_1 = 2, c_2 = -1/2 \quad y \quad c_3 = -2$$

Finalmente:

$$T(n) = 2^{n+1} - n 2^{n-1} - 2.$$

$$T(n) \in O(n 2^n)$$

## SUBSECCIÓN 6.2

**Ecuaciones no homogéneas**

Considerar la ecuación de la forma:

$$a_0 T(n) + a_1 T(n-1) + \dots + a_k T(n-k) = b^n p(n)$$

$a_i$  y  $b$  son números reales  
 $p(n)$  polinomio en  $n$  de grado  $d$

Ecuación característica:

$$(a_0 x^k + a_1 x^{k-1} + \dots + a_k)(x - b)^{d+1} = 0$$

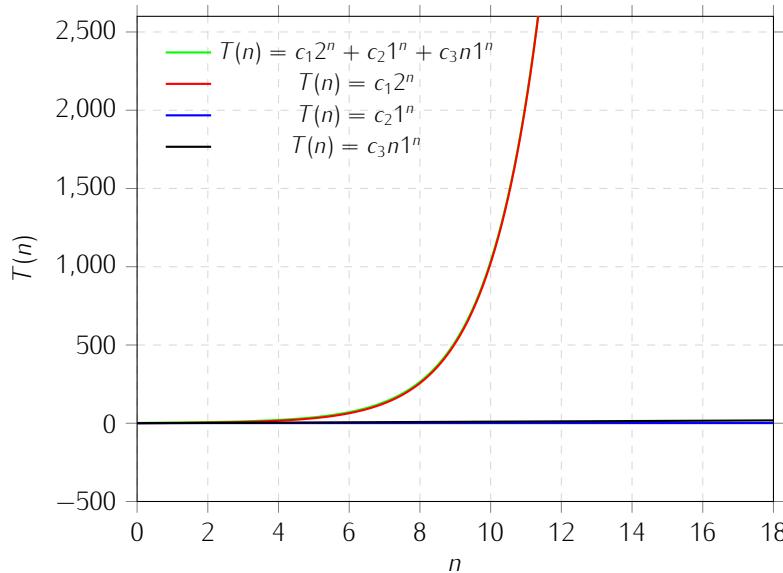
Ejemplo 3

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

De ahí se ve claramente que  $b = 1$ ,  $p(n) = n^1$  y  $d = 1$   
 Ec. característica  $(x - 2)(x - 1)^2 = 0$ , por tanto

$$T(n) = c_1 2^n + c_2 1^n + c_3 n 1^n$$

$$T(n) \in O(2^n)$$



## SUBSECCIÓN 6.3

**Cambio de variables**

Se aplica cuando  $n$  es potencia de un real  $a$ , osea  $n = a^k$ .

Ejemplo 4

$$T(n) = 4T\left(\frac{n}{2}\right) + n, \quad n > 3$$

$$T(1) = 1 \text{ y } T(2) = 6$$

Si  $n = 2^k$  podemos escribir:

$$T(2^k) = 4T(2^{k-1}) + 2^k$$

$$\frac{2^k}{2} = 2^{-1}2^k = 2^{k-1}$$

Cambio de variable nuevamente,  $t_k = T(2^k)$ :

$$t_k = 4t_{k-1} + 2^k$$

Esta es una ec. no homogénea (en  $k$ ), cuya solución es:

$$t_k = c_1(2^k)^2 + c_22^k$$

Deshaciendo los cambios:

$$T(n) = c_1n^2 + c_2n$$

Calculando  $c_1$  y  $c_2$  de las condiciones iniciales,  $c_1 = 2$  y  $c_2 = -1$ .

$$T(n) = 2n^2 - n$$

$$T(n) \in O(n^2)$$

#### SUBSECCIÓN 6.4

## Iteraciones

---

También se dice desenrollando.

*Ejemplo 5* Resuelva la siguiente ecuación de recurrencia. Suponga que  $p > q^r$ .

$$T(n) = pT\left(\frac{n}{q}\right) + kn^r, T(1) = 1$$

Solución: Desenrollando la ecuación  $j - 1$  veces:

$$T(n) = kn^r \left( 1 + \frac{p}{q^r} + \left( \frac{p}{q^r} \right)^2 + \cdots + \left( \frac{p}{q^r} \right)^{j-1} \right) + p^j T\left(\frac{n}{q^j}\right)$$

que es lo mismo que:

$$T(n) = kn^r \left( \left( \frac{p}{q^r} \right)^0 + \left( \frac{p}{q^r} \right)^1 + \left( \frac{p}{q^r} \right)^2 + \cdots + \left( \frac{p}{q^r} \right)^{j-1} \right) + p^j T\left(\frac{n}{q^j}\right)$$

que es:

$$T(n) = kn^r \sum_{i=0}^{j-1} \left( \frac{p}{q^r} \right)^i + p^j T\left(\frac{n}{q^j}\right)$$

Usando:

$$\sum_{j=1}^n c^j = \frac{c^{n+1} - 1}{c - 1}, c \neq 1$$

Como  $p > q^r$  (en este caso):

$$T(n) = kn^r \frac{\left(\frac{p}{q^r}\right)^j - 1}{\frac{p}{q^r} - 1} + p^j T\left(\frac{n}{q^j}\right)$$

Suponiendo que  $n = q^j \Rightarrow \log_q n = j$ :

$$T(n) = kn^r \frac{\left(\frac{p}{q^r}\right)^{\log_q n} - 1}{\frac{p}{q^r} - 1} + p^{\log_q n} T(1)$$

Dado que:

$$\left(\frac{p}{q^r}\right)^{\log_q n} = \frac{(q^{\log_q p})^{\log_q n}}{q^{r \log_q n}} = \frac{(q^{\log_q n})^{\log_q p}}{q^{\log_q n^r}} = \frac{n^{\log_q p}}{n^r}$$

y que  $T(1) = 1$  y considerando  $K' = \frac{k}{\frac{p}{q^r} - 1}$ :

$$T(n) = K'n^r \left( \frac{n^{\log_q p}}{n^r} - 1 \right) + n^{\log_q p} = (K' + 1)n^{\log_q p} - K'n^r$$

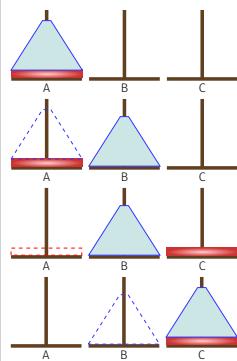
Luego:

$$T(n) \in O(n^{\log_q p})$$

Pendiente: 1.  $p = q^r$ , 2.  $p < q^r$

Ejemplo 6

### Torres de Hanoi



### Algoritmo 7 Torres de Hanoi

---

```

1: procedure HANOI(disc, source, destination, auxiliary)
2:   if disc = 1 then
3:     mover disc from source to destination
4:   else
5:     HANOI(disc - 1, source, auxiliary, destination)
6:     mover disc from source to destination
7:     HANOI(disc - 1, auxiliary, destination, source)

```

---

$$T(n) = T(n - 1) + 1 + T(n - 1)$$

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

Usando solución para recurrencias no homogéneas:

$$T(n) - 2T(n-1) = 1$$

Claramente se aprecia que  $b = 1$ ,  $p(n) = 1$ , luego  $d = 0$ .

Ecuación característica:

$$(x - 2)(x - 1) = 0$$

Cuya solución es:

$$T(n) = c_1 2^n + c_2 1^n$$

Como para  $n = 1$  disco se realiza un movimiento,  $T(1) = 1$ , para  $n = 2$ :

$$T(2) = 2T(1) + 1 = 3$$

Buscando las constantes, tenemos:

$$T(1) = c_1 2^1 + c_2 = 1$$

$$T(2) = c_1 2^2 + c_2 = 3$$

Cuya solución es,  $c_1 = 1$  y  $c_2 = -1$ .

Finalmente:

$$T(n) = 2^n - 1$$

$T(n)$  expresa la cantidad de movimientos dependiendo de la cantidad de discos.

Ej:  $T(4) = 2^4 - 1 = 15$ , 15 movimientos para 4 discos.

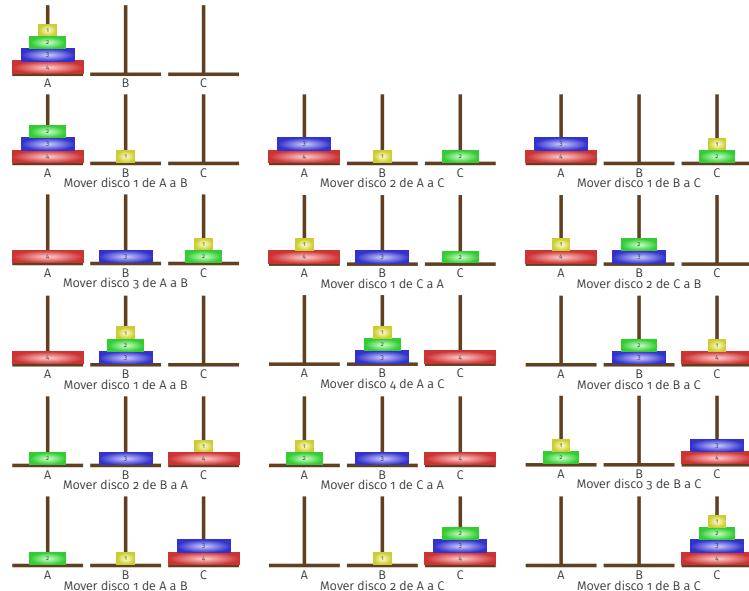
Secuencia Hanoi:

$n$	1	2	3	4	5
$T(n)$	1	3	7	15	31

```

void hanoi(int a, char from, char to, char aux){
    if(a==1)
        printf("\t\tMover disco 1 de %c a %c\n",from,to);
    else{
        hanoi(a-1,from,aux,to);
        printf("\t\tMover disco %d de %c a %c\n",a,from,to);
        hanoi(a-1,aux,to,from);
    }
}
Se llama:
hanoi(4, 'A', 'C', 'B');

```



$$\begin{aligned}T(1) &= 1 \\T(n) &= 8(2 \cdot T(n/2) + 1),\end{aligned}$$

Análisis:

$$\begin{aligned}T(n) &= 16 \cdot T(n/2) + 8 \\&= 16(16T(n/2^2) + 8) + 8 \\&= 16^2T(n/2^2) + 16 \cdot 8 + 8 \\&= 16^3T(n/2^3) + 16^2 \cdot 8 + 16 \cdot 8 + 8 \\&= 16^3T(n/2^3) + 16^2 \cdot 8 + 16^1 \cdot 8 + 16^0 \cdot 8 \\&= \dots \\&= 16^kT(n/2^k) + \dots + 16^2 \cdot 8 + 16 \cdot 8 + 16^0 \cdot 8\end{aligned}$$

Ejemplo 7

$$\begin{aligned}&= 16^kT(n/2^k) + 8 \sum_{i=0}^{k-1} 16^i \\&= 16^kT(n/2^k) + 8 \left( \frac{16^k}{15} - \frac{1}{15} \right)\end{aligned}$$

Como  $2^k = n$ ,  $16^k = 2^{4k} = (2^k)^4$

$$\begin{aligned}&= (2^k)^4T(n/2^k) + 8 \left( \frac{(2^k)^4}{15} - \frac{1}{15} \right) \\&= n^4 + \frac{8n^4}{15} + \frac{8}{15} \\&= \frac{23}{15}n^4 + \frac{8}{15}.\end{aligned}$$

```
int Test (int s, int t, int n) {
    int i;
    if (n==1) {
        if(found_between(s,t))
            return 1;
        else
            return 0;
    }
    for (i = 0; i <= 7; i++) {
        if(Test(s,t-1,n%2) && Test(s-1,t,n%2))
            return 1;
        else
            return 0;
    }
}
```

Ejemplo 8 Resolver, donde  $n$  es potencia de 2.

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 3T\left(\frac{n}{2}\right) + n & \text{si } n > 1 \end{cases}$$

Calculamos algunos valores de la secuencia:

$n$	1	2	4	8	16	32
$T(n)$	1	5	19	65	211	665

Ejemplo:

$$\begin{aligned}T(16) &= 3T(8) + 16 \\&= 3(65) + 16 \\&= 211\end{aligned}$$

De esta manera podemos ver que:

$n$	$T(n)$
$2^0$	1
$2^1$	$3^1 \cdot +2^1$
$2^2$	$3^2 \cdot 1 + 3 \cdot 2 + 2^2$
$2^3$	$3^3 \cdot 1 + 3^2 \cdot 2 + 3 \cdot 2^2 + 2^3$
$2^4$	$3^4 \cdot 1 + 3^3 \cdot 2 + 3^2 \cdot 2^2 + 3 \cdot 2^3 + 2^4$
$2^5$	$3^5 \cdot 1 + 3^4 \cdot 2 + 3^3 \cdot 2^2 + 3^2 \cdot 2^3 + 3 \cdot 2^4 + 2^5$
...	...

Para  $k$ -veces:

$$\begin{aligned}
T(2^k) &= 3^k \cdot 2^0 + 3^{k-1} \cdot 2^1 + 3^{k-2} \cdot 2^2 + \cdots + 3^1 \cdot 2^{k-1} + 3^0 \cdot 2^k \\
&= \sum_{i=0}^k 3^{k-i} \cdot 2^i \\
&= 3^k \sum_{i=0}^k \left(\frac{2}{3}\right)^i \\
&= 3^k \left[ \frac{1 - \left(\frac{2}{3}\right)^{k+1}}{1 - \frac{2}{3}} \right] \\
&= 3^k \left[ \frac{1 - \left(\frac{2}{3}\right)^{k+1}}{\frac{1}{3}} \right] \\
&= 3^{k+1} \left[ 1 - \left(\frac{2}{3}\right)^{k+1} \right] \\
&= 3^{k+1} - \frac{3^{k+1} 2^{k+1}}{3^{k+1}} \\
&= 3^{k+1} - 2^{k+1}
\end{aligned}$$

Como  $n = 2^k$ , entonces  $k = \lg n$ . Luego:

$$\begin{aligned}
T(n) &= T(2^{\lg n}) \\
&= 3^{1+\lg n} - 2^{1+\lg n}
\end{aligned}$$

Y  $3^{\lg n} = n^{\lg 3}$ . Entonces:

$$\begin{aligned}
T(n) &= 3^1 3^{\lg n} - 2^1 2^{\lg n} \\
&= 3n^{\lg 3} - 2n
\end{aligned}$$

Por lo tanto,  $T(n) \in \mathcal{O}(n^{\lg 3})$

Nota:  $\lg 3 = 1.585$

**7.1** Encuentre las soluciones asintóticas de las siguientes recurrencias:

- a)  $T(n) = T(n - 1) + 5n^2 - 3n$
- b)  $a_n = 5a_{n-1} + 6a_{n-2} = 0, n \geq 2, a_0 = 1, a_1 = 3$
- c)  $2a_{n+2} - 11a_{n+1} + 5a_n = 0, n \geq 0, a_0 = 2, a_1 = -8$
- d)  $3a_{n+1} = 2a_n + a_{n-1} = 0, n \geq 1, a_0 = 7, a_1 = 3$
- e)  $a_{n+2} + a_n = 0, n \geq 0, a_0 = 0, a_1 = 3$
- f)  $a_{n+2} + 4a_n = 0, n \geq 0, a_0 = a_1 = 1$
- g)  $a_n - 6a_{n-1} + 9a_{n-2} = 0, n \geq 2, a_0 = 5, a_1 = 12$
- h)  $a_n + 2a_{n-1} + 2a_{n-2} = 0, n \geq 2, a_0 = 1, a_1 = 3$
- i)  $a_{n+1} - a_n = 2n + 3, n \geq 0, a_0 = 1$
- j)  $a_{n+1} - a_n = 3n^2 - n, n \geq 0, a_0 = 3$
- k)  $a_{n+1} - a_n = 5, n \geq 0, a_0 = 1$
- l)  $a_n + na_{n-1} = n!n \geq 1, a_0 = 1$
- m)  $a_{n+1} - 2a_n = 2^n, n \geq 0, a_0 = 1$

**7.2** Si  $a_n, n \geq 0$ , es una solución de la relación de recurrencia  $a_{n+1} - da_n = 0$  y  $a_3 = 153/49$ ,  $a_5 = 1377/2401$ , ¿cuánto vale  $d$ ?

**7.3** Si  $a_0 = 0, a_1 = 1, a_2 = 4$  y  $a_3 = 37$  satisfacen la relación de recurrencia  $a_{n+2} + ba_{n+1} + ca_n = 0$ , donde  $n \geq 0$  y  $b, c$  son constantes, encuentre  $a_n$ .

**7.4** Resolver  $a_{n+2} - 6a_{n+1} + 9a_n = 3(2^n) + 7(3^n), n \geq 0, a_0 = 1, a_1 = 4$

**7.5** Resuelva la recurrencia, determine el orden y compruebe por Teorema Maestro:

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ 3T\left(\frac{n}{4}\right) + \Theta(\sqrt{n}) & \text{si } n > 0 \end{cases}$$

**7.6** Encuentre la complejidad de la siguiente función:

```

1 int func1 (int n){
2     int i=1;
3     while(i<n){
4         int j=n;
5         while(j>0)
6             j = j/2;
7             i=2*i;
8     }
9 }
```

**7.7** Considere el siguiente algoritmo para ordenar un arreglo de enteros de tamaño  $n$ :

- Dividir el arreglo en  $\frac{n}{k}$  pedazos de tamaño  $k$
- Ordenar cada pedazo usando ordenamiento por inserción
- Mezclar los pedazos

Analice este algoritmo obteniendo el orden temporal del mismo. En base a su resultado ¿Qué valor de  $k$  usaría Ud.? Justifique.

**7.8** Indicar cual es el objetivo del siguiente código. Además encuentre del orden temporal.

```

1 int prog1 ( int a[], int x, int i, int j )
2 {
3     if ( i > j )
4         return -1;
5     if ( i == j ){
6         if ( a[i] == x )
7             return i;
8         else
9             return -1;
10    }
11    else{
12        int n1 = ( 2 * i + j ) / 3;
13        int n2 = ( i + 2 * j ) / 3;
14
15        if ( x <= a[n1] )
16            j = n1;
17        else if ( x >= a[n2] )
18            i = n2;
19        else{
20            i = n1 + 1;
21            j = n2 - 1;
22        }
23        return prog1 ( a, x, i, j );
24    }
25 }

```

- 7.9** Hallar una ecuación de recurrencia lineal homogénea y sus condiciones iniciales cuyo término general sea:  $a_n = 3^{n+2} + n3^{n-2}$ .

- 7.10** Determine  $\text{sum}[1 \dots k]$

Contar

```

1: function COUNT( $n$ ) ▷  $n = k^2$  para algún  $k$  entero
2:    $k \leftarrow \sqrt{n}$ 
3:   for  $j \leftarrow 1$  to  $k$  do
4:      $\text{sum}[j] \leftarrow 0$ 
5:     for  $i \leftarrow 1$  to  $j^2$  do
6:        $\text{sum}[j] \leftarrow \text{sum}[j] + i$ 
return  $\text{sum}[1 \dots k]$ 

```

- 7.11** ¿Cuál es el valor returned por la siguiente función? Exprese su respuesta en función de  $n$ . Usando notación  $O$ , indique el peor caso en la ejecución.

```

1 int pesky(int n){
2     int i,j,k;
3     int r = 0;
4
5     for(i=1; i <= n-1; i++)
6         for(j=1; j <= i; j++)
7             for(k=j; k <= i+j; k++)
8                 r = r+1;
9     return(r);
10 }

```

- 7.12** Escribir el código en C para sumar dos matrices de  $nxm$  y dejar el resultado en una tercera matriz. Indique el orden de su código. Repita el problema para multiplicación de matrices.

- 7.13** Diseñe un algoritmo para calcular la moda de un conjunto de valores, de tamaño  $n$ . Su algoritmo debe ser  $O(n)$ .

- 7.14** Indique la relación de inclusión entre los diferentes órdenes:

$O(n), O(\log n), O(\sqrt{n}), O(n^3), O(n^n), O(1), O(n \log n), O(2^n), O(n^2)$ .

Ejemplo  $O(n) \subset O(n \log n)$

**7.15** Dada la ecuación de recurrencia,

$$f(n) = \begin{cases} d & \text{si } n = 1 \\ af(n/c) + bn^x & \text{si } n \geq 2 \end{cases}$$

determinar la solución, considerando el caso en que  $a = c^x$  y el caso  $a \neq c^x$

**7.16** Considere el siguiente algoritmo: suponga que la operación crucial es examinar un elemento. El algoritmo examina los  $n$  elementos de un conjunto y, de alguna manera, eso le permite descartar  $2/5$  de los elementos para entonces realizar una llamada recursiva sobre los restantes  $3/5$  elementos. Escriba una ecuación de recurrencia que describa este comportamiento.

**7.17** Escriba la ecuación de recurrencia.

```

1 int sort(int A[], int n, int i, int j){ // n potencia de 3
2     int k;
3
4     if(i<j){
5         k = ((j-i)+1)/3;
6         sort(A,n,i,i+k-1);
7         sort(A,n,i+k,i+2k-1);
8         sort(A,n,i+2k,j);
9         Merge(A,i,i+k,i+2k,j);
10        // Merge intercala
11        // A[i..(i+k-1)],A[(i+k)..(i+2k-1)] y
12        // A[i+2k .. j] en A[i..j] con un costo de 5n/3-2
13    }
14 }
```

# Diseño

El diseño de algoritmos es un proceso esencial en la informática que se centra en la creación de procedimientos sistemáticos para resolver problemas computacionales. Un algoritmo es una secuencia de pasos bien definidos que toma una entrada y produce una salida deseada. El diseño de algoritmos no solo busca encontrar una solución, sino también la solución más eficiente posible, optimizando el uso de recursos como el tiempo de ejecución y la memoria.

La importancia de encontrar soluciones eficientes radica en la capacidad de manejar grandes volúmenes de datos y realizar cálculos complejos en un tiempo razonable. En un mundo donde la cantidad de datos y la complejidad de los problemas crecen exponencialmente, los algoritmos eficientes son cruciales para el rendimiento de aplicaciones en áreas como la inteligencia artificial, la bioinformática, la logística y más. Un algoritmo bien diseñado puede significar la diferencia entre una aplicación práctica y una que es inviable debido a sus requerimientos de recursos.

En este contexto, existen varias estrategias de diseño de algoritmos que se utilizan para abordar problemas complejos. A continuación, se describen brevemente algunas de las más comunes:

- **Divide y Vencerás:** Esta estrategia implica dividir un problema en subproblemas más pequeños, resolver cada subproblema de manera recursiva y luego combinar sus soluciones para obtener la solución del problema original.
- **Programación Dinámica:** Resuelve problemas complejos dividiéndolos en subproblemas más simples, resolviendo cada uno de manera iterativa y almacenando sus soluciones para evitar cálculos redundantes.
- **Greedy (Voraz):** Consiste en tomar decisiones que parecen óptimas en el momento, con la esperanza de encontrar una solución globalmente óptima.
- **Backtracking (Retroceso):** Se utiliza para resolver problemas de búsqueda y optimización, explorando todas las posibles soluciones de manera sistemática.
- **Branch and Bound (Ramificación y Acotación):** Similar al backtracking, pero con una técnica adicional para acotar el espacio de búsqueda.
- **Algoritmos de Búsqueda Local:** Comienzan con una solución inicial y la mejoran iterativamente mediante pequeñas modificaciones.
- **Algoritmos Genéticos:** Inspirados en la evolución biológica, utilizan técnicas como la selección, cruce y mutación para evolucionar soluciones.
- **Programación Lineal y Entera:** Utiliza técnicas matemáticas para optimizar una función objetivo sujeta a restricciones lineales.
- **Algoritmos de Monte Carlo:** Utilizan técnicas de muestreo aleatorio para obtener aproximaciones a soluciones de problemas complejos.

Cada una de estas estrategias tiene sus propias ventajas y limitaciones, y la elección de la estrategia adecuada depende de la naturaleza del problema a resolver. Comprender estas técnicas es esencial para diseñar algoritmos eficientes y efectivos en una amplia variedad de aplicaciones.

## SECCIÓN 8

## Divide y Vencerás

---

El paradigma «Divide y Vencerás» (DyV) es una técnica de diseño de algoritmos que se basa en dividir un problema en subproblemas más pequeños, resolver estos subproblemas de manera recursiva y combinar sus soluciones para obtener la solución del problema original.

## SUBSECCIÓN 8.1

### Características

---

1. **División:** El problema se divide en subproblemas más pequeños del mismo tipo.
2. **Conquista:** Los subproblemas se resuelven recursivamente.
3. **Combinación:** Las soluciones de los subproblemas se combinan para obtener la solución del problema original.

## SUBSECCIÓN 8.2

### Estructura general

---

#### **Algoritmo 8** Estructura de Divide y Vencerás

---

```

1: function DyV(problema)
2:   if tamaño(problema) ≤ tamaño_minimo then
3:     return resolver_directamente(problema)
4:   subproblemas ← dividir(problema)
5:   soluciones ← ∅
6:   for cada subproblema en subproblemas do
7:     soluciones ← soluciones ∪ DyV(subproblema)
8:   return combinar(soluciones)

```

---

## SUBSECCIÓN 8.3

### Ecuaciones de Recurrencia

---

La complejidad de un algoritmo DyV se puede expresar mediante una ecuación de recurrencia de la forma:

$$T(n) = \begin{cases} c & \text{si } n \leq n_0 \\ aT(n/b) + f(n) & \text{si } n > n_0 \end{cases}$$

Donde:

- $T(n)$  es el tiempo de ejecución para un problema de tamaño  $n$
- $a$  es el número de subproblemas en que se divide el problema
- $b$  es el factor de reducción del tamaño del problema
- $f(n)$  es el costo de dividir y combinar
- $n_0$  es el tamaño mínimo para resolver directamente
- $c$  es el costo de resolver un problema de tamaño mínimo

SUBSECCIÓN 8.4

## Casos Especiales

---

### 8.4.1. División en Dos Partes Iguales

$$T(n) = \begin{cases} c & \text{si } n \leq n_0 \\ 2T(n/2) + f(n) & \text{si } n > n_0 \end{cases}$$

### 8.4.2. División en Tres Partes Iguales

$$T(n) = \begin{cases} c & \text{si } n \leq n_0 \\ 3T(n/3) + f(n) & \text{si } n > n_0 \end{cases}$$

SUBSECCIÓN 8.5

## Teorema Maestro

---

Para resolver ecuaciones de recurrencia de la forma:

$$T(n) = aT(n/b) + f(n)$$

Donde  $a \geq 1$ ,  $b > 1$  y  $f(n)$  es una función asintóticamente positiva, se puede aplicar el Teorema Maestro:

1. Si  $f(n) = O(n^{\log_b a - \epsilon})$  para alguna constante  $\epsilon > 0$ , entonces:

$$T(n) = O(n^{\log_b a})$$

2. Si  $f(n) = O(n^{\log_b a} \log^k n)$  para alguna constante  $k \geq 0$ , entonces:

$$T(n) = O(n^{\log_b a} \log^{k+1} n)$$

3. Si  $f(n) = \Omega(n^{\log_b a + \epsilon})$  para alguna constante  $\epsilon > 0$ , y si  $af(n/b) \leq cf(n)$  para alguna constante  $c < 1$  y todo  $n$  suficientemente grande, entonces:

$$T(n) = O(f(n))$$

SUBSECCIÓN 8.6

## Ejemplos Clásicos

---

### 8.6.1. Merge Sort

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

Complejidad:  $O(n \log n)$

### 8.6.2. Multiplicación de Matrices de Strassen

$$T(n) = \begin{cases} c & \text{si } n \leq 1 \\ 7T(n/2) + n^2 & \text{si } n > 1 \end{cases}$$

Complejidad:  $O(n^{\log_2 7}) \approx O(n^{2.81})$

En este caso:

- $a = 7$  (número de subproblemas): El algoritmo divide cada matriz en 4 submatrices y realiza 7 multiplicaciones recursivas
- $b = 2$  (factor de reducción): Cada submatriz tiene tamaño  $n/2 \times n/2$
- $f(n) = n^2$  (costo de dividir y combinar): El costo de sumar y restar matrices

### 8.6.3. Quick Sort (caso promedio)

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

Complejidad:  $O(n \log n)$

### 8.6.4. Búsqueda Binaria

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

Complejidad:  $O(\log n)$

SUBSECCIÓN 8.7

## Ventajas y Desventajas

### ■ Ventajas:

- Fácil de implementar
- Eficiente para problemas grandes
- Permite paralelización

### ■ Desventajas:

- Overhead de recursión
- No siempre es la mejor solución
- Puede requerir memoria adicional

Otra forma general de la ecuación de recurrencia:

$$T(n) = \begin{cases} cn^k & \text{si } 1 \leq n < b \\ aT\left(\frac{n}{b}\right) + cn^k & \text{si } n \geq b \end{cases}$$

que tiene por solución:

$$T(n) \in \begin{cases} O(n^k) & \text{si } a < b^k \\ O(n^k \log n) & \text{si } a = b^k \\ O(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

SUBSECCIÓN 8.8

## Búsqueda Binaria

Si  $k = 1$  se llama técnica de reducción.

```
int binarySearchRec(int X[], int l, int r, int key){
    if (l > r)
        return -1;
    else
    {
        int mid = l + (r - 1) / 2;
        if (X[mid] == key)
            return mid;
        if (X[mid] > key)
            return binarySearchRec(X, l, mid - 1, key);
        else
            return binarySearchRec(X, mid + 1, r, key);
    }
}
```

```
int binarySearchIt(int X[], int l, int r, int key){
    while (l <= r){
        int mid = l + (r - 1) / 2;
        if (X[mid] == key)
            return mid;
        if (X[mid] < key)
            l = mid + 1;
        else
            r = mid - 1;
    }
    return -1;
}
```

En el caso iterativo y el recursivo se puede ver que:

$$T(n) = \begin{cases} O(1) & \text{si } n \leq 1 \\ T\left(\frac{n}{2}\right) + O(1) & \text{si } n > 1 \end{cases}$$

Luego  $T(n) = c_1 \log n$ ,  $T(n) \in O(\log n)$ . La búsqueda binaria mas que ser una técnica DV pura, es un caso de simplificación.

SUBSECCIÓN 8.9

## Exponenciación

Dado un número  $a$  y un entero positivo  $n$ , suponga que se desea computar  $a^n$ . El método estandar o ingenuo (naïve method) propone un ciclo simple para realizar  $n - 1$  multiplicaciones por  $a$ :

### Algoritmo 9 Slow Power

---

```

1: function SLOWPOWER( $a, n$ ) ▷  $a^n$ 
2:    $x \leftarrow a$ 
3:   for  $i \leftarrow 2$  to  $n$  do
4:      $x \leftarrow x \cdot a$ 
      return  $a$ 

```

---

Ejemplo 9 |  $3^{27} = 3 \cdot 3 = 7625597484987$

Este algoritmo iterativo necesita  $n$  multiplicaciones.

Existe un método mucho más rápido, llamado Pingala, que utiliza la siguiente fórmula recursiva:

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ (a^{\frac{n}{2}})^2 & \text{si } n > 0 \text{ y } n \text{ es par} \\ (a^{\lfloor \frac{n}{2} \rfloor})^2 \cdot a & \text{otro caso} \end{cases}$$



**Figura 7.** Acharya Pingala

### Algoritmo 10 Pingala Power

---

```

1: function PINGALAPOWER( $a, n$ ) ▷  $a^n$ 
2:   if  $n = 1$  then
3:     return  $a$ 
4:   else
5:      $x \leftarrow \text{PINGALAPOWER}(a, \lfloor n/2 \rfloor)$ 
6:     if  $n$  es par then
7:       return  $x \cdot x$ 
8:     else
9:       return  $x \cdot x \cdot a$ 

```

---

Ejemplo 10 | Pingala:

$$\begin{aligned}
3^{27} &= 3 \cdot (3^{13})^2 \\
&= 3 \cdot (3 \cdot (3^6)^2)^2 \\
&= 3 \cdot (3 \cdot ((3^3)^2)^2)^2 \\
&= 3 \cdot (3 \cdot ((3 \cdot 3^2)^2)^2)^2 \\
&= 3 \cdot (3 \cdot ((3 \cdot (3 \cdot 3))^2)^2)^2 \\
&= 7625597484987
\end{aligned}$$

Este algoritmo satisface la recurrencia  $T(n) \leq T(n/2) + 2$ . Cuya solución es  $T(n) =$

$O(\log n)$ . Existe otra identidad que se puede utilizar:

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ (a^2)^{\frac{n}{2}} & \text{si } n > 0 \text{ y } n \text{ es par} \\ (a^2)^{\lfloor \frac{n}{2} \rfloor} \cdot a & \text{otro caso} \end{cases}$$

Cuyo algoritmo es:

---

### Algoritmo 11 Peasant Power

---

```

1: function PEASANTPOWER( $a, n$ )  $\triangleright a^n$ 
2:   if  $n = 1$  then
3:     return  $a$ 
4:   else if  $n$  es par then
5:     return PEASANTPOWER( $a^2, n/2$ )
6:   else
7:     return PEASANTPOWER( $a^2, \lfloor n/2 \rfloor$ )  $\cdot a$ 

```

---

Ejemplo 11 Peasant:

$$\begin{aligned} 3^{27} &= 3 \cdot (3^2)^{13} \\ &= 3 \cdot (3 \cdot (3 \cdot (3^2)^2)^6) \\ &= 3 \cdot (3 \cdot (3 \cdot ((3^2)^2)^2)^3) \\ &= 3 \cdot (3 \cdot (3 \cdot (3 \cdot (3^2)^2)^2))^2 \\ &= 7625597484987 \end{aligned}$$

Este algoritmo también requiere de  $O(\log n)$  multiplicaciones. Aunque ambos métodos son efectivos, no son óptimos en algunos casos. Por ejemplo:

- Pingala:  $a \rightarrow a^2 \rightarrow a^3 \rightarrow a^6 \rightarrow a^7 \rightarrow a^{14} \rightarrow a^{15}$
- Peasant:  $a \rightarrow a^2 \rightarrow a^4 \rightarrow a^8 \rightarrow a^{12} \rightarrow a^{14} \rightarrow a^{15}$
- Óptimo:  $a \rightarrow a^2 \rightarrow a^3 \rightarrow a^5 \rightarrow a^{10} \rightarrow a^{15}$

SUBSECCIÓN 8.10

## Mínimo y Máximo

---

Cada *min* o *max* realiza una comparación, luego:

$$t(n) = \begin{cases} 0 & \text{si } n = 1 \\ t\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + t\left(\left\lceil \frac{n}{2} \right\rceil\right) + 2 & n > 1 \end{cases}$$

No es el mejor algoritmo ya que  $t(n) = 2n - 2$ . Esto se puede mejorar:

No hay mejora asintótica pero disminuyen las comparaciones:  $t(n) = (3/2)n - 2$

SUBSECCIÓN 8.11

## Más problemas clásicos

**Algoritmo 12** Mínimo y Máximo

---

```

1: function MINMAX( $v, i, j$ ) ▷  $v[i..j]$ 
2:    $n \leftarrow j - i + 1$ 
3:   if  $n = 1$  then
4:      $\langle a, b \rangle \leftarrow \langle v[i], v[i] \rangle$ 
5:   else
6:      $k \leftarrow i - 1 + \lfloor n/2 \rfloor$ 
7:      $\langle c, d \rangle \leftarrow \text{MINMAX}(v, i, k)$ 
8:      $\langle e, f \rangle \leftarrow \text{MINMAX}(v, k + 1, j)$ 
9:      $\langle a, b \rangle \leftarrow \langle \min(c, e), \max(d, f) \rangle$ 
return  $\langle a, b \rangle$ 

```

---

**Algoritmo 13** Mínimo y Máximo

---

```

1: function MINMAX( $v, i, j$ ) ▷  $v[i..j]$ 
2:    $n \leftarrow j - i + 1$ 
3:   if  $n = 1$  then
4:      $\langle a, b \rangle \leftarrow \langle v[i], v[i] \rangle$ 
5:   else if  $n = 2$  then
6:     if  $v[i] \leq v[j]$  then
7:        $\langle a, b \rangle \leftarrow \langle v[i], v[j] \rangle$ 
8:     else
9:        $\langle a, b \rangle \leftarrow \langle v[j], v[i] \rangle$ 
10:   else
11:      $k \leftarrow i - 1 + \lfloor n/2 \rfloor$ 
12:      $\langle c, d \rangle \leftarrow \text{MINMAX}(v, i, k)$ 
13:      $\langle e, f \rangle \leftarrow \text{MINMAX}(v, k + 1, j)$ 
14:      $\langle a, b \rangle \leftarrow \langle \min(c, e), \max(d, f) \rangle$ 
return  $\langle a, b \rangle$ 

```

---

- MergeSort/QuickSort
- Encontrar Mediana y  $k$ -ésimo elemento más pequeño
- QuickSelect
- Skyline problem
- Multiplicación de enteros grandes
- Multiplicación de matrices(Strassen)
- Par de puntos más cercanos

## SECCIÓN 9

**Programación Dinámica**

La «Programación Dinámica» es una técnica de diseño de algoritmos que resuelve problemas complejos dividiéndolos en subproblemas más simples, almacenando las solu-

ciones de estos subproblemas para evitar recalcularlas. Es especialmente útil cuando un problema tiene subproblemas superpuestos y una estructura óptima.

SUBSECCIÓN 9.1

## Características Principales

---

1. **Subproblemas superpuestos:** El mismo subproblema se resuelve múltiples veces.
2. **Estructura óptima:** La solución óptima del problema contiene soluciones óptimas de sus subproblemas.
3. **Tabla de memorización:** Se almacenan las soluciones de los subproblemas para evitar recálculos.

SUBSECCIÓN 9.2

## Estructura General

---

### **Algoritmo 14** Estructura General de Programación Dinámica

```

1: function PD(problema)
2:   if solución ya calculada en tabla then
3:     return obtener_de_tabla(problema)
4:   if caso base then
5:     return resolver_directamente(problema)
6:   subproblemas ← dividir(problema)
7:   solución ← Ø
8:   for cada subproblema en subproblemas do
9:     solución ← combinar(solución, PD(subproblema))
10:    guardar_en_tabla(problema, solución)
11:   return solución

```

---

SUBSECCIÓN 9.3

## Tipos de Programación Dinámica

---

### 9.3.1. Top-Down (Memoización)

- Se resuelve el problema de forma recursiva
- Se almacenan las soluciones en una tabla
- Se evita recalcular subproblemas ya resueltos

### 9.3.2. Bottom-Up (Tabulación)

- Se resuelven los subproblemas más pequeños primero
- Se construye la solución progresivamente
- Se llena una tabla con todas las soluciones

SUBSECCIÓN 9.4

## Ventajas y Desventajas

---

### ■ Ventajas:

- Evita recálculos innecesarios
- Mejora significativamente la eficiencia en problemas con subproblemas superpuestos
- Garantiza la solución óptima

### ■ Desventajas:

- Requiere memoria adicional para almacenar la tabla
- No siempre es fácil identificar la estructura óptima
- Puede ser más complejo de implementar que soluciones ingenuas

SUBSECCIÓN 9.5

## Cuándo Usar Programación Dinámica

---

1. Cuando el problema tiene subproblemas superpuestos
2. Cuando el problema tiene una estructura óptima
3. Cuando se necesita optimizar una solución recursiva
4. Cuando el problema se puede dividir en subproblemas más pequeños

SUBSECCIÓN 9.6

## Fibonacci

---

Los números de Fibonacci están definidos como:

$$F_n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F_{n-1} + F_{n-2} & \text{en otro caso} \end{cases}$$

---

### Algoritmo 15

#### Fibonacci Recursivo

---

```

1: function RECFIB(n)
2:   if n = 0 then
3:     return 0
4:   else if n = 1 then
5:     return 1
6:   else
7:     return RECFIB(n - 1) + RECFIB(n - 2)

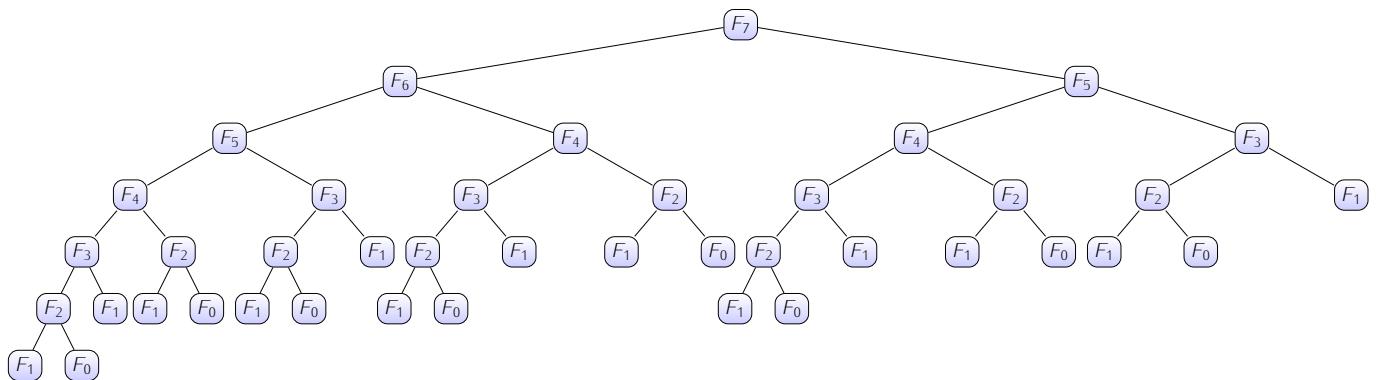
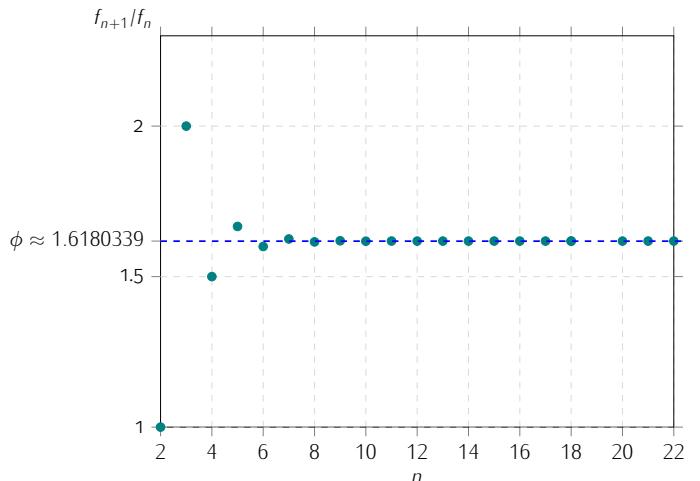
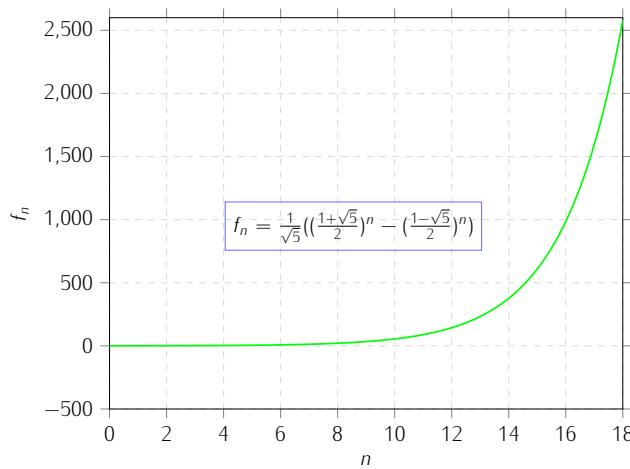
```

---



**Figura 8.** Leonardo de Pisa

Donde  $F_n \in O(\phi^n)$ , donde  $\phi = (1 + \sqrt{5})/2 \approx 1.61803$  se llama **razón aurea** o **golden ratio**.



Y si usamos memorización (memoria intermedia).

---

#### Algoritmo 16 Fibonacci Memorización

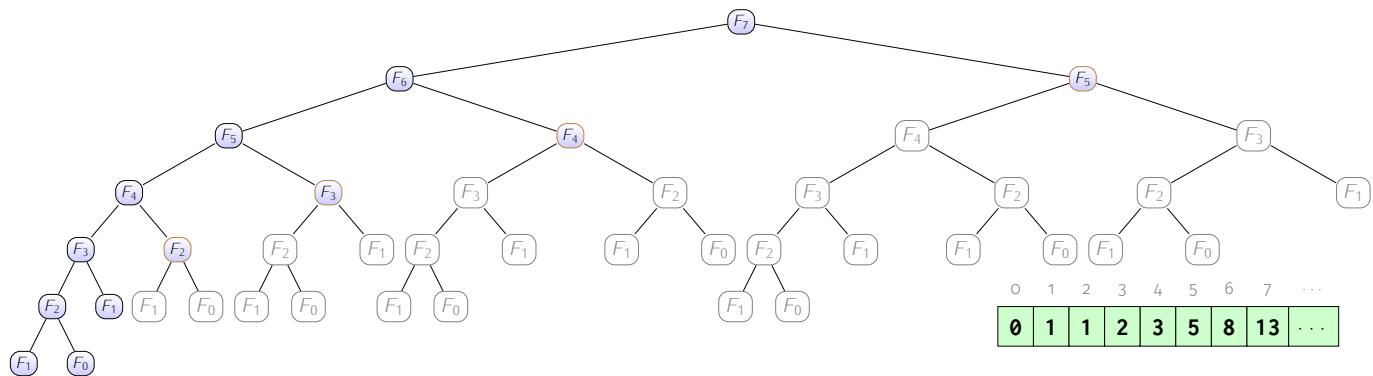
---

```

1: function MEMFIB( $n$ )
2:   if  $n = 0$  then
3:     return 0
4:   else if  $n = 1$  then
5:     return 1
6:   else if  $F[n]$  está indefinida then
7:      $F[n] \leftarrow \text{MEMFIB}(n - 1) + \text{MEMFIB}(n - 2)$ 
return  $F[n]$ 
```

---

Claramente se disminuye el tiempo de ejecución. ¿Cuánto?



Pero si reemplazamos la memorización recursiva con un simple ciclo **for** e intencionalmente llenamos el arreglo  $F[]$ .

---

### Algoritmo 17 Fibonacci Iterativo

---

```

1: function ITERFIB( $n$ )
2:    $F[0] \leftarrow 0$ 
3:    $F[1] \leftarrow 1$ 
4:   for  $i \leftarrow 2$  to  $n$  do
5:      $F[i] \leftarrow F[i - 1] + F[i - 2]$ 
return  $F[n]$ 

```

---

Este es el primer acercamiento al algoritmo de Programación Dinámica(*dynamic programming*).

### Pero no es necesario recordar todo

---

### Algoritmo 18 Fibonacci Iterativo2

---

```

1: function ITERFIB2( $n$ )
2:    $prev \leftarrow 1$ 
3:    $curr \leftarrow 0$ 
4:   for  $i \leftarrow 1$  to  $n$  do
5:      $next \leftarrow curr + prev$ 
6:      $prev \leftarrow curr$ 
7:      $curr \leftarrow next$ 
return  $curr$ 

```

---

### Más rápido

Si usamos la siguiente matriz para reformular la recurrencia de Fibonacci:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} y \\ x + y \end{bmatrix}$$

Esto tiene el mismo efecto que **IterFib2**

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix}$$

Podemos utilizar la identidad  $F_n = F_m F_{n-m-1} + F_{m+1} F_{n-m}$ , es decir:

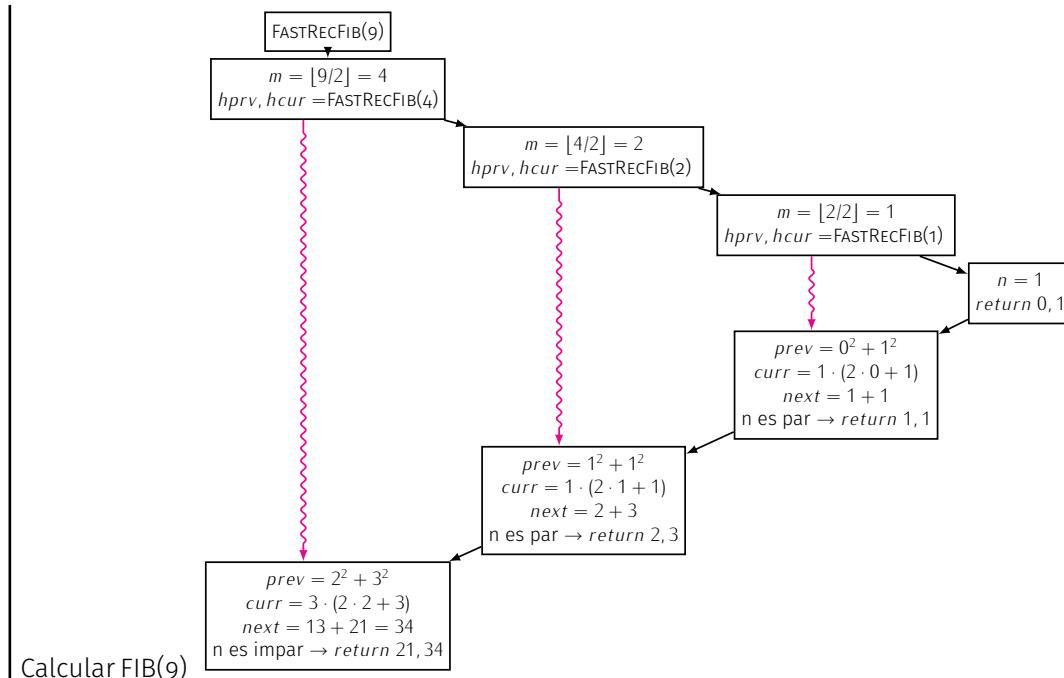
$$\begin{aligned} F_{2n-1} &= F_{n-1}^2 + F_n^2 \\ F_{2n} &= F_n(F_{n-1} + F_{n+1}) \\ &= F_n(2F_{n-1} + F_n) \end{aligned}$$

### Algoritmo 19 Fibonacci Recursivo Rápido

```

1: function FASTRECFIB( $n$ )                                ▷ Entrega  $F_{n-1}, F_n$ 
2:   if  $n = 1$  then
3:     return 0,1
4:    $m \leftarrow \lfloor n/2 \rfloor$ 
5:    $hprv, hcur \leftarrow \text{FASTRECFIB}(m)$ 
6:    $prev \leftarrow hprv^2 + hcur^2$ 
7:    $curr \leftarrow hcur \cdot (2 \cdot hprv + hcur)$ 
8:    $next \leftarrow prev + curr$ 
9:   if  $n$  es par then
10:    return  $prev, curr$ 
11:   else
12:    return  $curr, next$ 

```



#### 9.6.1. ¿Se puede hacer mejor?

SUBSECCIÓN 9.7

### Problema de la mochila o/1

Tenemos un conjunto  $S$  de  $n$  objetos, en el que cada objeto  $i$  tiene un beneficio  $b_i$  y un peso  $w_i$  positivos.

Objetivo: Seleccionar los elementos que garantizan un beneficio máximo pero con un peso global menor o igual que  $W$ .

Dado el conjunto  $S$  de  $n$  objetos, sea  $S_k$  el conjunto de los  $k$  primeros objetos (de 1 a  $k$ ):

Podemos definir  $B(k, w)$  como la ganancia de la mejor solución obtenida a partir de los elementos de  $S_k$  para una mochila de capacidad  $w$ .

Ahora bien, la mejor selección de elementos del conjunto  $S_k$  para una mochila de tamaño  $w$  se puede definir en función de selecciones de elementos de  $S_{k-1}$  para mochilas de menor capacidad.

¿Cómo calculamos  $B(k, w)$ ?

- O bien la mejor opción para  $S_k$  coincide con la mejor selección de elementos de  $S_{k-1}$  con peso máximo  $w$  (el beneficio máximo para  $S_k$  coincide con el de  $S_{k-1}$ ),,
- o bien es el resultado de añadir el objeto  $k$  a la mejor selección de elementos de  $S_{k-1}$  con peso máximo  $w - w_k$  (el beneficio para  $S_k$  será el beneficio que se obtenía en  $S_{k-1}$  para una mochila de capacidad  $w - w_k$  más el beneficio  $b_k$  asociado al objeto  $k$ ).

O sea:

$$B(k, w) = \begin{cases} B(k-1, w) & \text{si } x_k = 0 \\ B(k-1, w - w_k) + b_k & \text{si } x_k = 1 \end{cases}$$

Para resolver el problema de la mochila nos quedaremos con el máximo de ambos valores:

$$B(k, w) = \max\{B(k-1, w), B(k-1, w - w_k) + b_k\}$$

```
int [][] knapsack(W, w[1..n], b[1..n])
{
    for(p=0; p<=W; p++)
        B[0][p] = 0;

    for(k=1; k <= n; k++){
        for(p=0; p <= w[k]; p++)
            B[k][p] = B[k-1][p];
        for(p=w[k]; p <= W; p++)
            B[k][p] = max(B[k-1][p-w[k]]+b[k], B[k-1][p]);
    }
    return B;
}
```

Utilizamos:

- Si  $B[k][w] == B[k-1][w]$ , entonces el objeto  $k$  no se selecciona, se usa  $B[k-1][w]$
- Si  $B[k][w] != B[k-1][w]$ , se selecciona el objeto  $k$  y la solución es  $B[k-1][w-w[k]]$ .

Tiempo de ejecución es  $\Theta(nW)$ .

Ejemplo 13 | Tamaño mochila  $W = 11$ , Número de objetos  $n = 5$ .

	0	1	2	3	4	5	6	7	8	9	10	11
$\phi$	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0											
{1, 2}	0											
{1, 2, 3}	0											
{1, 2, 3, 4}	0											
{1, 2, 3, 4, 5}	0											

Objeto	Valor(b)	Peso(w)
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

max{B[k-1][p-w[k]]+b[k], B[k-1][p]}, k:3, p:5, w[k]:5, b[k]:18

	0	1	2	3	4	5	6	7	8	9	10	11
$\phi$	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1, 2}	0	1	6	7	7	7	7	7	7	7	7	7
{1, 2, 3}	0	1	6	7	7	18	19	24	25	25	25	25
{1, 2, 3, 4}	0	1	6	7	7	18	22	24	28	29	29	40
{1, 2, 3, 4, 5}	0	1	6	7	7	18	22	28	29	34	35	40

## SUBSECCIÓN 9.8

**Cambio de moneda**

Disponemos de  $n$  tipos de monedas de valor  $v_i$  y deseamos devolver un cambio de  $C$  unidades monetarias empleando el mínimo número posible de monedas de cada tipo.

Supondremos que tenemos una cantidad ilimitada de monedas de cada tipo.

Existen casos para los que no se puede aplicar el algoritmo *greedy*, por ejemplo, devolver \$ 8 con monedas de \$ 6, \$ 4 y \$ 1.

$$\text{cambio}(\{m_1, \dots, m_i\}, C) = \min \begin{cases} \text{cambio}(\{m_1, \dots, m_{i-1}\}, C) \\ 1 + \text{cambio}(\{m_1, \dots, m_i\}, C - m_i) \end{cases}$$

Ejemplo 14

	0	1	2	3	4	5	6	7	8
{1}	0	1	2	3	4	5	6	7	8
{1,4}	0	1	2	3	1	2	3	4	2
{1,4,6}	0	1	2	3	1	2	1	2	2

$C = 8$
$m_1 = 1$
$m_2 = 4$
$m_3 = 6$

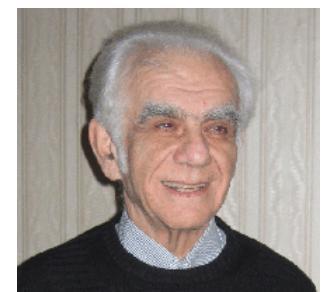
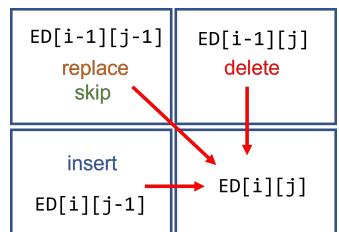


Figura 9. Stanisław Ulam

## SUBSECCIÓN 9.9

**Distancia de edición**

La distancia de edición (*edit distance*) entre dos cadenas es el mínimo número de letras(caracteres) insertadas, borradas o substituidas que requiere transformar una cadena en otra. También se llama *Levenshtein distance* o *Ulam distance*. Por ejemplo, la ED entre FOOD y MONEY es al menos 4:



FOOD → MOOD → MON\_D → MONED → MONEY

$$ED(i, j) = \begin{cases} i & \text{si } j = 0 \\ j & \text{si } i = 0 \\ \min \left\{ \begin{array}{l} ED(i, j - 1) + 1 \\ ED(i - 1, j) + 1 \\ ED(i - 1, j - 1) + A[i] \neq B[j] \end{array} \right\} & \text{en otro caso} \end{cases}$$

Casos:

- mismo carácter  $ED(i - 1, j - 1) + 0$
- Borrado  $ED(i - 1, j) + 1$
- Inserción  $ED(i, j - 1) + 1$
- Modificación  $ED(i - 1, j - 1) + 1$

### Algoritmo 20 Distancia de edición

```

1: function EDITDISTANCE( $A[1..m], B[1..n]$ )                                ▷  $A$  y  $B$ : cadenas
2:   for  $j \leftarrow 0$  to  $n$  do
3:      $ED[0, j] \leftarrow j$ 
4:   for  $i \leftarrow 1$  to  $m$  do
5:      $ED[i, 0] \leftarrow i$ 
6:     for  $j \leftarrow 1$  to  $n$  do
7:        $ins \leftarrow ED[i, j - 1] + 1$ 
8:        $del \leftarrow ED[i - 1, j] + 1$ 
9:       if  $A[i] = B[j]$  then
10:         $rep \leftarrow ED[i - 1, j - 1]$ 
11:      else
12:         $rep \leftarrow ED[i - 1, j - 1] + 1$ 
13:       $ED[i, j] \leftarrow \min\{ins, del, rep\}$ 
14:    return  $ED[m, n]$ 

```

ALGORITHM → ALTRUISTIC

	A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8
A	1	0	1	2	3	4	5	6	7
L	2	1	0	1	2	3	4	5	6
T	3	2	1	1	2	3	4	5	6
R	4	3	2	2	2	2	3	4	5
U	5	4	3	3	3	3	3	4	5
I	6	5	4	4	4	4	3	4	5
S	7	6	5	5	5	5	4	4	5
T	8	7	6	6	6	6	5	4	5
I	9	8	7	7	7	7	6	5	5
C	10	9	8	8	8	8	7	6	6
	A	L	G	O	R	I	T	H	M
	A	L	T	R	U	I	S	T	I

Ejemplo 15

A	L	G	O	R	I	T	H	M
A	L	T	R	U	I	S	T	I
A	L	G	O	R	I	T	H	M
A	L	T	R	U	I	S	T	I

SUBSECCIÓN 9.10

## Más problemas clásicos

- Números combinatorios
- Caminos mínimos
- Inversión de capital
- Problema del viajero
- Planificación de tareas
- Multiplicación de una secuencia de matrices
- Subsecuencia común mas larga(LCS)

SECCIÓN 10

## Otros: Greedy(Ávidos), Backtracking(Vuelta atrás)

SUBSECCIÓN 10.1

### Greedy

Dado un problema con  $n$  entradas el método consiste en obtener un subconjunto de éstas que satisfaga una determinada restricción definida para el problema. Siempre se elige la opción que ofrece el beneficio inmediato más alto o el menor costo, con la esperanza de que esta elección localmente óptima conduzca a una solución globalmente óptima. Cada uno de los subconjuntos que cumplen las restricciones diremos que son soluciones prometedoras. Una solución prometedora que maximice o minimice una función objetivo la denominaremos solución óptima.

- El enfoque es miope, las decisiones se toman utilizando únicamente la información disponible en cada paso, sin tener en cuenta los efectos que estas decisiones puedan tener en el futuro.
- Los algoritmos voraces resultan fáciles de diseñar, fáciles de implementar y, cuando funcionan, son eficientes.
- Se usan principalmente para resolver problemas de optimización

¿Por qué no utilizar siempre algoritmos voraces?

1. porque no todos los problemas admiten esta estrategia de resolución;

2. y porque la búsqueda de óptimos locales no tiene por qué conducir a un óptimo global.

Resumiendo, los algoritmos ávidos construyen la solución en etapas sucesivas, tratando siempre de tomar la decisión óptima para cada etapa. A la vista de todo esto no resulta difícil plantear un esquema general para este tipo de algoritmos:

---

**Algoritmo 21** Algoritmo Ávido
 

---

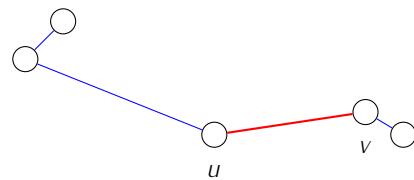
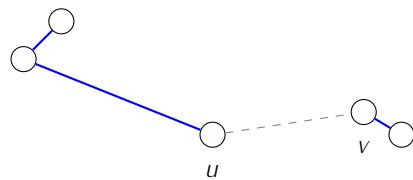
```

1: function GREEDY(C)                                ▷ C: Conjunto de entrada
2:   Conjunto = C
3:   Solucion = []
4:   Encontrado = False
5:   while not Esvacio(Conjunto) and not Encontrado do
6:     x = Seleccionar_mejor_candidato(Conjunto)
7:     Conjunto = [x]
8:     if EsFactible(Solucion + [x]) then
9:       Solucion = Solucion + [x]
10:      if EsSolucion(Solucion) then
11:        Encontrado = True
12:   return Solucion
  
```

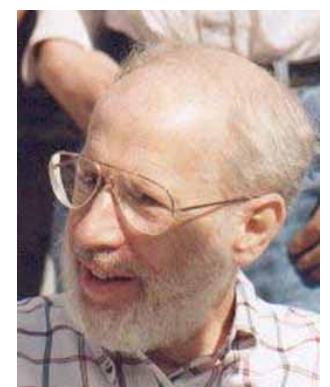
---

Nota: Un algoritmo de Union-and-Find realiza dos operaciones útiles en una estructura de datos de este tipo: FIND(*x*): determine en qué subconjunto se encuentra un elemento en particular. Esto se puede usar para determinar si dos elementos están en el mismo subconjunto. Encuentra al representante canónico de la componente conexa a la cual pertenece *x*.

UNION(*a,b*): unir dos subconjuntos en un solo subconjunto. Aquí primero tenemos que verificar si los dos subconjuntos pertenecen al mismo conjunto. Si no, entonces no podemos realizar la unión. Se fusionan las componentes canónicas representadas por *a* y *b*, respectivamente.



**Figura 11.** Otakar Borůvka

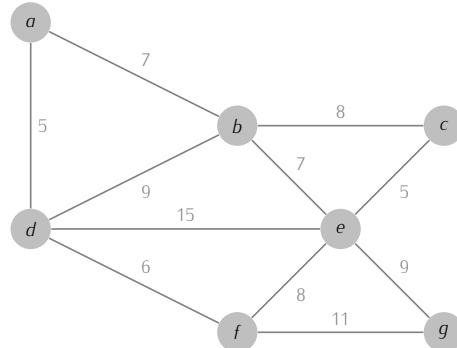


**Figura 12.** Joseph Kruskal

```

// A utility function to find the subset of an element i
int find(int parent[], int i)
{
    if (parent[i] == -1)
        return i;
    return find(parent, parent[i]);
}

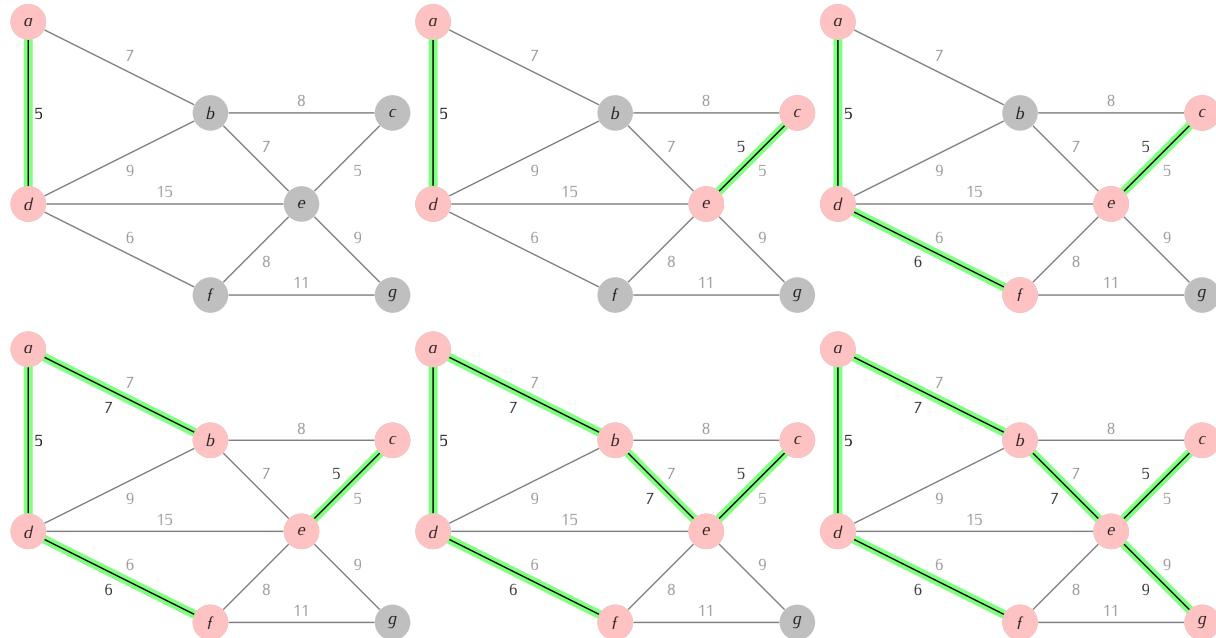
// A utility function to do union of two subsets
void Union(int parent[], int x, int y)
{
    parent[y] = x;
}
  
```

**Algoritmo 22 Árbol Cobertor Mínimo**

```

1: function KRUSKAL( $G(V, E)$ ,  $w$ )
2:    $SORT(E, w)$                                  $\triangleright$  Ordenar el conjunto de arcos
3:    $T = (V, \phi)$ 
4:    $C \leftarrow V$                              $\triangleright u, v \in V, C$  componentes conexas
5:   while  $|C| > 1$  do
6:      $e \leftarrow \{u, v\}$                        $\triangleright e$  es el siguiente arco en orden de costo creciente
7:     if  $FIND(u) \neq FIND(v)$  then
8:        $T \leftarrow T + e$ 
9:        $a \leftarrow FIND(u)$ 
10:       $b \leftarrow FIND(v)$ 
11:       $UNION(a, b)$ 
12:       $C \leftarrow C - 1$ 
return  $T$ 

```

Suma:  $5+5+6+7+7+9 = 39$ **10.1.1. Ejemplos de problemas No Aptos para Greedy**

- Problema de la Mochila (Knapsack Problem): Aunque una estrategia Greedy puede funcionar para la versión fraccionaria del problema, no es adecuada para la versión entera, donde los objetos no pueden ser divididos.
- Problema del Viajante de Comercio (TSP): Como se mencionó, elegir siempre la ciudad más cercana no garantiza el recorrido más corto.
- Problema de la Satisfacibilidad Booleana (SAT): No hay una elección local que garantice una solución globalmente satisfactoria.

SUBSECCIÓN 10.2

**Backtracking**

El backtracking (vuelta atrás) es una técnica algorítmica que busca soluciones a problemas de manera sistemática, probando diferentes opciones y retrocediendo cuando una opción no lleva a una solución válida. Es especialmente útil para problemas de búsqueda exhaustiva.

SUBSECCIÓN 10.3

## Características

---

1. **Búsqueda sistemática:** Explora todas las posibles soluciones de manera ordenada
2. **Retroceso:** Cuando una opción no es válida, vuelve atrás y prueba otra alternativa
3. **Poda:** Elimina ramas del árbol de búsqueda que no pueden llevar a una solución válida

Backtracking es una técnica útil para resolver el problema de las N-reinas. El algoritmo coloca una reina en una fila y luego intenta colocar la siguiente reina en una posición válida en la siguiente fila. Si se encuentra una posición donde no se puede colocar una reina sin ser atacada, el algoritmo retrocede y prueba una nueva posición para la reina anterior.

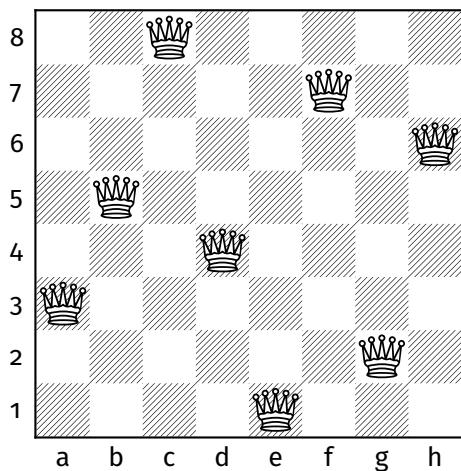
- Se puede entender como "opuesto" a avance rápido:
  - Avance rápido: añadir elementos a la solución y no deshacer ninguna decisión tomada.
  - Backtracking: añadir y quitar todos los elementos.
- Probar todas las combinaciones.

Este método en forma sistemática itera a través de todas las combinaciones posibles del espacio de búsqueda.

- Es una técnica general que debe ser adaptada para cada aplicación particular.
- Siempre puede encontrar todas las soluciones existentes
- El problema es el tiempo que toma en hacerlo.

La filosofía de estos algoritmos no sigue reglas fijas en la búsqueda de las soluciones. Podríamos hablar de un proceso de prueba y error en el cual se va trabajando por etapas construyendo gradualmente una solución. Para muchos problemas esta prueba en cada etapa crece de una manera exponencial, lo cual es necesario evitar.

Ejemplo 16



Solución representada por

[3, 6, 8, 2, 4, 1, 7, 5]

Colocar ocho reinas en un tablero de ajedrez sin que se den jaque. Dos reinas se dan jaque si comparten fila, columna o diagonal.

Fuerza bruta:

$$\binom{64}{8} = 4.426.165.368$$

---

**Algoritmo 23** *n*-Reinas
 

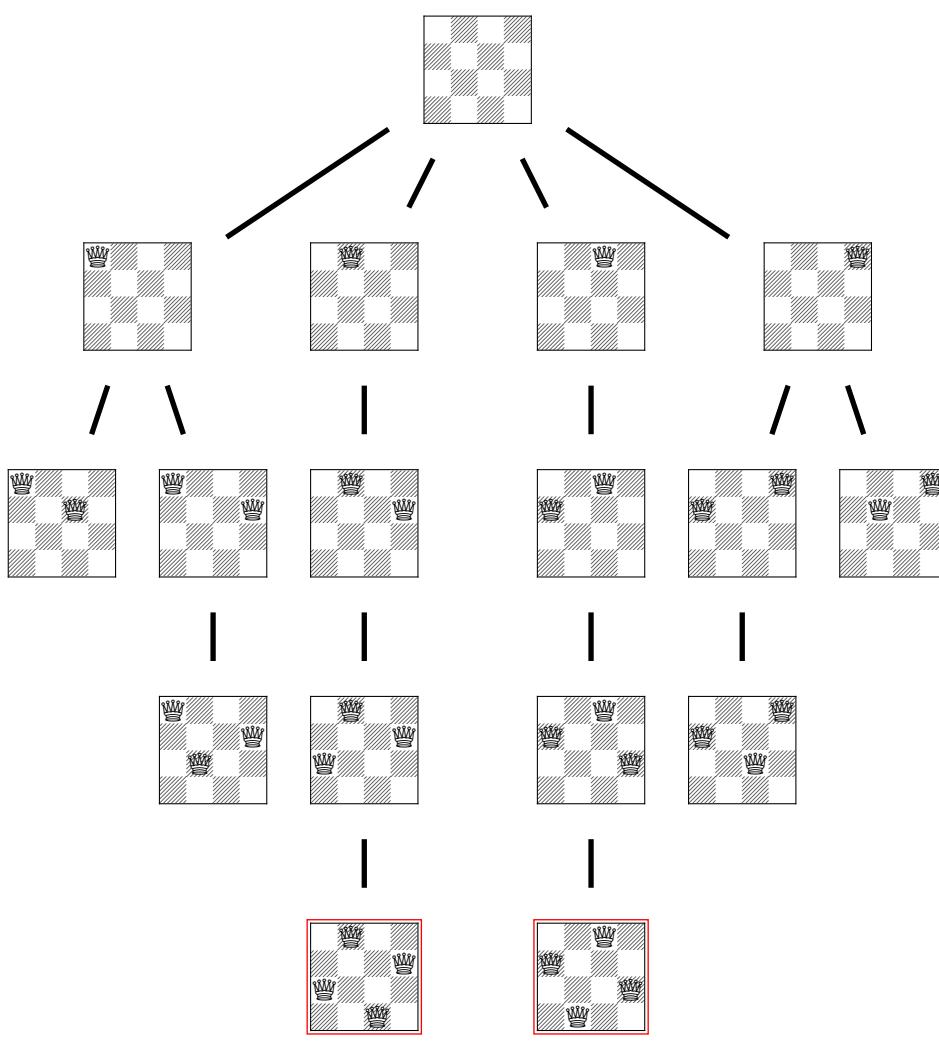
---

```

1: procedure PLACEQUEENS( $Q[1 \dots n]$ ,  $r$ )
2:   if  $r = n + 1$  then
3:     print  $Q[1 \dots n]$ 
4:   else
5:     for  $j \leftarrow 1$  to  $n$  do
6:        $legal \leftarrow True$ 
7:       if ( $Q[i] = j$ ) or ( $Q[i] = j + r - i$ ) or ( $Q[i] = j - r + i$ ) then
8:          $legal \leftarrow False$ 
9:       if  $legal$  then
10:         $Q[r] \leftarrow j$ 
11:        PLACEQUEENS( $Q[1 \dots n]$ ,  $r + 1$ )
  
```

---

Ejemplo 17



SUBSECCIÓN 10.4

## Cuándo usar backtracking

1. Cuando se necesita encontrar todas las soluciones posibles
2. Cuando el problema tiene restricciones que permiten poda efectiva
3. Cuando el espacio de búsqueda es finito pero grande
4. Cuando se requiere una solución óptima y no hay un algoritmo greedy aplicable

SECCIÓN 11

## Cuestiones y problemas

**11.1** Indique el comportamiento del algoritmo cuya ecuación de recurrencia es:

$$T(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ T\left(\frac{n}{2}\right) + 1 & \text{si } n > 1 \text{ y } n \text{ es potencia de 2} \\ T(n-1) + 1 & \text{otro caso} \end{cases}$$

**11.2** Considérese dos algoritmos  $A_1$  y  $A_2$  que resuelven el mismo problema y cuyos costes respectivos son  $1000n$  y  $n^2$  (en unidades de tiempo), donde  $n$  es el tamaño del problema. ¿ Para qué valores de  $n$  es preferible cada uno de los algoritmos mencionados?

**11.3** El algoritmo *búsqueda ternaria* es una variante de la búsqueda binaria, diseñada para que la zona del conjunto  $v$  en la que se busca el elemento  $x$ , se reduzca a la tercera parte en cada pasada del ciclo. Analice dicho algoritmo y comparelo con búsqueda binaria.

**11.4** Determine el algoritmo para multiplicar enteros grandes. Encontrar su coste. Este es el algoritmo de Karatsuba y Ofmann.

**11.5** Determine el algoritmo para multiplicar matrices en forma eficiente. Encontrar su coste. Este es el algoritmo de Strassen, también existe el algoritmo de Coppersmith y Winograd.

**11.6** Hacer un algoritmo para encontrar un subvector de longitud  $m$  cuya suma sea máxima dentro de un vector de longitud  $n$ , con  $m \leq n$ .

**11.7** Usando DyV, encontrar la suma de todos los numeros en un arreglo  $A[1 \dots n]$  de enteros. Indique el costo temporal. Justifique.

**11.8** Usando DyV, determinar el promedio de todos los numeros en un arreglo  $A[1 \dots n]$  de enteros, donde  $n$  es potencia de 2. Indique el costo temporal. Justifique.

**11.9** Sea  $A[1 \dots n]$  un arreglo de enteros y  $x$  un entero. Encuentre la frecuencia de  $x$  en  $A$ , esto es, el número de veces que aparece  $x$  en  $A$ .

**11.10** Escriba un algoritmo que encuentre el  $k$ -ésimo menor elemento de un arreglo  $A[1 \dots n]$  de numeros enteros no repetidos y no ordenado.

**11.11** Escriba un algoritmo para determinar si en un arreglo  $A[1 \dots n]$  existe el elemento  $A[i] = i$ , para  $1 \leq i \leq n$ . El arreglo esta ordenado y puede tener enteros negativos.

**11.12** Dada una matriz  $C[m, n]$ , que es solución para el problema de cambio de monedas, determinar un algoritmo que permita saber cuales son las monedas que representa la solución.

**11.13** Dada una matriz  $ED[m, n]$ , que representa la solución del problema distancia de edición, determinar un algoritmo que permita saber cuales son las operaciones que representa la solución.

**11.14** Dado un grafo  $G = (V, E)$  dirigido y acíclico con  $n$  vértices. Sean  $s$  y  $t$  dos vértices en  $V$  tal que el grado de entrada(*indegree*) de  $s$  es 0 y el *outdegree* de  $t$  es 0. Usando DP implemente un algoritmo para computar el camino mas largo en  $G$  desde  $s$  a  $t$ . Indique la complejidad de su algoritmo.

# Ordenación

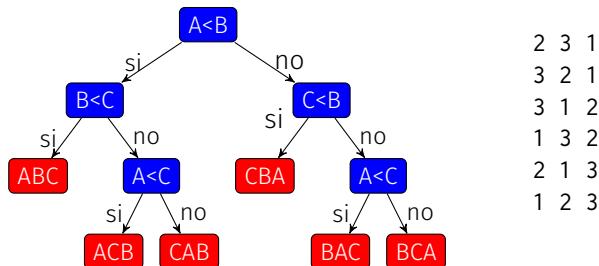
## Cota inferior

Supongamos que deseamos ordenar tres datos A, B y C. A continuación se muestra un árbol de decisión posible para resolver este problema. Los nodos internos del árbol representan comparaciones y los nodos externos representan salidas emitidas por el programa.

Se desea obtener un conjunto con sus elementos ordenados según  $\leq$ , por ejemplo. La idea es encontrar una permutación que cumpla con esa regla.

Se pueden generar todas las permutaciones y elegir aquella que cumpla con el orden total.

De todos los algoritmos posibles, nos centraremos en los que emplean como operación básica la comparación entre elementos (algoritmos de ordenación por comparación).



Todo árbol de decisión con  $H$  hojas tiene al menos altura  $\log_2 H$ , y la altura del árbol de decisión es igual al número de comparaciones que se efectúan en el peor caso.

En un árbol de decisión para ordenar  $n$  datos se tiene que  $H = n!$ , y por lo tanto se tiene que todo algoritmo que ordene  $n$  datos mediante comparaciones entre llaves debe hacer al menos  $\log_2 n!$  comparaciones en el peor caso.

Usando la aproximación de Stirling, se puede demostrar que  $\log_2 n! = n \log_2 n + \mathcal{O}(n)$ , por lo cual la cota inferior es de  $\mathcal{O}(n \log n)$ .

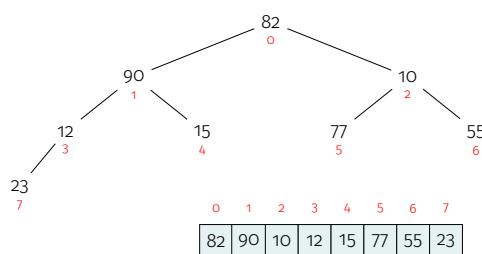
Si suponemos que todas las posibles permutaciones resultantes son equiprobables, es posible demostrar que el número promedio de comparaciones que cualquier algoritmo debe hacer es también de  $\mathcal{O}(n \log n)$ .

HEAPSORT es un método de ordenación que utiliza un heap(cola de prioridad) como estructura de datos. Un heap tiene como representación un árbol binario "semiordenado", pero en la práctica es un arreglo. Cada nodo del árbol corresponde a un elemento del arreglo. Si bien pareciera que el árbol puede construir balanceado y completo, usualmente el subárbol derecho puede carecer de hojas en el mismo nivel que el subárbol izquierdo.

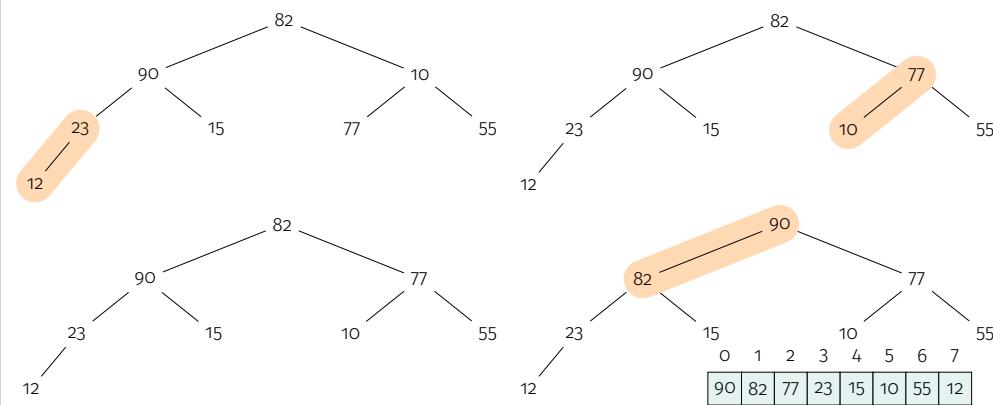


**Figura 13.** John William Joseph Williams

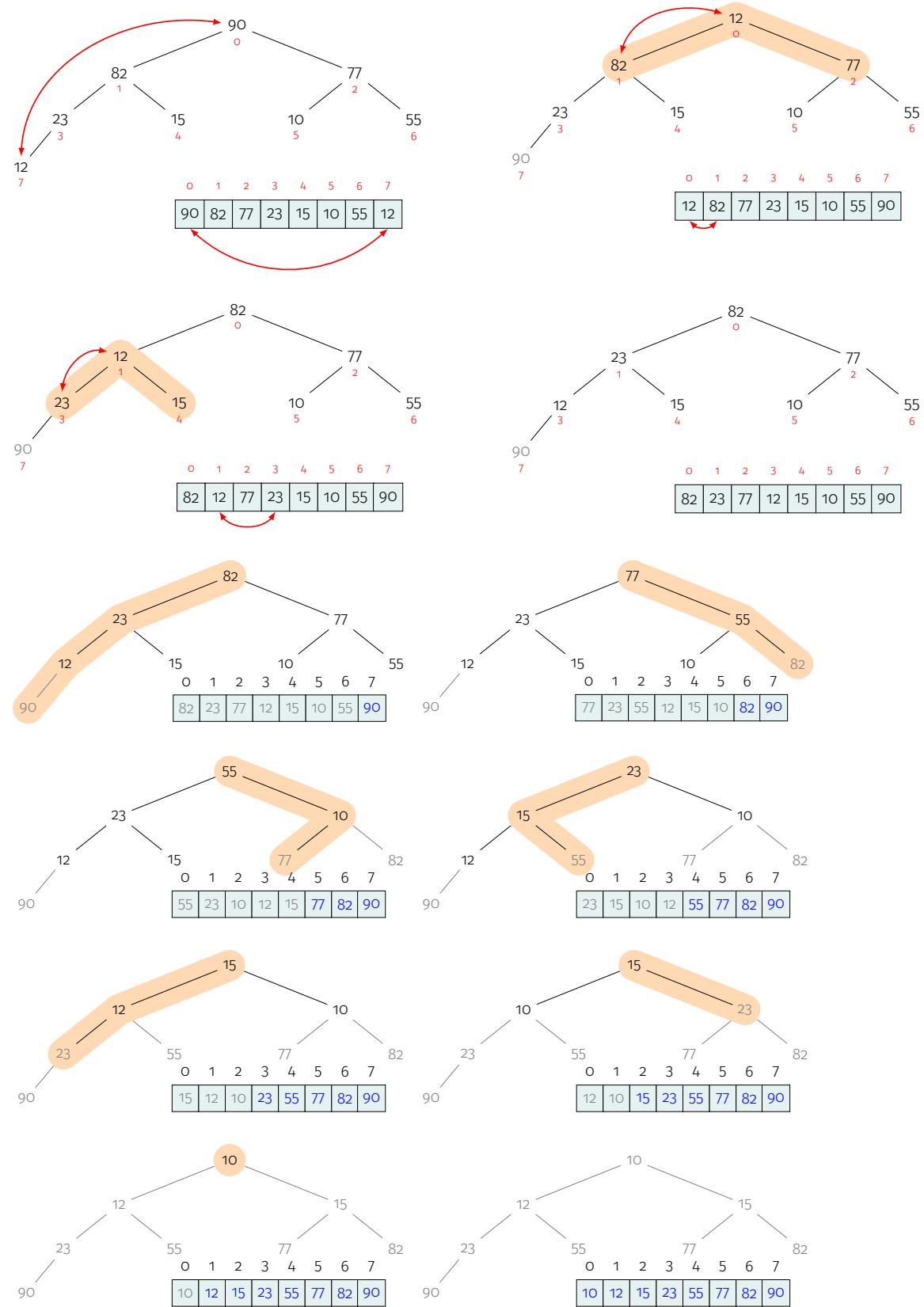
Ejemplo 18 | Se desea ordenar el conjunto: 82, 90, 10, 12, 15, 77, 55, 23.



## Construir el HEAP



## HEAPSORT



**Algoritmo 24** Max Heapify

```

1: procedure MAX-HEAPIFY( $A, i$ ) ▷ A arreglo
2:    $l \leftarrow \text{LEFT}(i)$ 
3:    $r \leftarrow \text{RIGHT}(i)$ 
4:   if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$  then
5:      $\text{largest} \leftarrow l$ 
6:   else
7:      $\text{largest} \leftarrow i$ 
8:   if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$  then
9:      $\text{largest} \leftarrow r$ 
10:    if  $\text{largest} \neq i$  then
11:      exchange  $A[i]$  with  $A[\text{largest}]$ 
12:      MAX-HEAPIFY( $A, \text{largest}$ )

```

**Algoritmo 25** HeapSort

```

1: procedure HEAPSORT( $A$ ) ▷ A arreglo
2:   BUILD-MAX-HEAP( $A$ )
3:   for  $i \leftarrow A.\text{length}$  downto 2 do
4:     exchange  $A[1]$  with  $A[i]$ 
5:      $A.\text{heap-size} \leftarrow A.\text{heap-size} - 1$ 
6:   MAX-HEAPIFY( $A, 1$ )

```

**Algoritmo 26** Construir Heap de Máximos

```

1: procedure BUILD-MAX-HEAP( $A$ ) ▷ A arreglo
2:    $A.\text{heap-size} \leftarrow A.\text{length}$ 
3:   for  $i \leftarrow \lfloor A.\text{length}/2 \rfloor$  downto 1 do
4:     MAX-HEAPIFY( $A, 1$ )

```

Para construir inicialmente el heap (BUILD-MAX-HEAP()), se llama a la función MAX-HEAPIFY() para cada nodo principal: hacia atrás, comenzando con el último nodo y terminando en la raíz del árbol.

Un heap de tamaño  $n$  tiene  $n/2$  nodos principales (redondeados hacia abajo):

Dado que la complejidad de la función MAX-HEAPIFY() es  $O(\log n)$ , la complejidad de BUILD-MAX-HEAP() es, por lo tanto  $\mathcal{O}(n \log n)$ .

Nota: en realidad la complejidad temporal de BUILD-MAX-HEAP() es  $\mathcal{O}(n)$ . Dado que esto no cambia la complejidad temporal general, no es obligatorio realizar este análisis en profundidad.

Como se aprecia la función BUILD-MAX-HEAP() llama a MAX-HEAPIFY() para cada nodo padre. Lo que no hemos considerado hasta ahora es que la profundidad de los subárboles, en los que se llama a MAX-HEAPIFY(), varía.

La función MAX-HEAPIFY() se llama como máximo para  $n/4$  árboles de profundidad 1, para  $n/8$  árboles de profundidad 2, para  $n/16$  árboles de profundidad 3, etc. El número máximo de operaciones de intercambio en la función MAX-HEAPIFY() es igual a la profundidad del subárbol en el que se llama. El máximo número de intercambios es:

$$S_{\max} = \frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \frac{n}{16} \cdot 3 + \frac{n}{32} \cdot 4 + \dots$$

Que finalmente demuestra que la construcción de un heap es lineal. La complejidad es  $\mathcal{O}(n)$ .

Sin embargo, la complejidad total es  $\mathcal{O}(n \log n)$ .

SECCIÓN 13

## MergeSort(fusión)

Este algoritmo funciona por *equilibrado binario*. Descompone el vector en dos subvectores de tamaños similares, los ordena recursivamente y combina los resultados en un vector ordenado.



Input: S O R T I N G E X A M P L  
 Divide: S O R T I N | G E X A M P L  
 Recursividad Izq: I N O R S T | G E X A M P L  
 Recursividad Der: I N O R S T | A E G L M P X  
 Merge: A E G I L M N O P R S T X

**Algoritmo 27** Merge

```

1: procedure MERGE( $A, p, q, r$ ) ▷  $A$  arreglo
2:    $n_1 \leftarrow q - p + 1$ 
3:    $n_2 \leftarrow r - q$ 
4:   Sean  $L[1..n_1 + 1]$  y  $R[1..n_2 + 1]$  nuevos arreglos
5:   for  $i \leftarrow 1, n_1$  do
6:      $L[i] \leftarrow A[p + i - 1]$ 
7:   for  $j \leftarrow 1, n_2$  do
8:      $R[j] \leftarrow A[q + j]$ 
9:    $L[n_1 + 1] \leftarrow \infty$ 
10:   $R[n_2 + 1] \leftarrow \infty$ 
11:   $i \leftarrow 1$ 
12:   $j \leftarrow 1$ 
13:  for  $k \leftarrow p, r$  do
14:    if  $L[i] \leq R[j]$  then
15:       $A[k] \leftarrow L[i]$ 
16:       $i \leftarrow i + 1$ 
17:    else
18:       $A[k] \leftarrow R[j]$ 
19:       $j \leftarrow j + 1$ 
```

**Algoritmo 28** Ordenación por mezcla

```

1: procedure MERGESORT( $A, low, high$ )
2:   if  $low < high$  then
3:      $mid \leftarrow \lfloor (low + high)/2 \rfloor$ 
4:     MERGESORT( $A, low, mid$ )
5:     MERGESORT( $A, mid + 1, high$ )
6:     MERGE( $A, low, mid, high$ )
```

Además al observar el algoritmo recursivo de MERGESORT, podemos ver que se hacen 2 llamadas recursivas a la misma función. Entonces el tiempo debería ser:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ \left\lfloor T\left(\frac{n}{2}\right)\right\rfloor + \left\lceil T\left(\frac{n}{2}\right)\right\rceil + bn & \text{si } n > 1 \end{cases}$$

Para hacer el análisis más sencillo podemos decir que si  $n$  es potencia de 2, tenemos:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T\left(\frac{n}{2}\right) + bn & \text{si } n > 1 \end{cases}$$

Si tuvieramos el caso general:

$$f(n) = \begin{cases} d & \text{si } n = 1 \\ af\left(\frac{n}{c}\right) + bn^x & \text{si } n \geq 1 \end{cases}$$

cuya solución es:

$$f(n) = \begin{cases} \Theta(n^x) & \text{si } a < c^x \\ \Theta(n^x \log n) & \text{si } a = c^x \\ \Theta(n^{\log_c a}) & \text{si } a > c^x \end{cases}$$

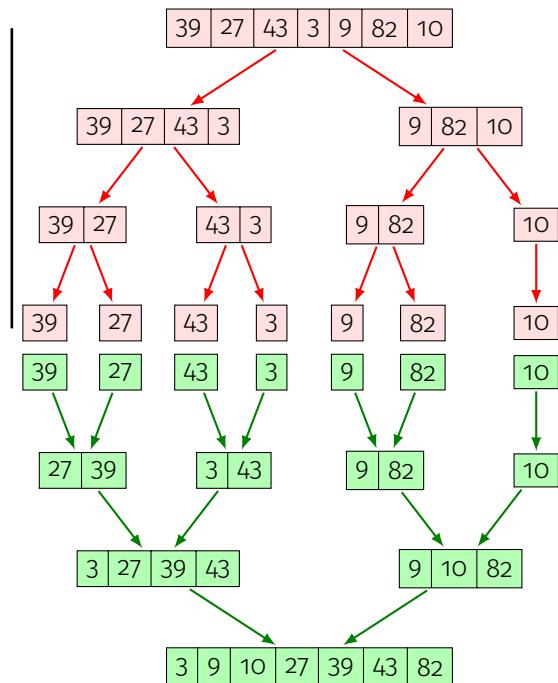
Finalmente, haciendo:  $x = 1$ ,  $a = 2$ ,  $c = 2$ , tenemos el caso  $a = c^x$ , cuya solución es:  $\Theta(n \log n)$ .

En el caso de que la división de la lista no fuera equilibrada, se tendría:

$$T(n) = \begin{cases} c_1 & \text{si } n \leq 1 \\ T(n - 1) + c_2 n + c_3 & \text{si } n > 1 \end{cases}$$

en cuyo caso la solución es:  $\Theta(n^2)$ .

Ejemplo 19

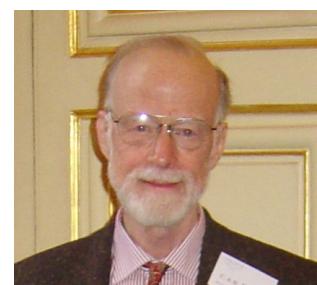
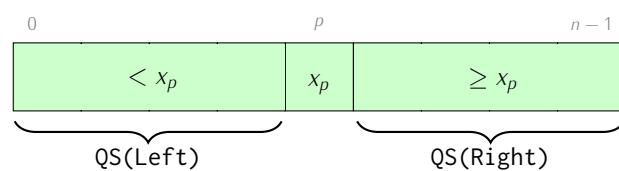


SECCIÓN 14

## Quicksort

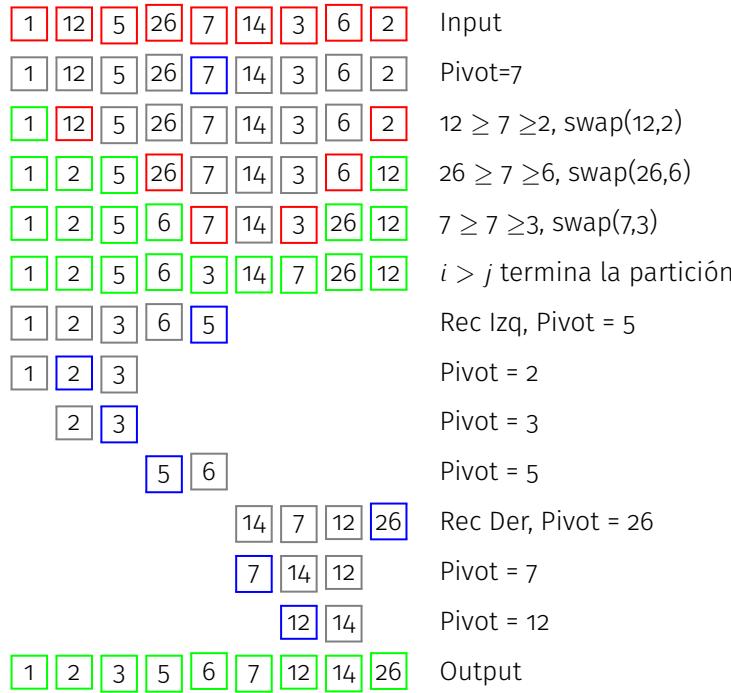
QUICKSORT es otro algoritmo recursivo de ordenación. El funcionamiento ocurre al dividir un arreglo en subarreglos más pequeños *antes* de la recursión, de forma que la mezcla de los subarreglos ordenados sea trivial. Esto es:

1. Elegir un elemento *pivote* del arreglo.
2. Particionar el arreglo en tres subarreglos, conteniendo los elementos más pequeños que el pivote, el pivote en sí y los elementos mayores.
3. Realizar QUICKSORT en el primer y último subarreglos.



**Figura 17.** Charles Antony Richard Hoare

Input: S O R T I N G E X A M P L  
 Pivot: S O R T I N G E X A M P L  
 Partición: A G O E I N L M P T X S R  
 Recursividad Izq: A E G I L M N O P T X S R  
 Recursividad Der: A E G I L M N O P R S T X

**Algoritmo 29** Ordenación rápida

```

1: procedure QUICKSORT( $A$ ) ▷  $A[1 \dots n]$ 
2:   if  $n > 1$  then
3:     Elegir como pivote al elemento  $A[p]$ 
4:      $r \leftarrow \text{PARTITION}(A, p)$ 
5:     QUICKSORT( $A[1 \dots r - 1]$ )
6:     QUICKSORT( $A[r + 1 \dots n]$ )

```

**Algoritmo 30** Partición Lomuto

```

1: function LOMUTO-PARTITION( $A, p$ ) ▷  $A[1 \dots n]$ 
2:    $A[p] \leftrightarrow A[n]$ 
3:    $l \leftarrow 0$ 
4:   for  $i \leftarrow 1$  to  $n - 1$  do
5:     if  $A[i] < A[n]$  then
6:        $l \leftarrow l + 1$ 
7:        $A[l] \leftrightarrow A[i]$ 
8:    $A[n] \leftrightarrow A[l + 1]$ 
9:   return  $l + 1$ 

```

**Algoritmo 31** Partición Hoare

```

1: function HOARE-PARTITION( $A, p, r$ ) ▷  $A[1 \dots n]$ 
2:    $x \leftarrow A[p]$ 
3:    $i \leftarrow p - 1$ 
4:    $j \leftarrow r + 1$ 
5:   while  $\text{TRUE}$  do
6:     repeat
7:        $j \leftarrow j - 1$ 
8:     until  $A[j] \leq x$ 
9:     repeat
10:     $i \leftarrow i + 1$ 
11:   until  $A[i] \geq x$ 
12:   if  $i < j$  then
13:     SWAP( $A[i], A[j]$ )
14:   else
15:     return  $j$ 

```

PARTITION claramente es de orden  $O(n)$ . Para Quicksort depende de  $r$ :

$$T(n) = T(r-1) + T(n-r) + O(n)$$

Supongamos que magicamente siempre elegimos el pivote como la mediana del arreglo  $A$ , podemos tener  $r = \lceil n/2 \rceil$  (dos subproblemas de aproximadamente mismo tamaño).

$$T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n)$$

O

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

Donde obtenemos que  $T(n) \in \mathcal{O}(n \log n)$

En el peor caso:

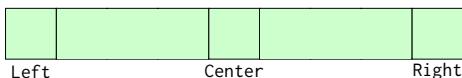
$$T(n) = T(n-1) + cn$$

Donde obtenemos que  $T(n) \in \mathcal{O}(n^2)$

Una parte fundamental es la PARTITION. Existen varias formas, como lo hemos mencionado, cabe destacar otra muy usada en la práctica: mediana de 3.

```

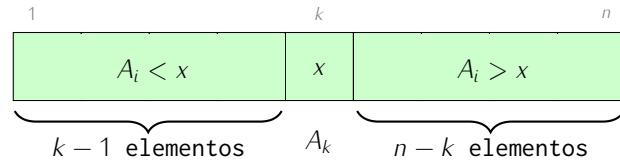
1  /* Return median of Left, Center, and Right */
2  /* Order these and hide the pivot */
3
4  int Median3( int A[ ], int Left, int Right )
5  {
6      int Center = ( Left + Right ) / 2;
7
8      if( A[ Left ] > A[ Center ] )
9          swap( &A[ Left ], &A[ Center ] );
10     if( A[ Left ] > A[ Right ] )
11         swap( &A[ Left ], &A[ Right ] );
12     if( A[ Center ] > A[ Right ] )
13         swap( &A[ Center ], &A[ Right ] );
14
15     /* Invariant: A[ Left ] <= A[ Center ] <= A[ Right ] */
16
17     swap( &A[ Center ], &A[ Right - 1 ] ); /* Hide pivot */
18     return A[ Right - 1 ];                  /* Return pivot */
19 }
```



SECCIÓN 15

## Quickselect

Es un algoritmo para encontrar el  $k$ -ésimo elemento de un arreglo de tamaño  $n$ , ( $1 \leq k \leq n$ ). En honor a su creador también se conoce como *Hoare's Selection Algorithm*.

**Algoritmo 32** QuickSelect

```

1: function QUICKSELECT( $A, k$ )  $\triangleright A[1 \dots n]$ 
2:   if  $n = 1$  then
3:     return  $A[1]$ 
4:   else
5:     Elegir un elemento pivote  $A[p]$ 
6:      $r \leftarrow \text{PARTITION}(A, p)$ 
7:     if  $k < r$  then
8:       return QUICKSELECT( $A[1 \dots r - 1], k$ )
9:     else if  $k > r$  then
10:      return QUICKSELECT( $A[r + 1 \dots n], k - r$ )
11:    else
12:      return  $A[r]$ 

```

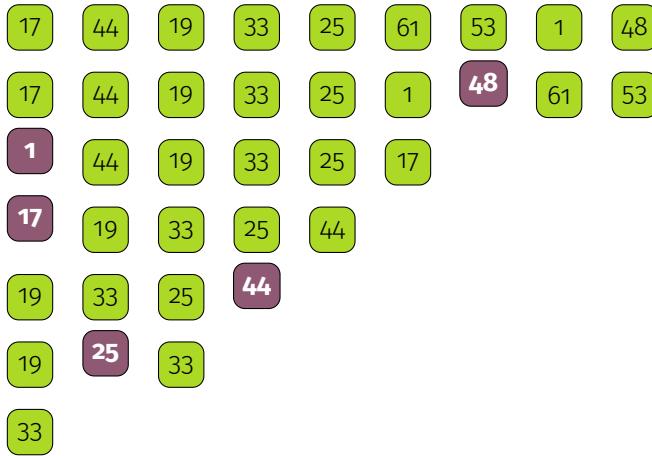
```

void Qselect( int A[ ], int k, int Left, int Right )
{
  if(Left == Right)
    return A[Left];
  int pos = partition(A, Left ,Right);
  int i = pos - Left + 1;

  if( i == k)
    return A[pos];
  else if(i>k)
    return Qselect(A,k, Left ,pos-1);
  else
    return Qselect(A,k-i, pos+1,Right);
}

```

Ejemplo 20 Encontrar el 5to menor elemento ( $k = 5$ ) del conjunto:  $\{17, 44, 19, 33, 25, 61, 53, 1, 48\}$ .



La partición puede ser Hoare o Lomuto, en ambos casos se tiene el mismo análisis.

La partición de Lomuto de un arreglo de  $n$  elementos realiza  $n + 1$  comparaciones y, como máximo  $n$  intercambios. Entonces, su complejidad es  $\Theta(n)$ . Denotemos como  $T_n$  el número de pasos (comparaciones e intercambios) que realiza QUICKSELECT al buscar en un arreglo de  $n$  elementos. Dado que QUICKSELECT siempre reduce el tamaño del subarreglo al elegir uno (porque el subarreglo ignorado está vacío o el pivote no está

incluido en el subarreglo), se mantiene la siguiente recurrencia en el peor caso:

$$\begin{aligned} T_n &= \Theta(n) + T_{n-1} \\ T_n &= \Theta(n) + \Theta(n-1) + \dots + \Theta(1) \\ &= \Theta(n + (n-1) + \dots + 1) \\ &= \Theta\left(\frac{n(n+1)}{2}\right) \\ &= \Theta(n^2) \end{aligned}$$

El mejor de los casos ocurre cuando QUICKSELECT elige el  $k$ -ésimo elemento como pivote en la primera llamada. Luego, el algoritmo realiza pasos de  $\Theta(n)$  durante la primera (y única) partición, después de lo cual termina. Por lo tanto, QUICKSELECT es  $\Theta(n)$  en el mejor caso.

En el caso promedio es  $\mathcal{O}(n)$ . Análisis caso promedio.

#### SUBSECCIÓN 15.1

## Resumen ordenación/búsqueda

---

Algoritmo	Mejor caso	Peor caso	Caso promedio	Espacio
BUBBLESORT	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
SELECTIONSORT	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
INSERTIONSORT	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
HEAPSORT	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$
MERGESORT	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
QUICKSORT	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$
SECUENCIALSEARCH	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
BINARYSEARCH	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
QUICKSELECT	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
BOYER-MOORE	$\mathcal{O}(n/m)$	$\mathcal{O}(mn)$	$\mathcal{O}(n)$	$\mathcal{O}(s)$
KMP	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(m)$
RABIN-KARP	$\mathcal{O}(n)$	$\mathcal{O}(mn)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
AHO-CORASICK	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\sum  p_i )$

#### SECCIÓN 16

## Cuestiones y problemas

---

**16.1** Dé un ejemplo de los mejores y peores casos de cada algoritmo de ordenación.

**16.2** Dé un ejemplo y analice casos desequilibrados de:

- MergeSort
- QuickSort

**16.3** Implemente búsqueda binaria en 2D.

**16.4** Para determinar si un ítem  $x$  se encuentra presente en una matriz cuadrada de ítems  $T(1..n, 1..n)$  cuyas filas y columnas están ordenadas crecientemente (de

izquierda a derecha y de arriba a abajo respectivamente), se utiliza el siguiente algoritmo: (Sea benevolente con el enunciado, admitiendo que siempre tenga sentido eso de "la diagonal"):

- Con búsqueda dicotómica (búsqueda binaria) se busca  $x$  en la "diagonal".
- Si encontramos  $x$ , terminamos.
- Si no, (encontramos  $p$  y  $q$  tales que  $p < x < q$ ) descartamos los dos trozos de la matriz donde no puede estar  $x$  e invocamos recursivamente el mismo procedimiento sobre los dos trozos restantes.

Implemente el algoritmo e indique de qué orden es en el peor caso.

**16.5** Cree un algoritmo que dado un arreglo lo transforme en *heap*. Indique cual sería el orden de dicho algoritmo.

**16.6** Un mínimo local de un arreglo  $A(a - 1 \dots b + 1)$  es un elemento  $A(k)$  que satisface  $A(k - 1) \geq A(k) \leq A(k + 1)$ . Suponemos que  $a \leq b$  y  $A(a - 1) \geq A(a)$  y  $A(b) \leq A(b + 1)$ ; estas condiciones garantizan la existencia de algún mínimo local. Diseña un algoritmo que encuentre algún mínimo local de  $A(a - 1 \dots b + 1)$  y que sea sustancialmente más rápido que el evidente de  $O(n)$  en el peor caso. ¿De qué orden es su algoritmo?

**16.7** Sea  $F(x)$  una función monótona decreciente y sea  $N$  el mayor entero que cumple  $F(N) \geq 0$ . Asumiendo que  $N$  existe, un algoritmo para determinar dicho  $N$  es el siguiente:

```
i=0;
while (f(i) > 0 )
    i=i+1;
N = i-1;
```

No obstante, esta solución tiene complejidad  $O(N)$ . Escríbase un algoritmo cuyo comportamiento asintótico sea mejor en función de  $N$ .

**16.8** Implemente una función que permita encontrar la variación con repetición de los caracteres de una cadena. Ej: In:AB, Out: AA, AB, BA, BB. Calcule la complejidad de su implementación.

**16.9** Implemente un algoritmo que ordene una lista de números enteros positivos y negativos de manera que los números positivos queden ordenados de menor a mayor y los negativos de mayor a menor. Ej: In: [5, -3, 2, -1, 4, -2], Out: [2, 4, 5, -3, -2, -1].

**16.10** Diseñe un algoritmo que ordene una lista de fechas en formato DD/MM/AAAA. Considere que las fechas pueden estar en diferentes formatos y que debe manejar años bisiestos.

**16.11** Implemente un algoritmo que ordene una lista de palabras considerando el orden alfabético inverso (de Z a A) y que ignore mayúsculas/minúsculas.

**16.12** Diseñe un algoritmo que ordene una lista de números decimales considerando solo la parte entera, pero manteniendo la parte decimal en su posición original. Ej: In: [3.2, 1.5, 2.8, 4.1], Out: [1.5, 2.8, 3.2, 4.1].

**16.13** Implemente un algoritmo que ordene una lista de direcciones IP (formato xxx.xxx.xxx.xxx) considerando cada octeto como un número independiente.

- 16.14** Diseñe un algoritmo que ordene una lista de versiones de software (formato X.Y.Z) considerando cada número como un valor independiente. Ej: In: [1.2.3, 1.1.0, 2.0.0], Out: [1.1.0, 1.2.3, 2.0.0].
- 16.15** Implemente un algoritmo que ordene una lista de nombres completos considerando primero el apellido y luego el nombre. Ej: In: ["Juan Pérez", "Ana García", "Carlos López"], Out: ["Ana García", "Carlos López", "Juan Pérez"].
- 16.16** Diseñe un algoritmo que ordene una lista de números romanos (I, II, III, IV, V, etc.) considerando su valor numérico correspondiente.
- 16.17** Implemente un algoritmo que ordene una lista de números binarios (representados como strings de os y 1s) considerando su valor decimal correspondiente sin convertirlos explícitamente a decimal.
- 16.18** Diseñe un algoritmo que ordene una lista de números hexadecimales (representados como strings) considerando su valor decimal correspondiente sin convertirlos explícitamente a decimal.
- 16.19** Implemente un algoritmo que ordene una lista de números en notación científica (ej: 1.23e-4) considerando su valor numérico correspondiente.
- 16.20** Diseñe un algoritmo que ordene una lista de números complejos considerando primero la parte real y luego la parte imaginaria.
- 16.21** Implemente un algoritmo que ordene una lista de matrices 2x2 considerando el valor de su determinante.
- 16.22** Diseñe un algoritmo que ordene una lista de polinomios (representados como arrays de coeficientes) considerando su grado y luego los coeficientes en orden descendente.

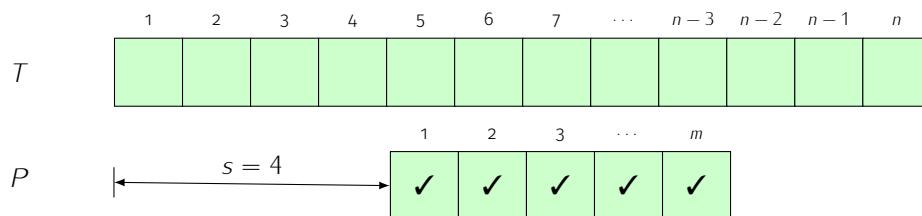
# Búsqueda exacta en texto (Exact string matching)

Encontrar todas las ocurrencias de un patrón en un texto. Luego se tiene un texto  $T[1..n]$  contiene  $n$  caracteres de un alfabeto finito, y un patrón  $P[1..m]$  de tamaño  $m$ . Ejemplo de alfabeto  $\Sigma = \{0, 1\}$ ,  $\Sigma = \{a, b, c, \dots, z\}$ .

$T$  y  $P$  son cadenas de caracteres del mismo alfabeto.

Se dice que el patrón  $P$  existe para algún desplazamiento  $s$  en el texto  $T$ , si:

$$\begin{aligned} T[s + 1 \dots s + m] &= P[1 \dots m], \text{ para } 0 \leq s \leq n - m \text{ o} \\ T[s + j] &= P[j], \text{ para } 1 \leq j \leq m \end{aligned}$$



Luego el problema de *string-matching* consiste en encontrar todos los desplazamientos  $s$  válidos de un patrón  $P$  sobre un texto  $T$ .

SECCIÓN 17

## Búsqueda secuencial

También se llama Náive o fuerza bruta.

### Algoritmo 33 Fuerza Bruta

---

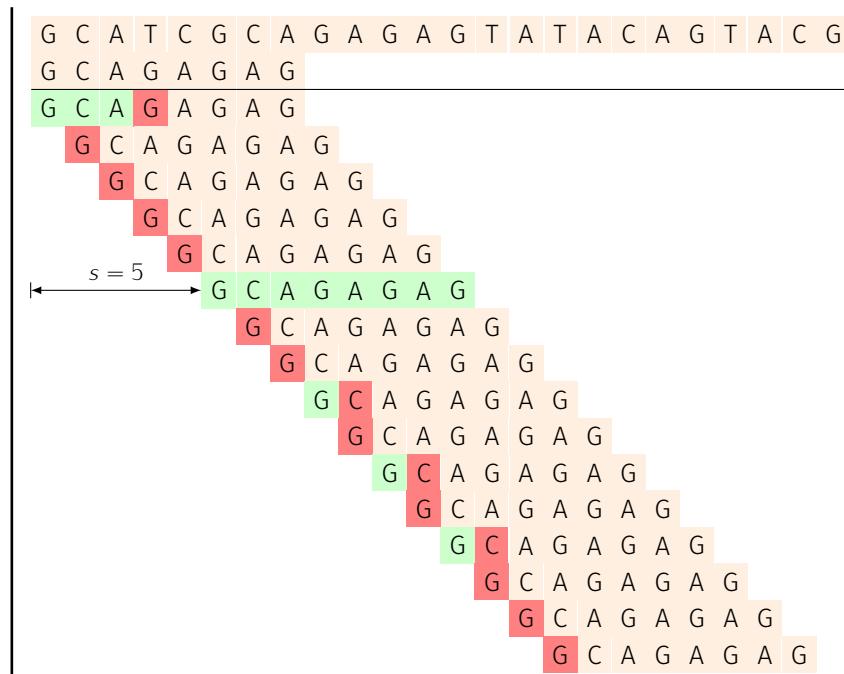
```

1: function BRUTEFORCESTRINGMATCH( $T, P$ ) ▷  $T[0 \dots n - 1], P[0 \dots m - 1]$ 
2:   for  $i \leftarrow 0$  to  $n - m$  do
3:      $j \leftarrow 0$ 
4:     while  $j < m$  and  $P[j] = T[i + j]$  do  $j \leftarrow j + 1$ ;
5:     if  $j = m$  then
6:       return  $i$ 
7:   return  $-1$ 

```

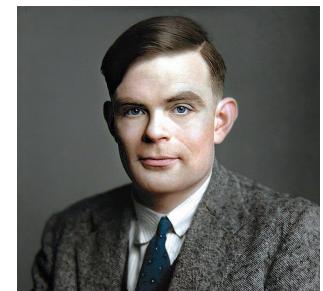
---

Ejemplo 21



SECCIÓN 18

## Autómata



**Figura 18.** Alan Turing

El autómata finito determinista (DFA) es una máquina de estados finitos que puede ser utilizada para buscar patrones en texto de manera eficiente. Un DFA consiste en:

- Un conjunto finito de estados  $Q$
- Un alfabeto finito  $\Sigma$
- Una función de transición  $\delta : Q \times \Sigma \rightarrow Q$
- Un estado inicial  $q_0 \in Q$
- Un conjunto de estados finales  $F \subseteq Q$

Para construir un DFA que busque un patrón  $P[1..m]$ , seguimos estos pasos:

1. Creamos  $m + 1$  estados numerados de 0 a  $m$
2. El estado 0 es el estado inicial
3. El estado  $m$  es el único estado final
4. Para cada estado  $i$  y cada carácter  $c \in \Sigma$ :
  - Si  $c = P[i + 1]$ , la transición va al estado  $i + 1$
  - Si no, la transición va al estado más alto  $j < i$  tal que  $P[1..j]$  es un sufijo de  $P[1..i]c$

Ejemplo 22 Consideremos el patrón  $P = GCAGAGAG$  sobre el alfabeto  $\Sigma = \{A, C, G, T\}$ . La tabla de transiciones se construye como sigue:

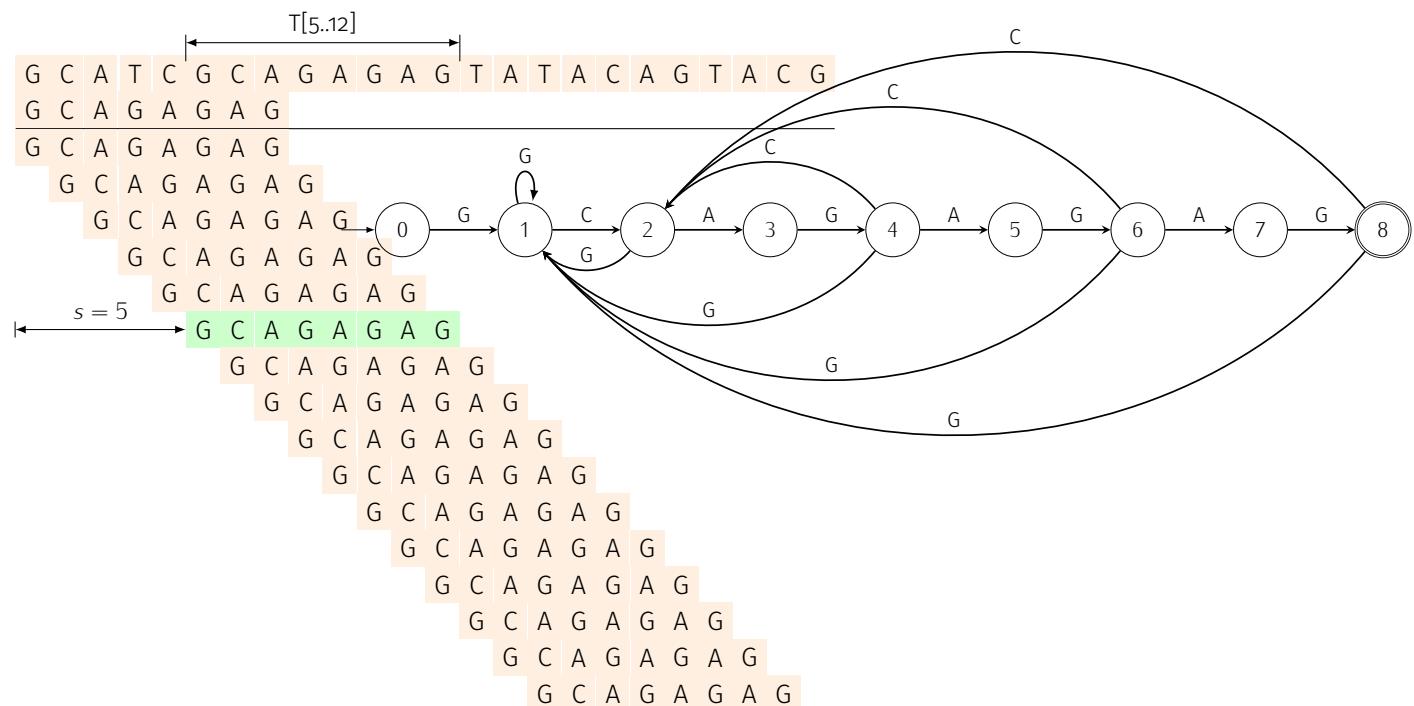
$$\begin{aligned}\Sigma &= \{A, C, G, T\} \\ Q &= \{0, 1, 2, 3, 4, 5, 6, 7, 8\} \\ q_0 &= 0 \\ F &= \{8\}\end{aligned}$$

$\delta$	A	C	G	T
0	0	0	1	0
1	0	2	1	0
2	3	0	1	0
3	0	0	4	0
4	5	2	1	0
5	0	0	6	0
6	7	2	1	0
7	0	0	8	0
8	0	2	1	0

La construcción de la tabla se explica así:

- Estado 0: Es el estado inicial. Solo avanza con 'G'
- Estado 1: Avanza con 'C', retrocede a 0 con 'A' o 'T', se mantiene con 'G'
- Estado 2: Avanza con 'A', retrocede a 0 con 'C' o 'T', retrocede a 1 con 'G'
- Y así sucesivamente...

Búsqueda del texto: **GCAGAGAG**



Las ventajas de usar un DFA para búsqueda de patrones son:

- Procesa cada carácter del texto exactamente una vez
- No necesita retroceder en el texto
- La complejidad temporal es  $\Theta(n)$  donde  $n$  es la longitud del texto

- | ■ La complejidad espacial es  $\Theta(m|\Sigma|)$  donde  $m$  es la longitud del patrón

SECCIÓN 19

## Karp-Rabin

Al igual que el algoritmo secuencial, el algoritmo de Rabin-Karp también desliza el patrón uno a uno pero utilizado para buscar/coincidir patrones en el texto mediante una función hash. Pero a diferencia del algoritmo secuencial, no viaja a través de todos los caracteres en la fase inicial, sino que filtra los caracteres que no coinciden comparando el valor hash del patrón con el valor hash del texto. Si coinciden entonces, se realiza la coincidencia de caracteres.

### Algoritmo 34 Karp Rabin

```

1: function KARPRABINSTRINGMATCH( $T, P, d, q$ ) ▷  $T[0 \dots n - 1], P[0 \dots m - 1]$ 
2:    $h \leftarrow d^{m-1} \bmod q$ 
3:    $p \leftarrow 0$ 
4:    $t_0 \leftarrow 0$ 
5:   for  $i \leftarrow 1$  to  $m$  do ▷ Preprocessing
6:      $p \leftarrow (dp + P[i]) \bmod q$ 
7:      $t_0 \leftarrow (dt_0 + T[i]) \bmod q$ 
8:   for  $s \leftarrow 0$  to  $n - m$  do ▷ Searching
9:     if  $p = t_s$  then
10:      if  $P[1 \dots m] = T[s + 1 \dots s + m]$  then
11:        print "Pattern occurs with shift"  $s$ 
12:      if  $s < n - m$  then
13:         $t_{s+1} \leftarrow (d(t_s - T[s + 1]h) + T[s + m + 1]) \bmod q$ 
14:   return -1

```

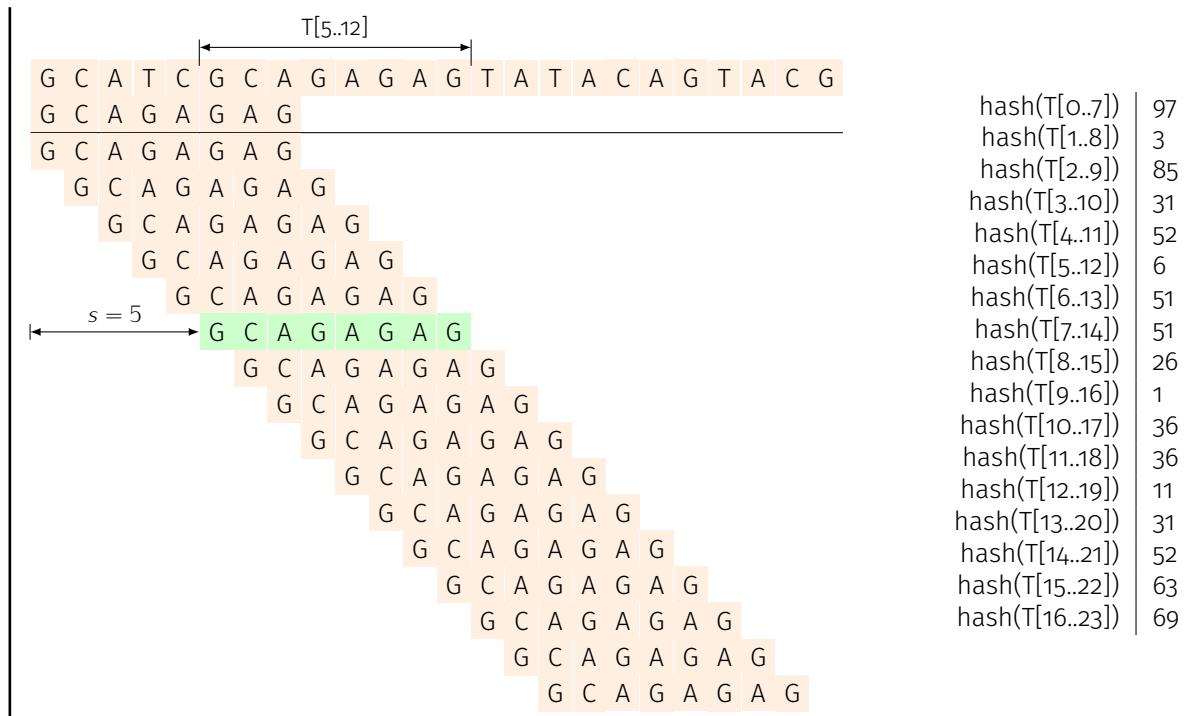
Ejemplo 23 | hash( $P[0..7]$ ) = 6



**Figura 19.** Richard M. Karp



**Figura 20.** Michael O. Rabin



## SECCIÓN 20

# Knuth-Morris-Pratt

KMP es uno de los algoritmos más famoso para resolver el problema de emparejamiento de cadenas, de creación de Knuth, Morris y Pratt(1977). Éste obtiene un mecanismo para computar el sufijo más largo del texto, que también es prefijo de P. Cuando la longitud del sufijo en el texto es igual a  $|P|$ , nos encontramos ante un emparejamiento.

---

**Algoritmo 35** Función fracaso

```

1: function COMPUTE-PREFIX-FUNCTION( $P$ )  $\triangleright P[1 \dots m]$ 
2:   Sea  $\pi[1 \dots m]$ 
3:    $\pi[1] \leftarrow 0$ 
4:    $k \leftarrow 0$ 
5:   for  $q \leftarrow 2$  to  $m$  do
6:     while  $k > 0$  and  $P[k + 1] \neq P[q]$  do
7:        $k \leftarrow \pi[k]$ 
8:       if  $P[k + 1] = P[q]$  then
9:          $k \leftarrow k + 1$ 
10:         $\pi[q] \leftarrow k$ 
11:   return  $\pi$ 

```

```

0 1 2 3 4 5
A B A C A B
0|1|0|1|1|2|
012345678901234567890
ABACABAABCABACABAABB
ABACAB
    ABACAB
        ABACAB
            ABACAB
                ABACAB
                    ABACAB
012345678901234567890
Found pattern at index 10

```

**Algoritmo 36** KMP

---

```

1: function KMPSTRINGMATCH( $T, P$ )
2:    $\pi \leftarrow \text{COMPUTE-PREFIX-FUNCTION}(P)$                                  $\triangleright T[1 \dots n], P[1 \dots m]$ 
3:    $q \leftarrow 0$                                                                 $\triangleright \text{Preprocessing}$ 
4:   for  $i \leftarrow 1$  to  $n$  do
5:     while  $q > 0$  and  $P[q + 1] \neq T[i]$  do
6:        $q \leftarrow \pi[q]$ 
7:     if  $P[q + 1] = T[i]$  then
8:        $q \leftarrow q + 1$ 
9:     if  $q = m$  then
10:      print "Pattern occurs with shift"  $i - m$ 
11:       $q \leftarrow \pi[q]$ 

```

---

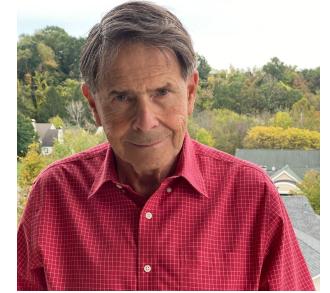
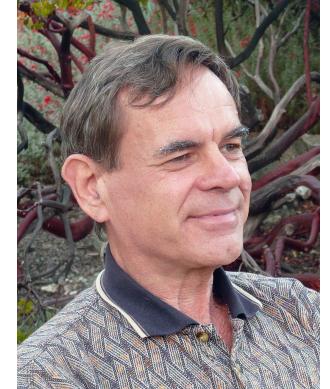
```

void KMPSearch(char* pat, char* txt)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int lps[M];

    computeLPSArray(pat, M, lps);

    int i = 0; // index for txt[]
    int j = 0; // index for pat[]
    while (i < N) {
        if (pat[j] == txt[i]) {
            j++; i++;
        }
        if (j == M) {
            printf("Found pattern at index %d\n", i - j);
            j = lps[j - 1];
        }
        else if (i < N && pat[j] != txt[i]) {
            if (j != 0) j = lps[j - 1];
            else i = i + 1;
        }
    }
}

```

**Figura 21.** James H. Morris**Figura 22.** Vaughan Pratt

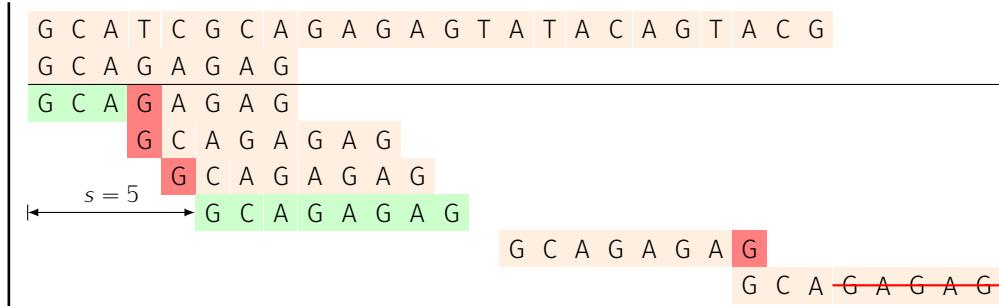
```

void computeLPSArray(char* pat, int M, int* lps)
{
    lps[0] = 0;
    int j = 0;
    int i = 1;

    while (i < M) {
        if (pat[j] == pat[i]) {
            lps[i] = j + 1;
            i++;
            j++;
        } else if (j > 0) {
            j = lps[j-1];
        } else { // no match
            lps[i] = 0;
            i++;
        }
    }
}

```

Ejemplo 24



SECCIÓN 21

## Boyer-Moore



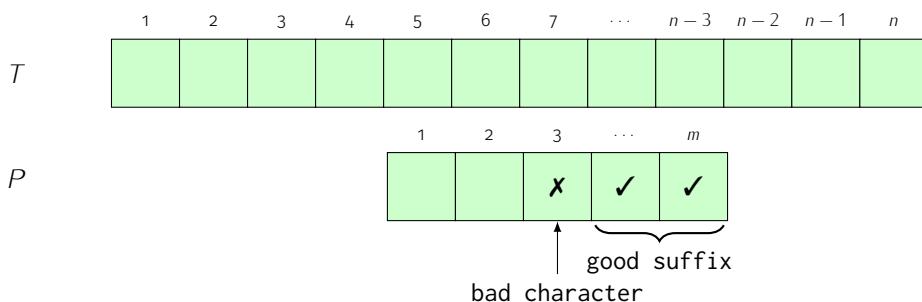
Figura 23. Robert S.Boyer

El algoritmo de Boyer y Moore se publicó el mismo año que KMP, 1977. La idea es buscar el patrón de derecha a izquierda. BM usa dos heurísticas para determinar la distancia de desplazamiento: *bad-character* y *good-suffix*.

**bad-character:** se refiere al carácter que causa la discrepancia(*mismatch*), alinea la posición del texto que provocó la colisión con el primer carácter del patrón que se empareja con él;

**good-suffix:** al mover el patrón hacia la derecha, se empareja con un fragmento de texto que se emparejó anteriormente.

El algoritmo decide cual de las heurísticas debe utilizar eligiendo la que provoca mayor desplazamiento del patrón. Sin embargo, esta elección implica realizar una comparación entre dos enteros para cada carácter leído del texto.



SUBSECCIÓN 21.1

## Horspool

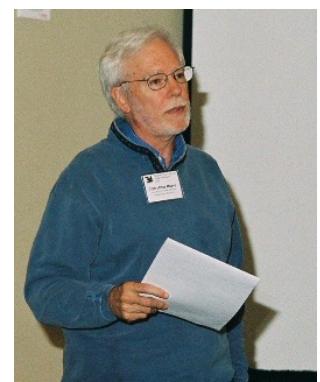


Figura 24. J. Strother Moore



Figura 25. Nigel Horspool

La primera simplificación de BM es Boyer-Moore-Horspool (1980). En comparación con el algoritmo de Boyer-Moore, Horspool mejora la regla del carácter incorrecto(*badcharacter*). Para volver a hacer coincidir, el algoritmo Horspool alinea el último carácter de la cadena principal en la ventana de coincidencia actual con el carácter más cercano a él en la cadena de patrón.

**Algoritmo 37** Función next

---

```

1: function COMPUTE-NEXT-FUNCTION( $P$ ) ▷  $P[0 \dots m - 1]$ 
2:    $\lambda[x] = \min\{j(j = m) \wedge ((1 \leq j < m) \vee (P[m - j] = x))\}$ 

```

---

**Algoritmo 38** BMH

---

```

1: function BMHSTRINGMATCH( $T, P$ ) ▷  $T[0 \dots n - 1], P[0 \dots m - 1]$ 
2:    $\lambda \leftarrow \text{COMPUTE-NEXT-FUNCTION}(P)$  ▷ Preprocessing
3:    $i \leftarrow m$ 
4:   while  $i \leq m$  do
5:      $k \leftarrow i$ 
6:      $j \leftarrow m$ 
7:     while  $j \geq 0$  and  $T[k] = P[j]$  do
8:        $k \leftarrow k - 1$ 
9:        $j \leftarrow k - 1$ 
10:    if  $j < 0$  then
11:      print "Pattern occurs with shift"  $k + 1$ 
12:     $i \leftarrow i + \lambda[T[i]]$ 

```

---

```

#define NO_OF_CHARS 256
void badCharHeuristic( char *str, int size,
                      int d[NO_OF_CHARS])
{
    int i;

    // Initialize all occurrences as m
    for (i = 0; i < NO_OF_CHARS; i++)
        d[i] = m;

    // Fill the actual value of last occurrence
    // of a character
    for (i = 0; i < size; i++)
        d[(int) str[i]] = m-i; // shift
}

```

```

void search( char *txt, char *pat)
{
    int m = strlen(pat);
    int n = strlen(txt);

    int d[NO_OF_CHARS];

    badCharHeuristic(pat, m, d);

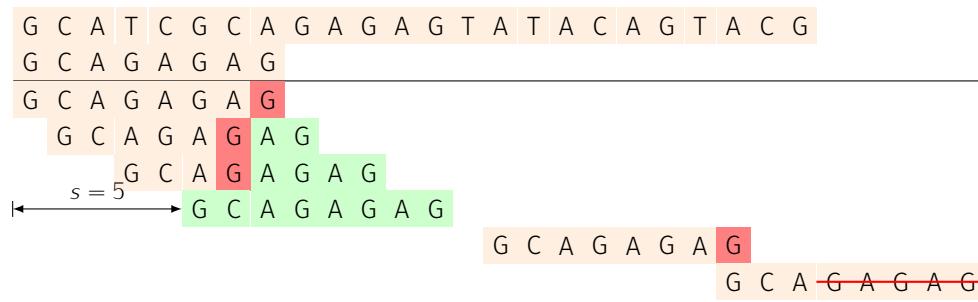
    int s = m;
    while(s <= n){
        int k=s;
        int j = m;

        while(j > 0 && pat[j] == txt[k]){
            j--;k--;
        }

        if (j == 0) {
            printf("\n pattern occurs at shift = %d", k+1);
            s=s+d[(int) txt[s]];
        }
    }
}

```

Ejemplo 25



**22.1** Sean  $A = a_1a_2 \dots a_n$  y  $B = b_1b_2 \dots b_n$  dos cadenas de largo  $n$ . Se define que  $B$  es una *rotación cíclica* de  $A$  si existe un índice  $k$ ,  $1 \leq k \leq n$ , tal que  $\forall i, 1 \leq i \leq n, a_i = b_{(k+i) \bmod n}$ . Describa un algoritmo que utilice Knuth-Morris-Pratt para determinar si  $B$  es una rotación cíclica de  $A$ .

**22.2** Aplicar el algoritmo KMP, BMH y BMS a los siguientes datos:  
patrón: abbababa y texto: aabbaabababbababb

Debe mostrar los cálculos previos (como por ejemplo la función de fracaso) e ir mostrando paso a paso como se realiza la búsqueda.

**22.3** Hacer las modificaciones necesarias en el algoritmo BMS para que cuente las veces que ocurre el match con el patrón.

**22.4** Modifique el algoritmo Boyer-Moore-Horspool para que, en caso de que no se haya podido calzar el patrón completo en el texto, devuelva el sufijo del patrón más largo encontrado.

**22.5**

- Calcule la función de fracaso del algoritmo Knuth-Morris-Pratt para el patrón de búsqueda: **abracadabra**.

- Aplique el algoritmo de Boyer-Moore-Sunday para buscar el *patrón* dentro del *texto*.

patrón: **tex**

texto: **un ejemplo de texto**.

Muestre paso a paso la alineación del patrón con el texto, e indique los caracteres que se comparan.

- Contruya un ejemplo de un patrón de largo  $m$  y un texto de largo  $n$  en el cual el algoritmo de Boyer-Moore-Horspool funcione en forma ineficiente.  
¿Cuánto demora el algoritmo en ese caso ?

**22.6** Modifique el algoritmo Knuth-Morris-Pratt para que retorne la posición del prefijo más largo del patrón (cadena de largo  $m$ ) y que calce con algún substring de  $x$  (texto de largo  $n > m$ ). Por ejemplo, para el texto dado por  $x = abbababba$  y el patrón  $y = ababa$  se debe obtener que el prefijo más largo es  $abab$  y se encuentra a partir de la posición 3 del texto  $x$  dado.

**22.7** Hacer las modificaciones necesarias en el algoritmo BMS para que:

- cuente las veces que ocurre el match con el patrón
- cada vez que haya un match con **patron1** cambiar el texto por **patron2**

**22.8** Indique cuando ocurre el peor caso y cuando el mejor caso para los algoritmos KMP y BMS. Dé un ejemplo.

**22.9** Dadas dos cadenas, ¿cómo se puede encontrar la sucadena común más larga?

**22.10** Dado un texto  $T$  ¿cómo se puede encontrar la subcadena que es el palíndrome más largo de  $T$ ?

**22.11** Implemente una función que permita invertir las palabras de una oración. Ej: In:"This is a Career Monk String", Out: "String Monk Career a is This". Calcule la complejidad de su implementación.

**22.12** Implemente una función que permita encontrar la variación con repetición de los caracteres de una cadena. Ej: In:AB, Out: AA, AB, BA, BB. Calcule la complejidad de su implementación.

### 22.13 Problema de Búsqueda de Patrones en Genomas

- Dado un genoma (texto largo) y un patrón de ADN, implemente un algoritmo que encuentre todas las ocurrencias del patrón considerando que:
  - El ADN solo contiene los caracteres A, C, G, T
  - Se debe considerar que las cadenas son circulares (el final se conecta con el inicio)
  - Se debe reportar la posición inicial de cada coincidencia
- Compare la eficiencia de usar KMP vs Boyer-Moore para este caso específico

### 22.14 Problema de Búsqueda de Palabras en Documentos

- Implemente un sistema que permita buscar palabras en un documento de texto considerando:
  - Ignorar mayúsculas/minúsculas
  - Ignorar acentos
  - Permitir búsqueda con comodines (\*)
- Analice qué algoritmo es más apropiado para cada caso

### 22.15 Problema de Detección de Plagio

- Implemente un sistema que detecte si un texto es una copia parcial de otro
  - Considere que el plagio puede ser una reordenación de palabras
  - Permita un cierto porcentaje de diferencia
  - Reporte las secciones similares encontradas
- Compare la eficiencia de diferentes algoritmos para este caso

SECCIÓN 23

## Análisis de Complejidad Detallado

---

SUBSECCIÓN 23.1

### Complejidad Temporal

---

#### ■ Fuerza Bruta

- Peor caso:  $\Theta((n - m + 1)m)$ 
  - Ocurre cuando el patrón aparece al final del texto
  - Se compara cada carácter del patrón con cada posición del texto
- Mejor caso:  $\Theta(n)$ 
  - Ocurre cuando el primer carácter nunca coincide
  - Solo se hace una comparación por posición

#### ■ KMP (Knuth-Morris-Pratt)

- Preprocesamiento:  $\Theta(m)$ 
  - Construcción de la tabla de fallos
  - Solo depende del tamaño del patrón

- Búsqueda:  $\Theta(n)$ 
  - Cada carácter del texto se procesa una sola vez
  - No hay retroceso en el texto
- Total:  $\Theta(n + m)$

#### ■ Boyer-Moore

- Preprocesamiento:  $\Theta(m + |\Sigma|)$ 
  - Construcción de tablas bad-character y good-suffix
  - $|\Sigma|$  es el tamaño del alfabeto
- Búsqueda:  $\Theta(n)$  en promedio
  - Mejor que KMP en la práctica para patrones largos
  - Puede saltar múltiples caracteres en una sola comparación
- Peor caso:  $\Theta(mn)$ 
  - Ocurre con patrones que tienen muchas repeticiones
  - Ejemplo: patrón "aaaa" en texto "aaaaaaaa"

#### ■ Boyer-Moore-Horspool (BMH)

- Preprocesamiento:  $\Theta(|\Sigma|)$ 
  - Solo construye la tabla bad-character
  - Más simple que BM original
- Búsqueda:  $\Theta(n)$  en promedio
  - Usa solo la heurística bad-character
  - Menos eficiente que BM original pero más simple
- Peor caso:  $\Theta(mn)$ 
  - Similar a BM original
  - Ocurre con patrones repetitivos

#### ■ Boyer-Moore-Sunday (BMS)

- Preprocesamiento:  $\Theta(|\Sigma|)$ 
  - Construye tabla de desplazamiento
  - Similar a BMH en complejidad
- Búsqueda:  $\Theta(n)$  en promedio
  - Usa el carácter siguiente al final de la ventana
  - Puede ser más eficiente que BMH en algunos casos
- Peor caso:  $\Theta(mn)$ 
  - Similar a BM y BMH
  - Ocurre con patrones repetitivos

#### ■ Rabin-Karp

- Preprocesamiento:  $\Theta(m)$ 
  - Cálculo del hash del patrón
- Búsqueda:  $\Theta(n)$  en promedio
  - Asumiendo buenas funciones hash
  - Pocas colisiones
- Peor caso:  $\Theta((n - m + 1)m)$ 
  - Ocurre con muchas colisiones de hash
  - Se degenera a fuerza bruta

SUBSECCIÓN 23.2

## Complejidad Espacial

---

- **Fuerza Bruta:**  $\Theta(1)$ 
  - Solo usa variables de control
- **KMP:**  $\Theta(m)$ 
  - Almacena la tabla de fallos
- **Boyer-Moore:**  $\Theta(m + |\Sigma|)$ 
  - Tablas bad-character y good-suffix
- **Boyer-Moore-Horspool:**  $\Theta(|\Sigma|)$ 
  - Solo tabla bad-character
  - Menos memoria que BM original
- **Boyer-Moore-Sunday:**  $\Theta(|\Sigma|)$ 
  - Tabla de desplazamiento
  - Similar a BMH
- **Rabin-Karp:**  $\Theta(1)$ 
  - Solo variables para hash

SUBSECCIÓN 23.3

## Relación entre las Heurísticas

---

- **Boyer-Moore Original**
  - Usa dos heurísticas:
    - Bad-character: usa el carácter que no coincide
    - Good-suffix: usa el sufijo que sí coincide
  - Toma el máximo desplazamiento entre ambas
- **Boyer-Moore-Horspool**
  - Simplifica BM usando solo bad-character
  - Modifica la heurística para usar el último carácter de la ventana
  - Más simple de implementar pero menos eficiente
- **Boyer-Moore-Sunday**
  - Usa una heurística similar a bad-character
  - Considera el carácter siguiente al final de la ventana
  - Puede ser más eficiente que BMH en algunos casos

SUBSECCIÓN 23.4

## Comparación de Algoritmos

---

### ■ Fuerza Bruta

- Ventajas:
  - Simple de implementar
  - Sin memoria adicional
- Desventajas:
  - Ineficiente para patrones largos
  - Muchas comparaciones innecesarias

### ■ KMP

- Ventajas:
  - Garantizado  $\Theta(n)$  en el peor caso
  - Bueno para patrones cortos
- Desventajas:
  - Preprocesamiento necesario
  - Memoria adicional

### ■ Boyer-Moore

- Ventajas:
  - Muy eficiente en la práctica
  - Bueno para patrones largos
- Desventajas:
  - Complejo de implementar
  - Más memoria

### ■ Boyer-Moore-Horspool

- Ventajas:
  - Simple de implementar
  - Menos memoria
- Desventajas:
  - Menos eficiente que BM original
  - No usa good-suffix

### ■ Boyer-Moore-Sunday

- Ventajas:
  - Simple de implementar
  - Menos memoria
- Desventajas:
  - Menos eficiente que BMH
  - No usa good-suffix

### ■ Rabin-Karp

- Ventajas:
  - Fácil de implementar
  - Bueno para múltiples patrones
- Desventajas:
  - Puede degenerar a fuerza bruta
  - Sensible a colisiones de hash

## SECCIÓN 24

**Implementaciones en C**

## SUBSECCIÓN 24.1

**Fuerza Bruta**

```
#include <stdio.h>
#include <string.h>

int bruteForceSearch(const char* text, const char* pattern) {
    int n = strlen(text);
    int m = strlen(pattern);

    for (int i = 0; i <= n - m; i++) {
        int j = 0;
        while (j < m && text[i + j] == pattern[j]) {
            j++;
        }
        if (j == m) {
            return i; // Encontrado en posición i
        }
    }
    return -1; // No encontrado
}

int main() {
    const char* text = "GCATCGCAGAGAGATACAGATACTAC";
    const char* pattern = "GCAGAGAG";

    int pos = bruteForceSearch(text, pattern);
    if (pos != -1) {
        printf("Patrón encontrado en posición: %d\n", pos);
    } else {
        printf("Patrón no encontrado\n");
    }
    return 0;
}
```

## SUBSECCIÓN 24.2

**KMP (Knuth-Morris-Pratt)**

```
#include <stdio.h>
#include <string.h>

void computeLPS(const char* pattern, int* lps, int m) {
    int len = 0;
    lps[0] = 0;
    int i = 1;

    while (i < m) {
        if (pattern[i] == pattern[len]) {
            lps[i] = len + 1;
            i++;
            len++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}
```

```

        len++;
        lps[i] = len;
        i++;
    } else {
        if (len != 0) {
            len = lps[len - 1];
        } else {
            lps[i] = 0;
            i++;
        }
    }
}

int KMPSearch(const char* text, const char* pattern) {
    int n = strlen(text);
    int m = strlen(pattern);
    int lps[m];

    computeLPS(pattern, lps, m);

    int i = 0; // índice para text[]
    int j = 0; // índice para pattern[]

    while (i < n) {
        if (pattern[j] == text[i]) {
            j++;
            i++;
        }

        if (j == m) {
            return i - j; // Encontrado en posición i-j
        } else if (i < n && pattern[j] != text[i]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }
    return -1; // No encontrado
}

int main() {
    const char* text = "GCATCGCAGAGAGATACAGATACTAC";
    const char* pattern = "GCAGAGAG";

    int pos = KMPSearch(text, pattern);
    if (pos != -1) {
        printf("Patrón encontrado en posición: %d\n", pos);
    } else {
        printf("Patrón no encontrado\n");
    }
    return 0;
}

```

## SUBSECCIÓN 24.3

**Boyer-Moore**

```

#include <stdio.h>
#include <string.h>

#define NO_OF_CHARS 256

void badCharHeuristic(const char* str, int size, int badchar[NO_OF_CHARS]) {
    // Inicializar todas las ocurrencias como -1
    for (int i = 0; i < NO_OF_CHARS; i++)
        badchar[i] = -1;

    // Llenar el valor actual de la última ocurrencia
    for (int i = 0; i < size; i++)
        badchar[(int)str[i]] = i;
}

```

```

void goodSuffixHeuristic(const char* pattern, int m, int* gs) {
    // Inicializar el array
    for (int i = 0; i < m; i++)
        gs[i] = m;

    // Calcular el sufijo más largo que también es prefijo
    int i = m, j = m + 1;
    while (i > 0) {
        if (pattern[i - 1] == pattern[j - 1]) {
            gs[j - 1] = i - 1;
            i--;
            j--;
        } else {
            if (i < j)
                gs[j - 1] = i;
            j--;
        }
    }
}

int boyerMooreSearch(const char* text, const char* pattern) {
    int n = strlen(text);
    int m = strlen(pattern);
    int badchar[NO_OF_CHARS];
    int gs[m];

    badCharHeuristic(pattern, m, badchar);
    goodSuffixHeuristic(pattern, m, gs);

    int s = 0; // s es el desplazamiento del patrón
    while (s <= (n - m)) {
        int j = m - 1;

        // Mientras los caracteres coinciden
        while (j >= 0 && pattern[j] == text[s + j])
            j--;

        if (j < 0)
            return s; // Patrón encontrado

        // Calcular el desplazamiento usando ambas heurísticas
        int badCharShift = j - badchar[(int)text[s + j]];
        int goodSuffixShift = gs[j];

        // Usar el mayor desplazamiento
        s += (badCharShift > goodSuffixShift) ? badCharShift : goodSuffixShift;
    }
    return -1; // No encontrado
}

int main() {
    const char* text = "GCATCGCAGAGAGATACAGATACTAC";
    const char* pattern = "GCAGAGAG";

    int pos = boyerMooreSearch(text, pattern);
    if (pos != -1) {
        printf("Patrón encontrado en posición: %d\n", pos);
    } else {
        printf("Patrón no encontrado\n");
    }
    return 0;
}

```

## SUBSECCIÓN 24.4

**Rabin-Karp**

```

#include <stdio.h>
#include <string.h>

#define d 256 // Tamaño del alfabeto
#define q 101 // Número primo para el hash

int rabinKarpSearch(const char* text, const char* pattern) {
    int n = strlen(text);

```

```

int m = strlen(pattern);
int h = 1;
int p = 0; // hash value for pattern
int t = 0; // hash value for text

// Calcular h = d^(m-1) mod q
for (int i = 0; i < m - 1; i++)
    h = (h * d) % q;

// Calcular el hash inicial del patrón y la primera ventana del texto
for (int i = 0; i < m; i++) {
    p = (d * p + pattern[i]) % q;
    t = (d * t + text[i]) % q;
}

// Deslizar el patrón sobre el texto
for (int i = 0; i <= n - m; i++) {
    // Verificar si los hashes coinciden
    if (p == t) {
        // Verificar si los caracteres coinciden
        int j;
        for (j = 0; j < m; j++) {
            if (text[i + j] != pattern[j])
                break;
        }
        if (j == m)
            return i; // Patrón encontrado
    }

    // Calcular el hash para la siguiente ventana
    if (i < n - m) {
        t = (d * (t - text[i] * h) + text[i + m]) % q;
        if (t < 0)
            t = (t + q);
    }
}
return -1; // No encontrado
}

int main() {
    const char* text = "GCATCGCAGAGAGATACAGATACTAC";
    const char* pattern = "GCAGAGAG";

    int pos = rabinKarpSearch(text, pattern);
    if (pos != -1) {
        printf("Patrón encontrado en posición: %d\n", pos);
    } else {
        printf("Patrón no encontrado\n");
    }
    return 0;
}

```

## SUBSECCIÓN 24.5

**Boyer-Moore-Horspool (BMH)**

```

#include <stdio.h>
#include <string.h>

#define NO_OF_CHARS 256

void badCharHeuristic(const char* str, int size, int badchar[NO_OF_CHARS]) {
    // Inicializar todas las ocurrencias como m
    for (int i = 0; i < NO_OF_CHARS; i++)
        badchar[i] = size;

    // Llenar el valor actual de la última ocurrencia
    for (int i = 0; i < size - 1; i++)
        badchar[(int)str[i]] = size - 1 - i;
}

int BMHSearch(const char* text, const char* pattern) {
    int n = strlen(text);
    int m = strlen(pattern);
    int badchar[NO_OF_CHARS];

```

```

badCharHeuristic(pattern, m, badchar);

int i = m - 1; // Posición actual en el texto
while (i < n) {
    int k = i;
    int j = m - 1;

    // Comparar de derecha a izquierda
    while (j >= 0 && text[k] == pattern[j]) {
        k--;
        j--;
    }

    if (j < 0) {
        return k + 1; // Patrón encontrado
    }

    // Desplazamiento usando la heurística bad-character
    i += badchar[(int)text[i]];
}
return -1; // No encontrado
}

int main() {
    const char* text = "GCATCGCAGAGAGATACAGATACTAC";
    const char* pattern = "GCAGAGAG";

    int pos = BMHSearch(text, pattern);
    if (pos != -1) {
        printf("Patrón encontrado en posición: %d\n", pos);
    } else {
        printf("Patrón no encontrado\n");
    }
    return 0;
}

```

## SUBSECCIÓN 24.6

**Boyer-Moore-Sunday (BMS)**

```

#include <stdio.h>
#include <string.h>

#define NO_OF_CHARS 256

void sundayHeuristic(const char* pattern, int size, int shift[NO_OF_CHARS]) {
    // Inicializar todas las ocurrencias como m+1
    for (int i = 0; i < NO_OF_CHARS; i++)
        shift[i] = size + 1;

    // Llenar el valor actual de la última ocurrencia
    for (int i = 0; i < size; i++)
        shift[(int)pattern[i]] = size - i;
}

int BMSearch(const char* text, const char* pattern) {
    int n = strlen(text);
    int m = strlen(pattern);
    int shift[NO_OF_CHARS];

    sundayHeuristic(pattern, m, shift);

    int i = 0; // Posición actual en el texto
    while (i <= n - m) {
        int j = 0;

        // Comparar de izquierda a derecha
        while (j < m && text[i + j] == pattern[j]) {
            j++;
        }

        if (j == m) {
            return i; // Patrón encontrado
        }
    }
}

```

```
// Desplazamiento usando el carácter siguiente al final de la ventana
if (i + m < n) {
    i += shift[(int)text[i + m]];
} else {
    break;
}
return -1; // No encontrado
}

int main() {
    const char* text = "GCATCGCAGAGAGATACAGATACTAC";
    const char* pattern = "GCAGAGAG";

    int pos = BMSSearch(text, pattern);
    if (pos != -1) {
        printf("Patrón encontrado en posición: %d\n", pos);
    } else {
        printf("Patrón no encontrado\n");
    }
    return 0;
}
```

# Repaso Matemáticas para Ciencias de la Computación

SUBSECCIÓN 24.7

## Potencias

---

$$a^0 = 1, a \neq 0$$

$$a^p a^q = a^{p+q}$$

$$\frac{a^p}{a^q} = a^{p-q}$$

$$(a^p)^q = a^{pq}$$

$$a^p + a^p = 2a^p \neq a^{2p}$$

$$a^{-p} = \frac{1}{a^p}$$

$$(ab)^p = a^p b^p$$

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

$$\sqrt[n]{a} = a^{1/n}$$

$$\sqrt[n]{a^m} = a^{m/n}$$

$$\sqrt[n]{\frac{a}{b}} = \frac{\sqrt[n]{a}}{\sqrt[n]{b}}$$

SUBSECCIÓN 24.8

## Logaritmos

---

Sea  $b$  un número real positivo mayor a 1,  $x$  un número real y suponga que para algún número real positivo  $y$  tenemos  $y = b^x$ . Entonces,  $x$  se llama el *logaritmo de y en base b*, y se escribe:

$$x = \log_b y$$

Aquí  $b$  es la base del logaritmo.

### 24.8.1. Identidades y propiedades

$$\log_a 1 = 0, \log_a a = 1$$

$$\log_b xy = \log_b x + \log_b y$$

$$\log_b \frac{x}{y} = \log_b x - \log_b y$$

$$\log_b c^y = y \log_b c, \text{ si } c > 0$$

$$e = \lim_{n \rightarrow \infty} \left(1 + \frac{1}{n}\right)^n = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots = 2.7182818\dots$$

$$\log_a x = \log_b x \log_a b \text{ o } \log_b x = \frac{\log_a x}{\log_a b}$$

$$x^{\log_b y} = y^{\log_b x}, x, y > 0$$

SUBSECCIÓN 24.9

## Piso(*floor*) y techo(*ceiling*)

$\lfloor x \rfloor$  denota el piso de  $x$ , y se define como el entero más grande cercano o igual a  $x$ .  $\lceil x \rceil$  denota el techo de  $x$ , y se define como el entero más cercano o igual a  $x$ . Ejemplo:

$$\lfloor \sqrt{2} \rfloor = 1, \lceil \sqrt{2} \rceil = 2, \lfloor \sqrt{-2.5} \rfloor = -3, \lceil \sqrt{-2.5} \rceil = -2$$

$$\lfloor \pi \rfloor = 3, \lceil \pi \rceil = 4, \lfloor e \rfloor = 2, \lceil e \rceil = 3$$

#### 24.9.1. Identidades y propiedades

$$a - 1 < \lfloor a \rfloor \leq a \leq \lceil a \rceil < a + 1$$

$$\lceil x/2 \rceil + \lfloor x/2 \rfloor = x$$

$$\lfloor -x \rfloor = -\lceil x \rceil$$

$$\lceil -x \rceil = \lfloor -x \rfloor$$

$$\lceil \sqrt{\lceil x \rceil} \rceil = \lceil \sqrt{x} \rceil$$

$$\lceil n/2 \rceil = \begin{cases} n/2 & \text{si } n \text{ es par} \\ (n-1)/2 & \text{si } n \text{ es impar} \end{cases}$$

SUBSECCIÓN 24.10

## Factorial y coeficiente Binomial

$$0! = 1, n! = n(n-1)!, \sin \geq 1$$

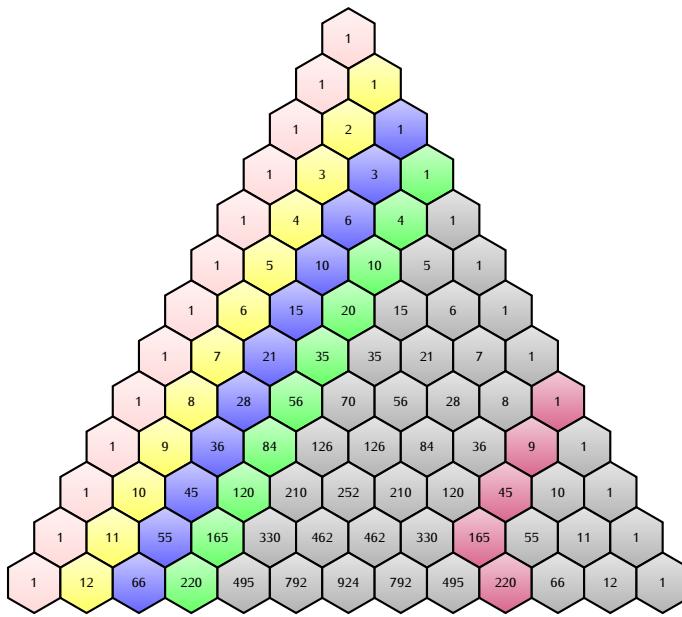
$$30! = 265252859812191058636308480000000.$$

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n, e = 2.7182818\dots$$

$$30! \approx 264517095922964306151924784891709$$

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + \frac{1}{288n^2} - \frac{139}{5140n^3} + O\left(\frac{1}{n^4}\right)\right)$$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$



### 24.10.1. Identidades y propiedades

$$\binom{n}{k} = \binom{n}{n-k}$$

$$\binom{n}{n} = \binom{n}{0} = 1$$

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

$$\binom{n}{k} = \frac{n}{k} \binom{n-1}{n-k}$$

$$(1+x)^n = \sum_{j=0}^n \binom{n}{j} x^j$$

$$(x+y)^n = \sum_{k=0}^n \binom{n}{k} x^k y^{n-k}$$

$$\binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{n} = 2^n$$

$$1 + 2 + \dots + n = \binom{1}{1} + \binom{2}{1} + \cdots + \binom{n}{1} = \binom{n+1}{2}$$

$$\sum_{j \text{ par}} \binom{n}{j} = \sum_{j \text{ impar}} \binom{n}{j}$$

$$\sum_{k \leq n} \binom{r+k}{k} = \binom{r+n+1}{n}$$

$$\sum_{m \leq k \leq n} \binom{k}{m} = \binom{n+1}{m+1}$$

SUBSECCIÓN 24.11

## Sumatorias

---

$$\sum_{j=1}^n a_j = \sum_{1 \leq j \leq n} a_j = a_1 + a_2 + a_3 + \cdots + a_n$$

### 24.11.1. Identidades y propiedades

$$\sum_{j=1}^n a_{n-j+1} = a_{n-1+1} + a_{n-2+1} + \cdots + a_{n-n+1} = \sum_{j=1}^n a_j$$

$$\begin{aligned}\sum_{j=1}^n a_{n-j} &= \sum_{1 \leq j \leq n} a_{n-j} \\ &= \sum_{1 \leq n-j \leq n} a_{n-(n-j)} \\ &= \sum_{1-n \leq n-j-n \leq n-n} a_{n-(n-j)} \\ &= \sum_{1-n \leq -j \leq 0} a_j \\ &= \sum_{0 \leq j \leq n-1} a_j \\ &= \sum_{j=0}^{n-1} a_j\end{aligned}$$

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

$$\sum_{k=1}^n k(k+1)(k+2) = \frac{n(n+1)(n+2)(n+3)}{4}$$

$$\sum_{j=1}^n j^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=1}^n \frac{1}{k(k+1)(k+2)} = \frac{n(n+3)}{4(n+1)(n+2)}$$

$$\sum_{j=1}^n j^3 = \frac{n^2(n+1)^2}{4}$$

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

$$\sum_{k=1}^n k(k+1) = \frac{n(n+1)(n+2)}{3}$$

$$\sum_{j=0}^{\infty} c^j = \frac{1}{1-c}, |c| < 1$$

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

$$\sum_{j=0}^n jc^j = \sum_{j=1}^n jc^j = \frac{nc^{n+2} - nc^{n+1} - c^{n+1} + c}{(c-1)^2}, c \neq 1$$

$$\sum_{j=1}^n c^j = \frac{c^{n+1} - 1}{c - 1}, c \neq 1$$

$$\sum_{j=0}^{\infty} jc^j = \frac{c}{(1-c)^2}, |c| < 1$$

$$\frac{\log(n+1)}{\log e} \leq \sum_{j=1}^n \frac{1}{j} \leq \frac{\log n}{\log e} + 1$$

$$H_n = \sum_{j=1}^n \frac{1}{j}$$

SUBSECCIÓN 24.12

## Series

---

Sea  $a_1, a_2, \dots, a_n$ , donde  $a_i = a_1 + (i-1)k$ :

$$a_1 + a_2 + \dots + a_n = \frac{n(a_1 - a_n)}{2}$$

Sea  $a_1, a_2, \dots, a_n$ , donde  $a_i = ar^{i-1}$ :

$$a + ar + ar^2 + \dots + ar^{n-1} = \frac{a(1 - r^n)}{1 - r} = \frac{a_1 - ra_n}{1 - r}$$

$$a + (a+d)r + (a+2d)r^2 + \dots + (a+(n-1)d)r^{n-1} = \frac{a(1 - r^n)}{1 - r} + \frac{rd(1 - nr^{n-1} + (n-1)r^n)}{(1 - r)^2}$$

$$1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$$

$$1^2 + 2^2 + 3^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

$$1^3 + 2^3 + 3^3 + \dots + n^3 = \frac{n^2(n+1)^2}{4} = (1 + 2 + 3 + \dots + n)^2$$

$$2^0 + 2^1 + 2^2 + \dots + 2^n = 2^{n+1} - 1$$

$$a^0 + a^1 + a^2 + \dots + a^n = \frac{a^{n+1} - 1}{a - 1}, \text{ y si } 0 < a < 1 \text{ entonces } \sum_{i=0}^n a^i \leq \frac{1}{1-a}$$

$$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \dots = \ln 2$$

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots = \frac{\pi}{4}$$

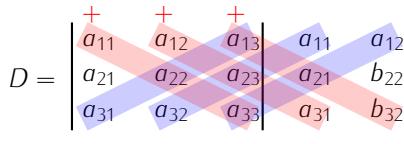
$$\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \dots = \frac{\pi^2}{6}$$

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{N} \approx \ln N, \text{ el error tiende a } \gamma$$

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} + \dots = O(\lg(n)), \gamma \approx 0.577215664901$$

### 24.12.1. Matrices

Determinante: Regla de Sarrus,

$$D = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11}a_{22}a_{33} + a_{21}a_{32}a_{13} + a_{31}a_{12}a_{23} - a_{13}a_{22}a_{31} - a_{23}a_{32}a_{11} - a_{33}a_{12}a_{21}$$


Dadas las matrices de la misma dimensión  $A = (a_{ij})$  y  $B = (b_{ij})$ , entonces:

$$A + B = (a_{ij} + b_{ij})$$

$$A - B = (a_{ij} - b_{ij})$$

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{bmatrix}$$

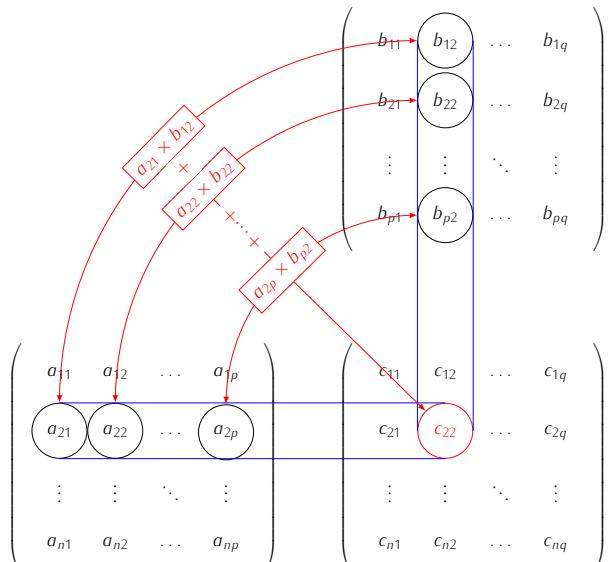
$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^n a_{ik}b_{kj}$$

$B: p$  filas  $q$  columnas

Transpuesta:

$$(A^T)_{ij} = A_{ji}, 1 \leq i \leq n, 1 \leq j \leq m$$

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}^T = \begin{bmatrix} a & c & e \\ b & d & f \end{bmatrix}$$

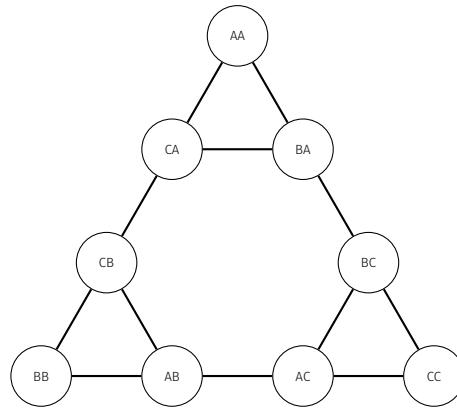
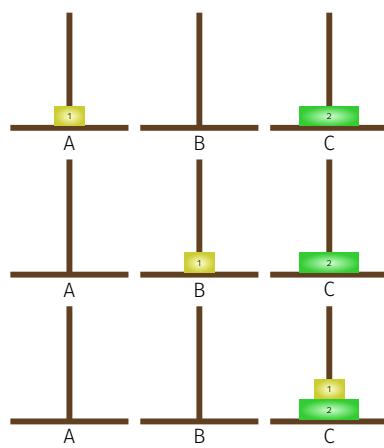
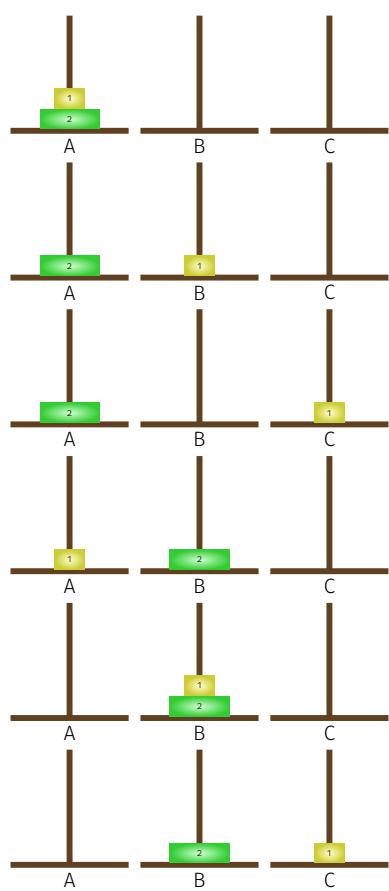
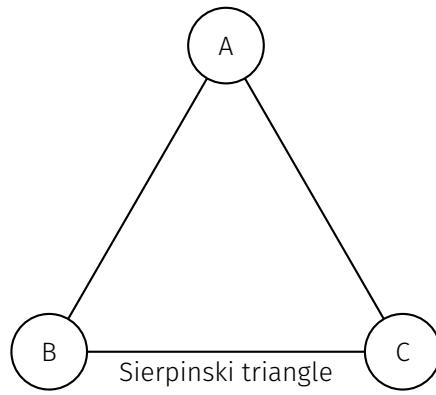
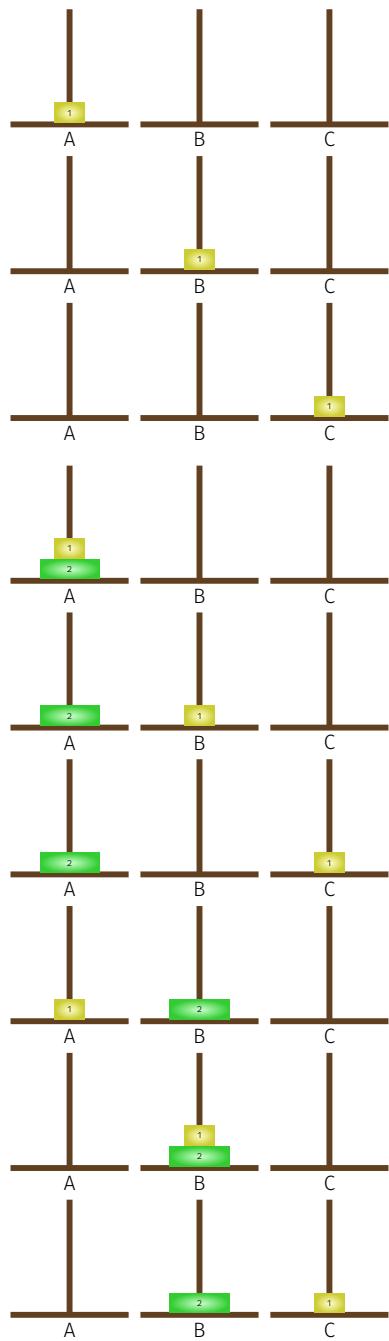


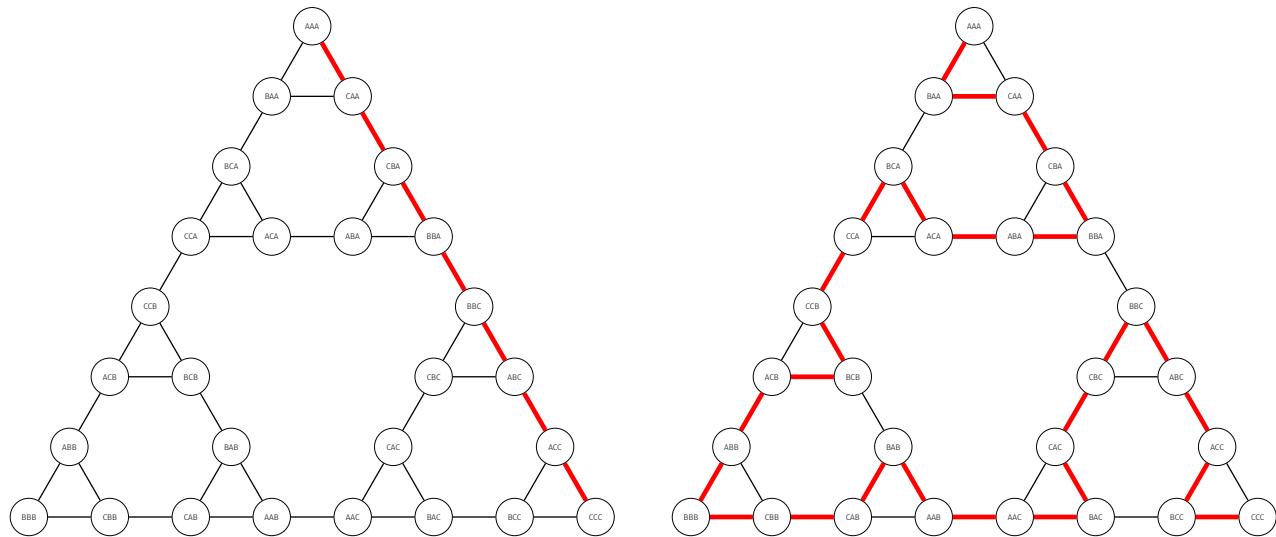
$C = A \times B: n$  filas  $q$  columnas

$\alpha A$	$v N$
$\beta B$	$\xi \Xi$
$\gamma \Gamma$	$o O$
$\delta \Delta$	$\pi \Pi$
$\epsilon \varepsilon E$	$\rho \varrho P$
$\zeta Z$	$\sigma \Sigma$
$\eta H$	$\tau T$
$\theta \vartheta \Theta$	$v Y$
$u I$	$\phi \varphi \Phi$
$\kappa K$	$\chi \chi X$
$\lambda \Lambda$	$\psi \Psi$
$\mu M$	$\omega \Omega$

#### SUBSECCIÓN 24.13

### Hanoi





## Referencias

- [1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT press, 2009.
- [2] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710, 1966.