

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*

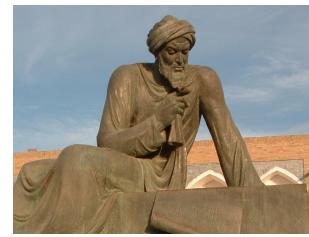
jose.canuman@umag.cl

Índice

I Referencias	2
II Introducción	4
1. Pero, ¿Qué es un algoritmo?	4
1.1. Los algoritmos en la historia	5
1.1.1. Antigüedad	5
1.1.2. Edad Media	5
1.1.3. Edad Moderna	5
1.2. Los límites de los algoritmos: P y NP	6
	7
III Análisis	9
2. ¿Cómo medir un algoritmo?	9
3. Orden asintótico	9
3.1. Simplificación	11
4. Análisis: Estimación de la ejecución	12
4.1. Peor caso, mejor caso y promedio	13
4.2. Mínimo	14
4.2.1. Búsqueda secuencial	15
5. Ordenación por comparación	17
5.1. Intercambio	17
5.2. Selección	18
5.3. Inserción	19
6. Solución de ecuaciones de recurrencias	20
6.1. Recurrencias homogéneas	20
6.2. Ecuaciones no homogéneas	21
6.3. Cambio de variables	22
6.4. Iteraciones	23
7. Cuestiones y problemas	26
IV Diseño	30

8. Divide y Vencerás	31
8.1. Características	31
8.2. Estructura general	31
8.3. Ecuaciones de Recurrencia	31
8.4. Casos Especiales	32
8.4.1. División en Dos Partes Iguales	32
8.4.2. División en Tres Partes Iguales	32
8.5. Teorema Maestro	32
8.6. Ejemplos Clásicos	32
8.6.1. Merge Sort	32
8.6.2. Multiplicación de Matrices de Strassen	32
8.6.3. Quick Sort (caso promedio)	33
8.6.4. Búsqueda Binaria	33
8.7. Ventajas y Desventajas	33
8.8. Búsqueda Binaria	34
8.9. Exponenciación	34
8.10. Mínimo y Máximo	36
8.11. Más problemas clásicos	36
9. Programación Dinámica	37
9.1. Características Principales	37
9.2. Estructura General	37
9.3. Tipos de Programación Dinámica	37
9.3.1. Top-Down (Memoización)	37
9.3.2. Bottom-Up (Tabulación)	38
9.4. Ventajas y Desventajas	38
9.5. Cuándo Usar Programación Dinámica	38
9.6. Fibonacci	38
9.6.1. ¿Se puede hacer mejor?	41
9.7. Problema de la mochila 0/1	41
9.8. Cambio de moneda	43
9.9. Distancia de edición	43
9.10. Más problemas clásicos	44
10. Otros: Greedy(Ávidos), Backtracking(Vuelta atrás)	45
10.1. Greedy	45
10.1.1. Ejemplos de problemas No Aptos para Greedy	47
10.2. Backtracking	47
10.3. Características	47
10.4. Cuándo usar backtracking	49
11. Diferencias con Backtracking	50
12. Estructura General	50
13. Análisis de Branch and Bound	50
13.1. Pasos para el Análisis	50

14. Ejemplos Clásicos	51
14.1. Problema del Viajante (TSP)	51
14.1.1. Análisis del Problema	51
14.1.2. Ejemplo Numérico	51
14.1.3. Implementación de Cotas	54
14.1.4. Optimizaciones	54
14.2. Problema de la Mochila 0/1	55
14.2.1. Análisis del Problema	55
15. Ventajas y Desventajas	55
16. Cuándo Usar Branch and Bound	55
17. Técnicas de Optimización	56
18. Cuestiones y problemas	57
V Repaso Matemáticas para Ciencias de la Computación	59
18.1. Potencias	59
18.2. Logaritmos	59
18.2.1. Identidades y propiedades	59
18.3. Piso(<i>floor</i>) y techo(<i>ceiling</i>)	59
18.3.1. Identidades y propiedades	60
18.4. Factorial y coeficiente Binomial	60
18.4.1. Identidades y propiedades	61
18.5. Sumatorias	61
18.5.1. Identidades y propiedades	61
18.6. Series	62
18.6.1. Matrices	63
18.7. Hanoi	64



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**.

PARTE

Referencias



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.

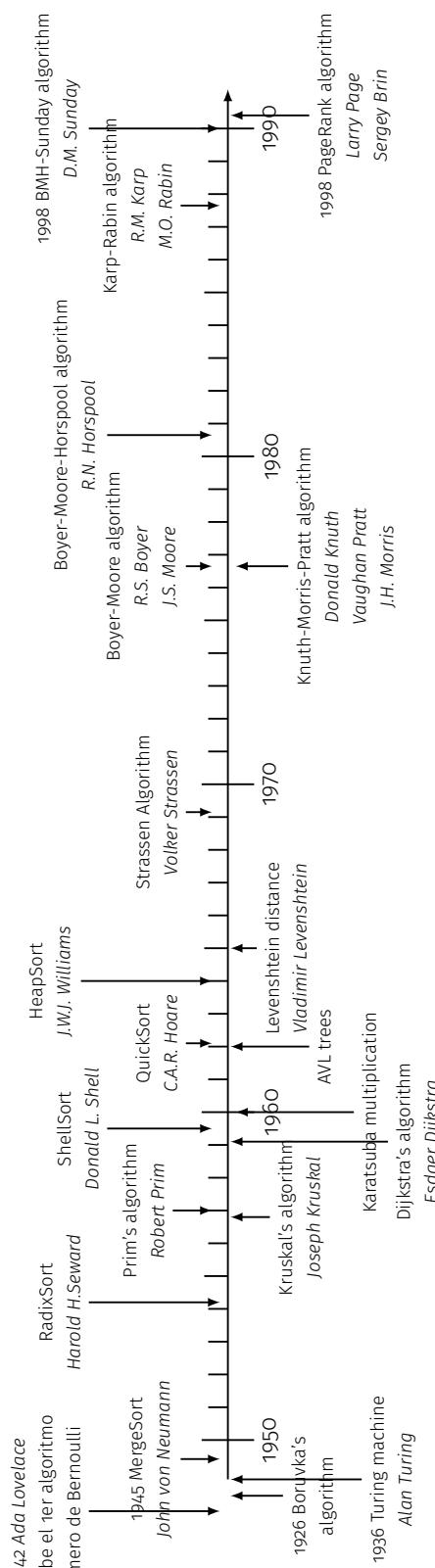
Usar un método de diseño para aplicar un algoritmo.

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.

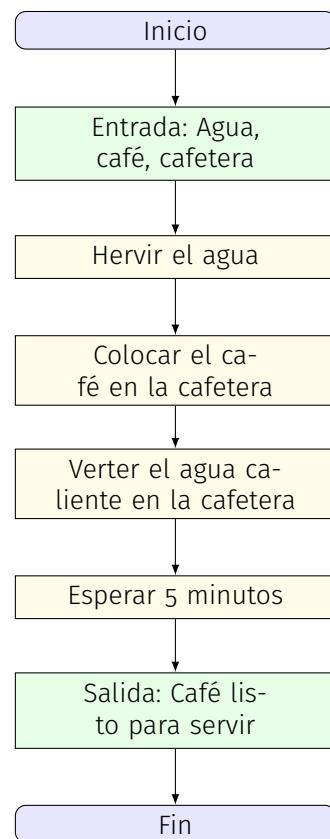


Figura 3. Descripción y diagrama de flujo para el proceso de preparar café.

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.



Figura 1. Donald Ervin Knuth

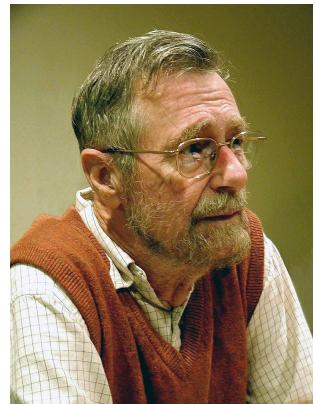


Figura 2. Edsger Wybe Dijkstra

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.

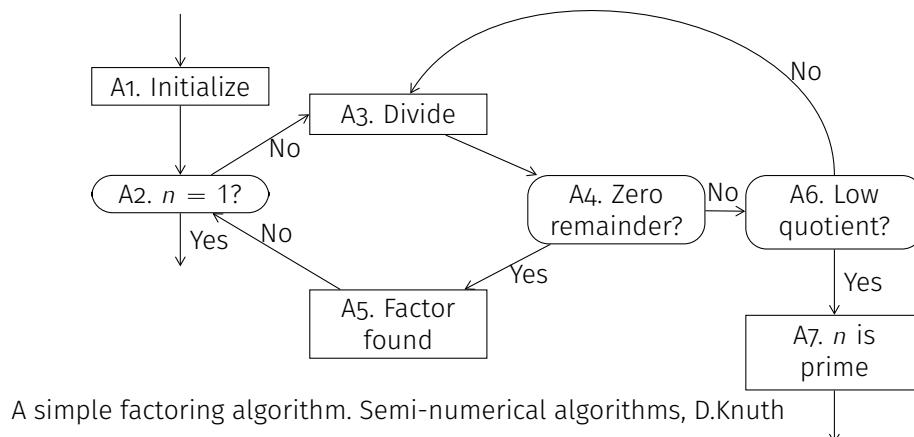


Figura 4. Un algoritmo simple de factorización. Algoritmos semi-numéricos, D. Knuth.

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.



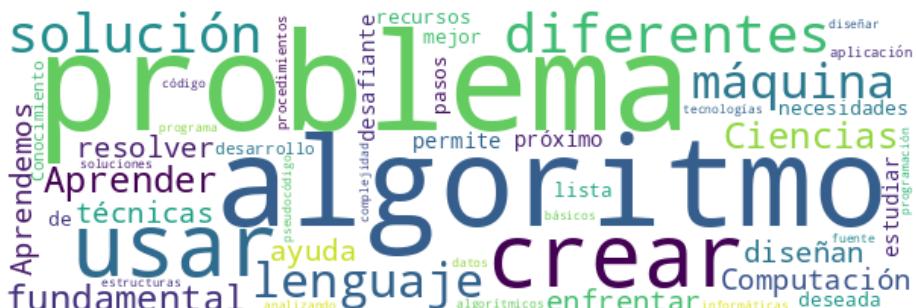
Figura 5. Línea del tiempo: evolución de los algoritmos.

¿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.

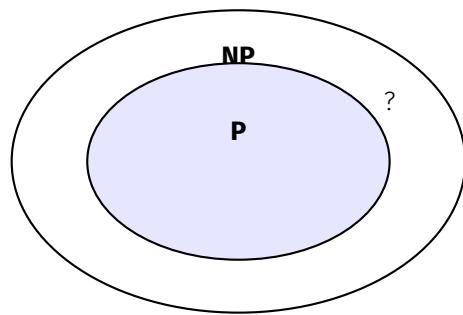


Figura 6. Relación entre las clases P y NP .

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);
}
```

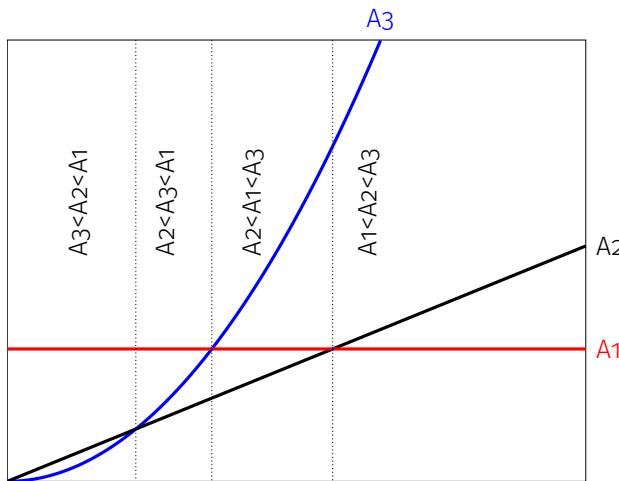
```
//A3.c
#include <stdio.h>

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:

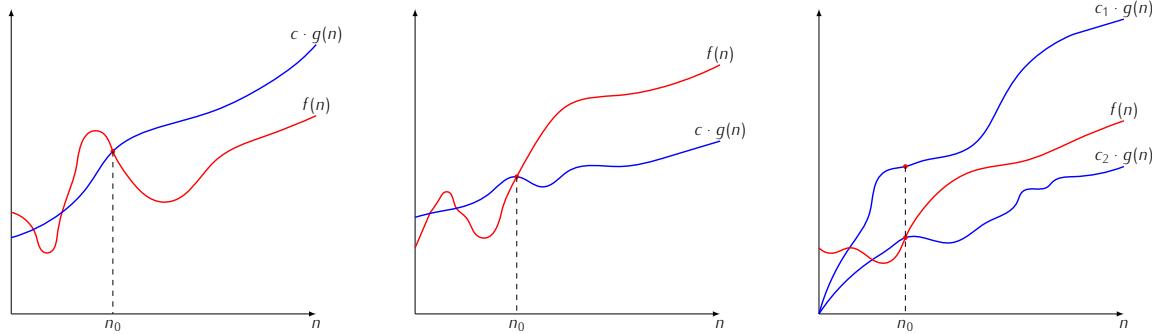
- Asignación de variables estructuradas (ej. vectores) cuyo número de elementos dependa del tamaño
- Ciclos cuyo número de iteraciones dependa del tamaño

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":



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

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

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

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

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

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

SUBSECCIÓN 3.1

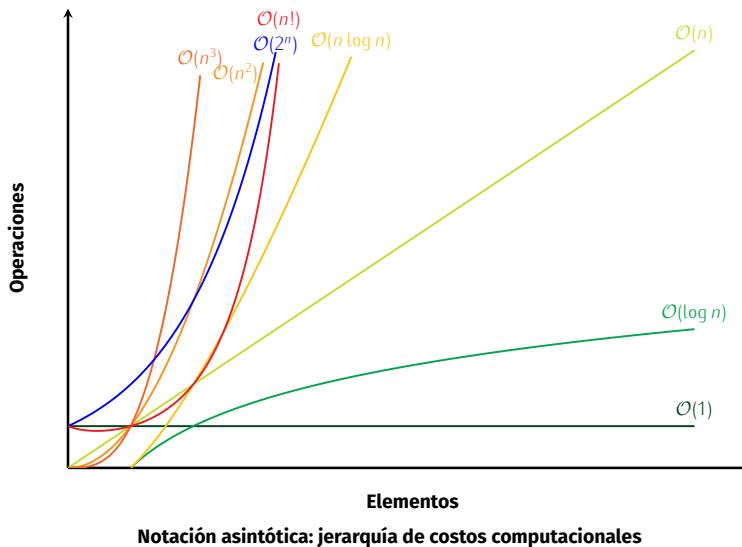
Simplificación

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

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

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

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



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.

```

1 void
2 InsertionSort( int A[ ], int N ){
3     int j, P;
4     int Tmp;
5
6     for( P = 1; P < N; P++ ){
7         Tmp = A[ P ];
8         for( j = P; j > 0 && A[ j - 1 ] > Tmp; j-- )
9             A[ j ] = A[ j - 1 ];
10            A[ j ] = Tmp;
11    }
12 }
```

	line	cost	times
6	c_1	n	
7	c_2	$n - 1$	
8	c_3	$\sum_{j=1}^n j$	
9	c_4	$\sum_{j=1}^n (j - 1)$	
10	c_5	$n - 1$	

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

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

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:

$$T(n) = c_1 \cdot n + 2 \cdot c_2 \cdot (n - 1)$$

$$T(n) = (c_1 + 2c_2)n - 2c_2$$

$$T(n) = a \cdot n + b \in \mathcal{O}(n)$$

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	$\sum_{j=1}^n \sum_{k=1}^j k$

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

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  $r$ 

```

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

$$t_{\max}(n) = n \in O(n)$$

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

```

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

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

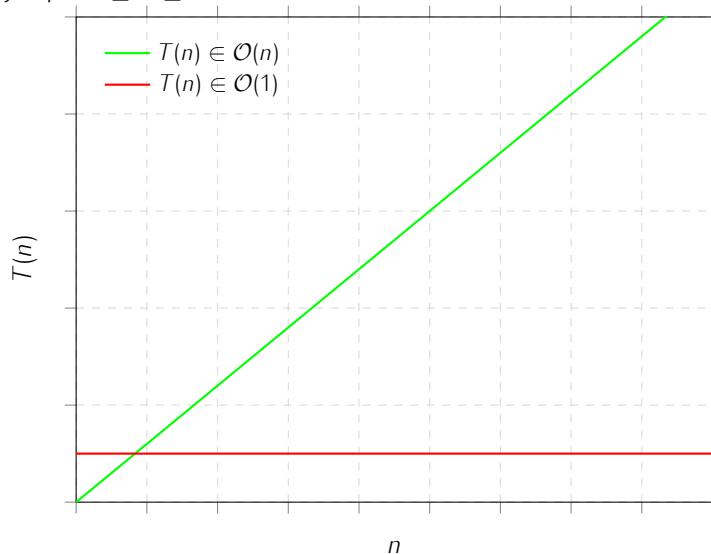
Hipótesis de equiprobabilidad:

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

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 }
```

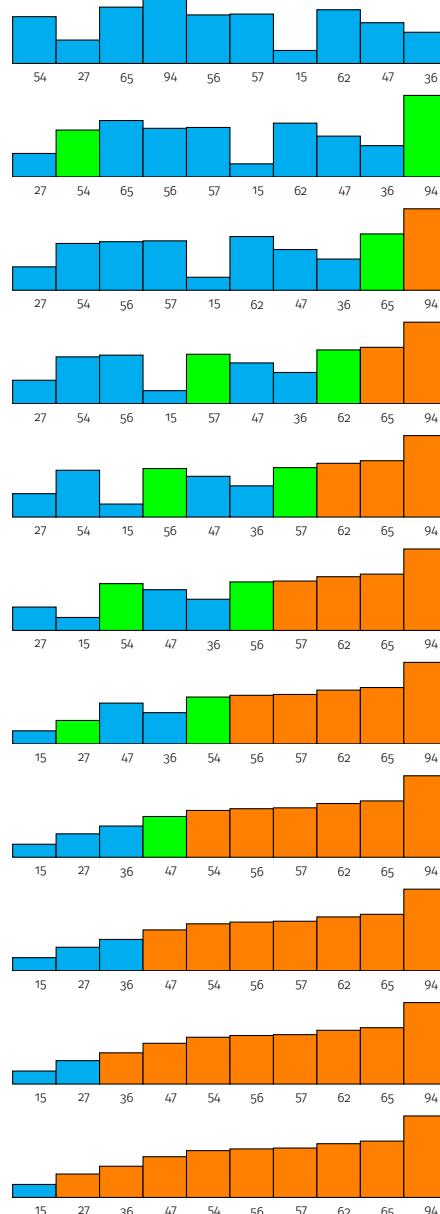
Algoritmo 4 Ordenación por intercambio

```

1: procedure SWAPSORT( $V$ )  $\triangleright V[0..n - 1]$ 
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:          $V[j] \leftrightarrow V[j - 1]$   $\triangleright \text{swap}(V[j], V[j - 1])$ 
```

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

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 }
```

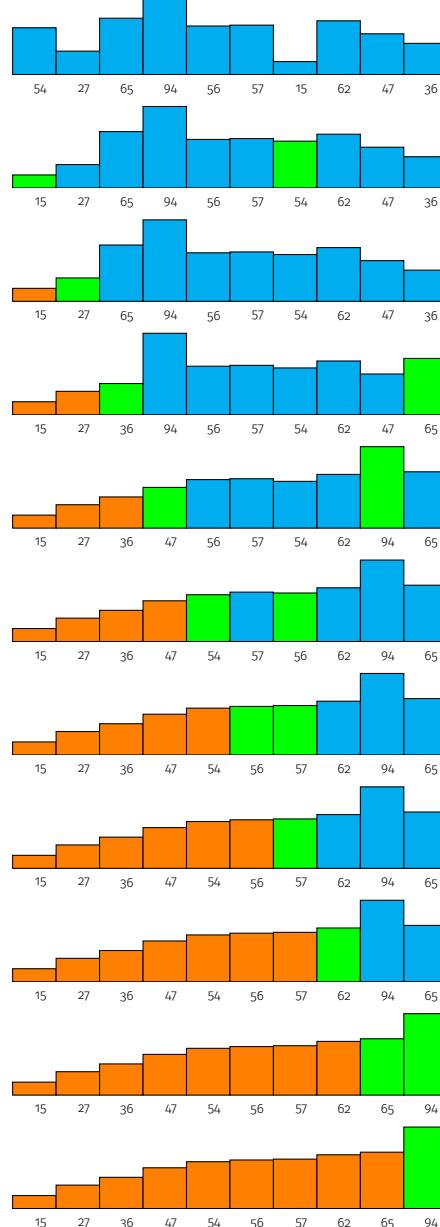
Algoritmo 5 Ordenación por Selección

```

1: procedure SELECTIONSORT( $V$ )  $\triangleright V[0..n - 1]$ 
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 esta 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    }
```

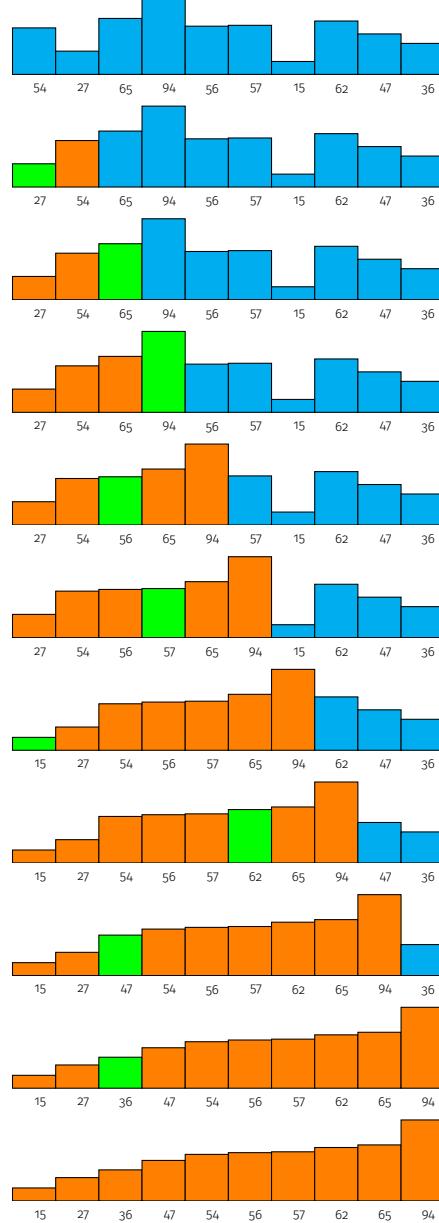
Algoritmo 6 Ordenación por Inserción

```

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:

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

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

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:

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

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

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:

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

Ecuación característica:

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

Entonces:

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

$$c_1 = 2, c_2 = -1/2 \text{ y } c_3 = -2$$

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:

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

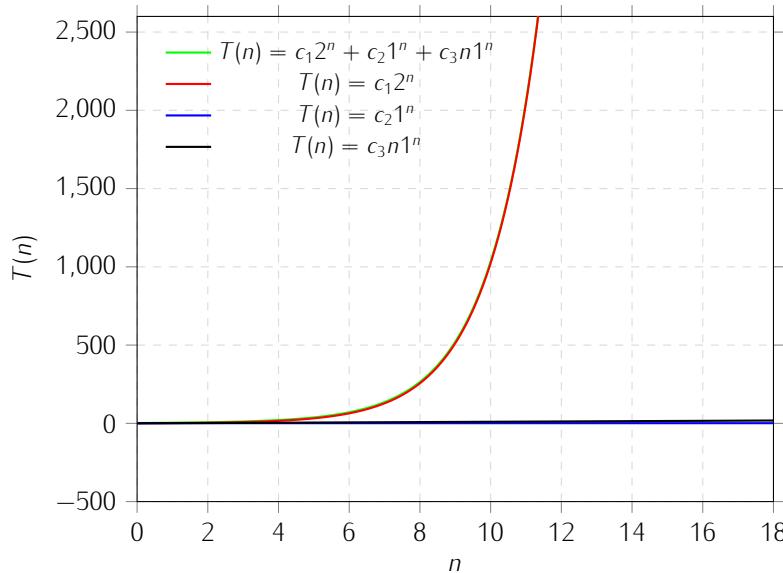
Ejemplo 3 |

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

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:

$$t_k = c_1(2^k)^2 + c_2 2^k$$

Deshaciendo los cambios:

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

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 $K' = \frac{k}{\frac{p}{q^r} - 1}$:

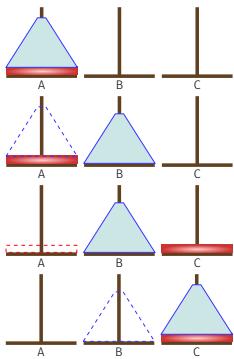
$$T(n) = K' n^r \left(\frac{n^{\log_q p}}{n^r} - 1 \right) + n^{\log_q p} = (K' + 1)n^{\log_q p} - K' n^r$$

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.

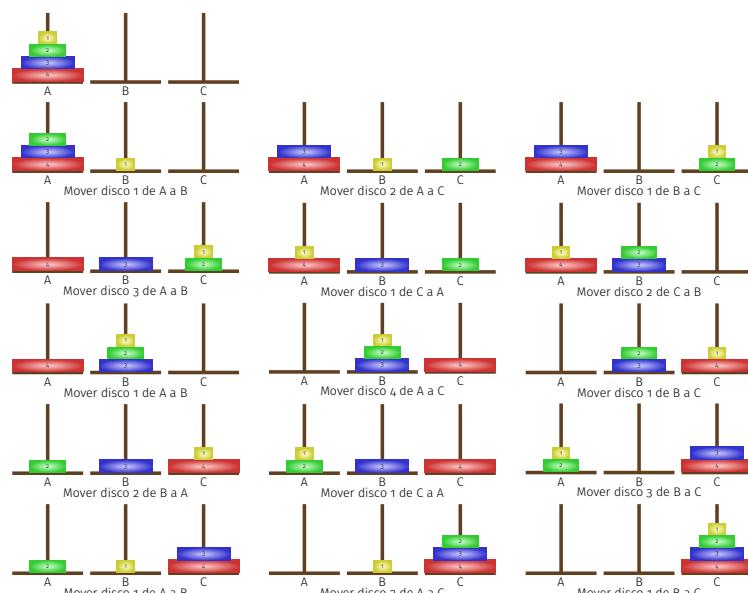
Secuencia Hanoi:

<i>n</i>	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:

$$\begin{aligned}T(n) &= 16 \cdot T(n/2) + 8 \\&= 16(16T(n/2^2) + 8) + 8 \\&= 16^2T(n/2^2) + 16 \cdot 8 + 8 \\&= 16^3T(n/2^3) + 16^2 \cdot 8 + 16 \cdot 8 + 8 \\&= 16^3T(n/2^3) + 16^2 \cdot 8 + 16^1 \cdot 8 + 16^0 \cdot 8 \\&= \dots \\&= 16^kT(n/2^k) + \dots + 16^2 \cdot 8 + 16 \cdot 8 + 16^0 \cdot 8 \\&= 16^kT(n/2^k) + 8 \sum_{i=0}^{k-1} 16^i \\&= 16^kT(n/2^k) + 8 \left(\frac{16^k}{15} - \frac{1}{15} \right)\end{aligned}$$

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

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

Ejemplo 7

```
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: $\lg 3 = 1.585$

Cuestiones y problemas

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( $a, n$ )  $\triangleright a^n$ 
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

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:      $\langle a, b \rangle \leftarrow \langle \min(c, e), \max(d, f) \rangle$ 
10:    return  $\langle a, b \rangle$ 

```

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:

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:      $\langle a, b \rangle \leftarrow \langle \min(c, e), \max(d, f) \rangle$ 
15:   return  $\langle a, b \rangle$ 

```

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

SUBSECCIÓN 8.11

Más problemas clásicos

- 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 soluciones 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:

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

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



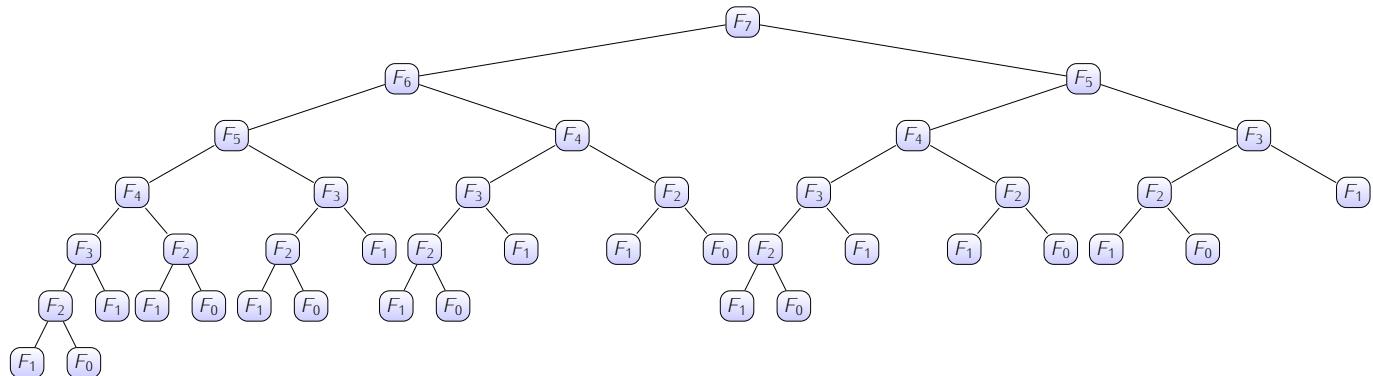
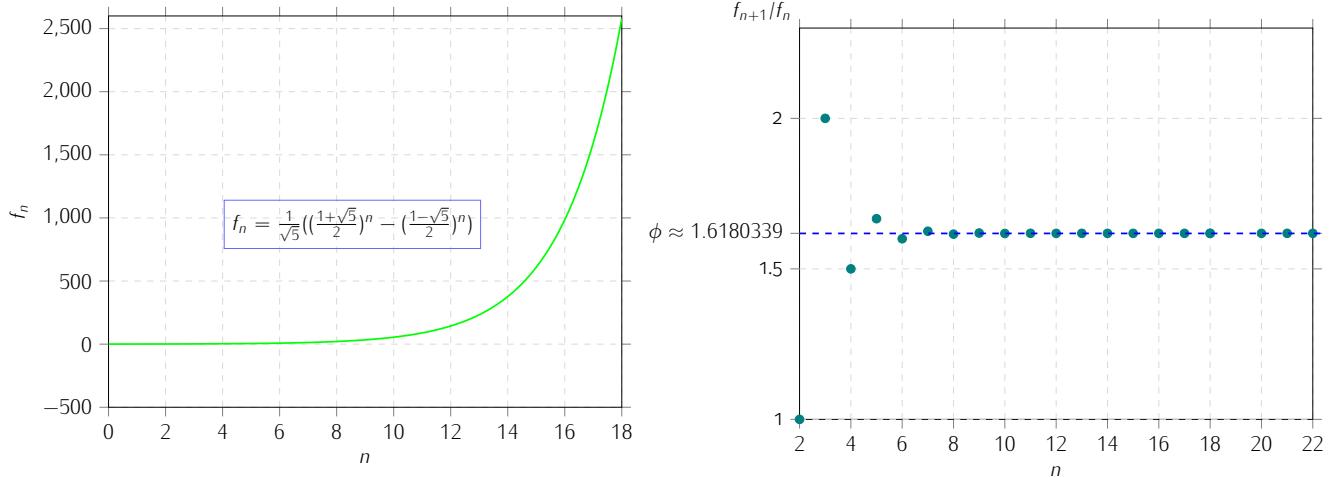
Figura 8. Leonardo de Pisa

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( $n - 2$ )

```



Y si usamos memorización (memoria intermedia).

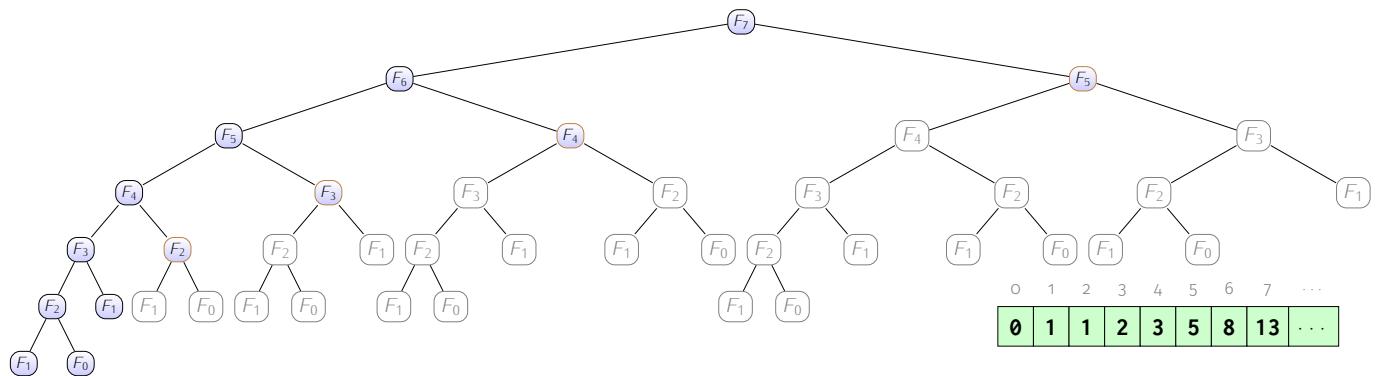
Algoritmo 16 Fibonacci Memorización

```

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:

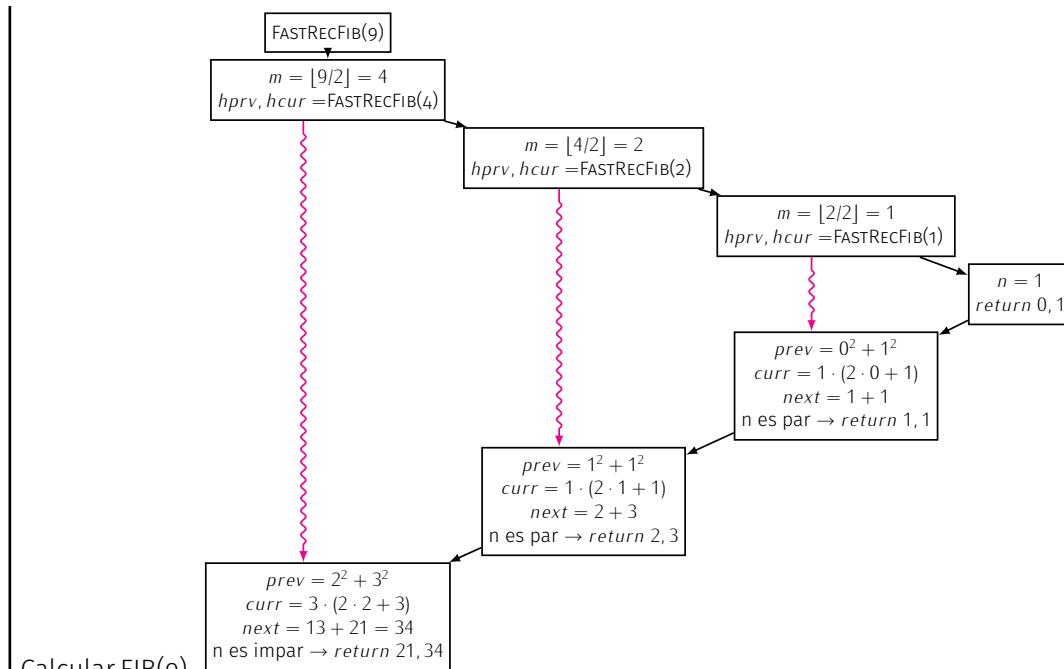
$$\begin{aligned} F_{2n-1} &= F_{n-1}^2 + F_n^2 \\ F_{2n} &= F_n(F_{n-1} + F_{n+1}) \\ &= F_n(2F_{n-1} + F_n) \end{aligned}$$

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:    $hprev, hcur \leftarrow$  FASTRECFIB( $m$ )
6:    $prev \leftarrow hprev^2 + hcur^2$ 
7:    $curr \leftarrow hcur \cdot (2 \cdot hprev + hcur)$ 
8:    $next \leftarrow prev + curr$ 
9:   if  $n$  es par then
10:    return  $prev, curr$ 
11:   else
12:    return  $curr, next$ 

```



Ejemplo 12

Calcular FIB(9)

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
\emptyset	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

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	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.

$$\text{cambio}(\{m_1, \dots, m_i\}, C) = \min \left\{ \begin{array}{l} \text{cambio}(\{m_1, \dots, m_{i-1}\}, C) \\ 1 + \text{cambio}(\{m_1, \dots, m_i\}, C - m_i) \end{array} \right.$$

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

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

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 $ED(i - 1, j - 1) + 1$



Figura 9. Stanisław Ulam

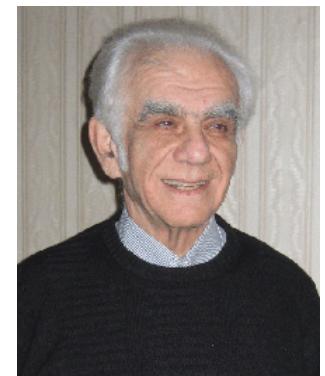
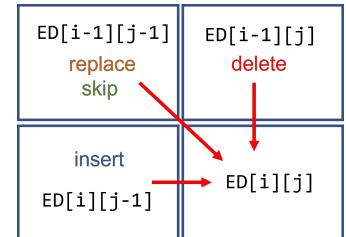


Figura 10. Vladimir I. Levenshtein

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  $ED[m, n]$ 

```

Ejemplo 15

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

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 cumplan 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
return Solucion

```



Figura 11. Otakar Borůvka

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.

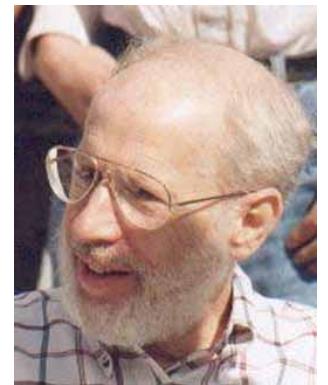
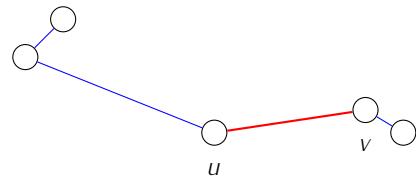
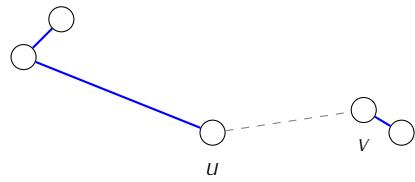
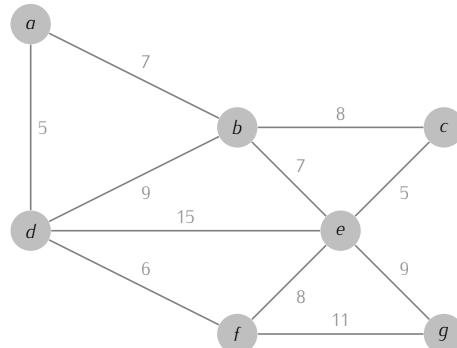


Figura 12. Joseph Kruskal

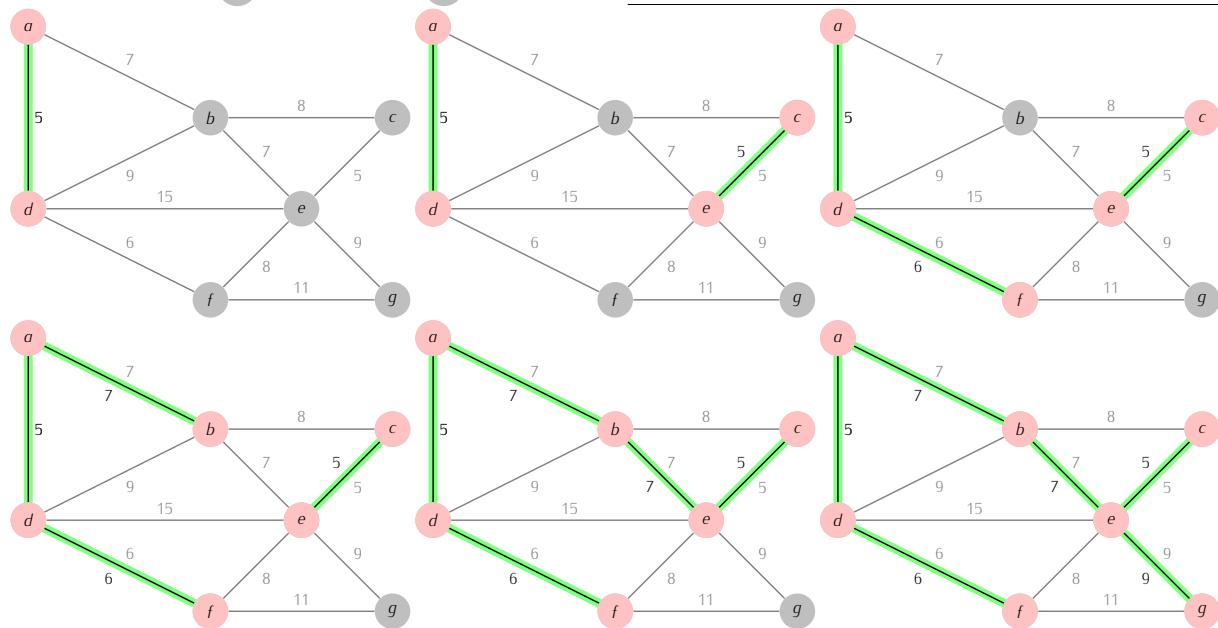
```
// A utility function to find the subset of an element i
int find(int parent[], int i)
{
    if (parent[i] == -1)
        return i;
    return find(parent, parent[i]);
}

// 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( $E, w$ )                                ▷ Ordenar el conjunto de arcos
3:    $T = (V, \emptyset)$ 
4:    $C \leftarrow V$                                 ▷  $u, v \in V, C$  componentes conexas
5:   while  $|C| > 1$  do
6:      $e \leftarrow \{u, v\}$                           ▷  $e$  es el siguiente arco en orden de costo creciente
7:     if  $\text{FIND}(u) \neq \text{FIND}(v)$  then
8:        $T \leftarrow T + e$ 
9:        $a \leftarrow \text{FIND}(u)$ 
10:       $b \leftarrow \text{FIND}(v)$ 
11:       $\text{UNION}(a, b)$ 
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.

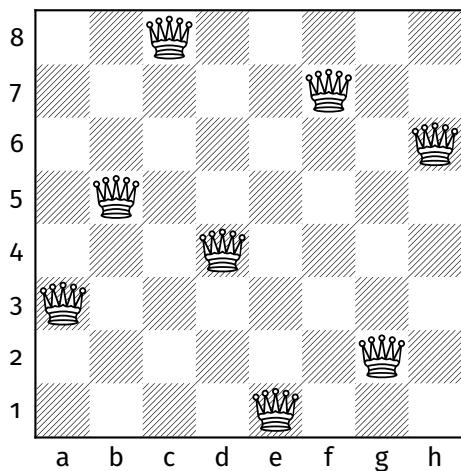
- Se puede entender como "opuesto" a avance rápido:
 - Avance rápido: añadir elementos a la solución y no deshacer ninguna decisión tomada.
 - Backtracking: añadir y quitar todos los elementos.
- Probar todas las combinaciones.

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:

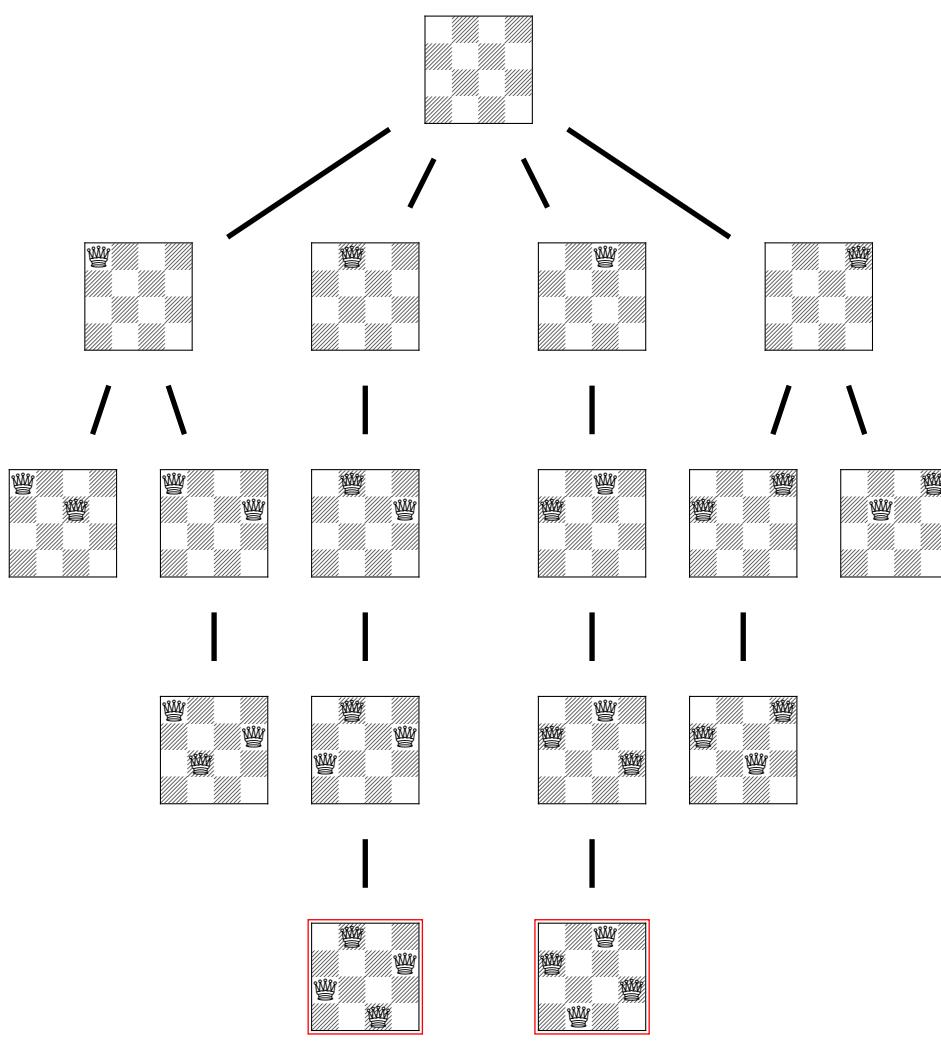
$$\binom{64}{8} = 4.426.165.368$$

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

Branch and Bound

Branch and Bound (Ramificación y Acotamiento) es una técnica algorítmica que combina la búsqueda sistemática con cotas para eliminar partes del espacio de búsqueda. A diferencia del backtracking, utiliza información sobre cotas para podar el árbol de búsqueda de manera más efectiva.

SECCIÓN 11

Diferencias con Backtracking

1. Uso de cotas:

- Branch and Bound usa cotas superiores e inferiores
- Backtracking solo usa restricciones de factibilidad

2. Orden de exploración:

- Branch and Bound puede usar diferentes estrategias (mejor primero, amplitud, etc.)
- Backtracking típicamente usa profundidad primero

3. Objetivo:

- Branch and Bound busca optimizar (maximizar o minimizar)
- Backtracking busca soluciones factibles

SECCIÓN 12

Estructura General

Algoritmo 24 Estructura General de un Algoritmo Branch and Bound

```

1: function BRANCHANDBOUND(estado_inicial)
2:   mejor_solución ← null
3:   mejor_valor ←  $\infty$ 
4:   cola ← {estado_inicial}
5:   while cola no está vacía do
6:     estado_actual ← seleccionar_estado(cola)
7:     cota_inferior ← calcular_cota_inferior(estado_actual)
8:     cota_superior ← calcular_cota_superior(estado_actual)
9:     if cota_inferior  $\geq$  mejor_valor then
10:       continuar
11:       if es_solución(estado_actual) then
12:         if valor(estado_actual) < mejor_valor then
13:           mejor_solución ← estado_actual
14:           mejor_valor ← valor(estado_actual)
15:           continuar
16:           nuevos_estados ← generar_hijos(estado_actual)
17:           cola ← cola  $\cup$  nuevos_estados
18:   return mejor_solución

```

SECCIÓN 13

Análisis de Branch and Bound

SUBSECCIÓN 13.1

Pasos para el Análisis

1. Definir el espacio de estados:

- Identificar las variables de decisión
- Establecer las restricciones
- Definir la función objetivo

2. Calcular cotas:

- Cota inferior: mejor solución factible conocida
- Cota superior: solución relajada del problema
- Justificar la validez de las cotas

3. Analizar la complejidad:

- Tamaño del espacio de estados
- Efectividad de las cotas
- Estrategia de selección de estados

SECCIÓN 14

Ejemplos Clásicos

SUBSECCIÓN 14.1

Problema del Viajante (TSP)

14.1.1. Análisis del Problema

1. Espacio de estados:

- Cada estado representa un camino parcial
- Variables: ciudades visitadas, costo actual
- Restricciones: visitar cada ciudad una vez

2. Cotas:

- Cota inferior: costo actual + MST de ciudades restantes
- Cota superior: solución heurística (ej: vecino más cercano)
- Las cotas son admisibles y consistentes

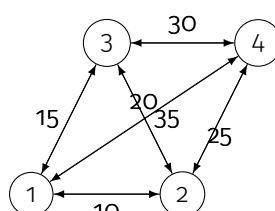
3. Complejidad:

- Sin cotas: $O(n!)$
- Con cotas: $O(2^n)$ en promedio
- Depende de la calidad de las cotas

14.1.2. Ejemplo Numérico

Consideremos un TSP con 4 ciudades y la siguiente matriz de distancias:

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 10 & 0 & 35 & 25 \\ 15 & 35 & 0 & 30 \\ 20 & 25 & 30 & 0 \end{bmatrix}$$



Paso 1: Inicialización

- Estado inicial: {1} (ciudad inicial)
- Cota superior inicial: 100 (solución heurística)
- Mejor solución: null

Paso 2: Generación del Árbol

1. Nivel 1:

- Estado: {1}
- Cota inferior: $10 + \text{MST}(2,3,4) = 10 + 55 = 75$
- Cota superior: 100
- Hijos: {1,2}, {1,3}, {1,4}

2. Nivel 2:

- Estado {1,2}: cota inferior = $10 + 25 + \text{MST}(3,4) = 10 + 25 + 30 = 65$
- Estado {1,3}: cota inferior = $15 + 30 + \text{MST}(2,4) = 15 + 30 + 25 = 70$
- Estado {1,4}: cota inferior = $20 + 25 + \text{MST}(2,3) = 20 + 25 + 35 = 80$

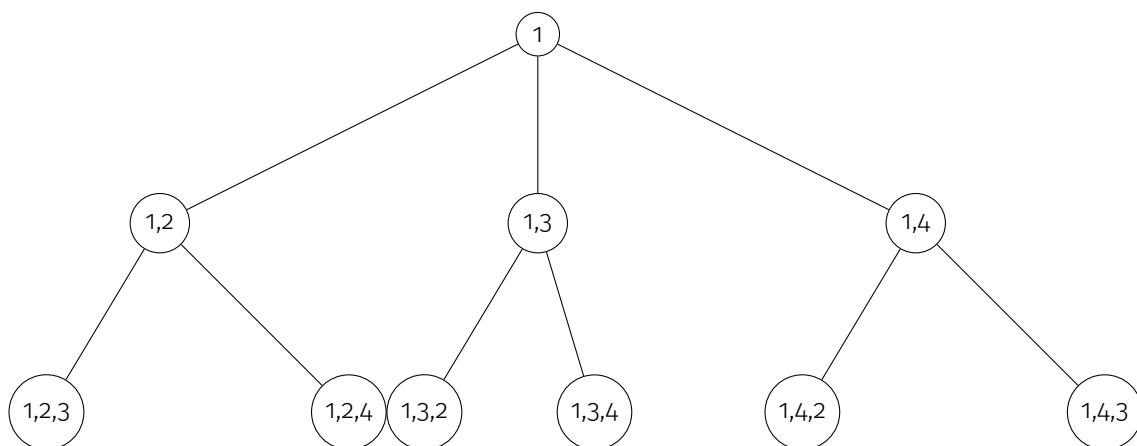
3. Nivel 3:

- Estado {1,2,3}: cota inferior = $10 + 25 + 30 + 20 = 85$
- Estado {1,2,4}: cota inferior = $10 + 25 + 25 + 15 = 75$
- Estado {1,3,2}: cota inferior = $15 + 30 + 25 + 20 = 90$
- Estado {1,3,4}: cota inferior = $15 + 30 + 25 + 10 = 80$
- Estado {1,4,2}: cota inferior = $20 + 25 + 25 + 15 = 85$
- Estado {1,4,3}: cota inferior = $20 + 25 + 30 + 10 = 85$

Paso 3: Poda y Solución

- Se podan los estados con cota inferior ≥ 100
- Se explora primero el estado {1,2} por tener menor cota inferior
- La solución óptima es: $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$
- Costo total: $10 + 25 + 25 + 15 = 75$

Representación del Árbol de Búsqueda



Cálculo Detallado de MST(2,3,4) Para calcular el MST(2,3,4), consideramos solo las ciudades 2, 3 y 4 con sus distancias:

$$\begin{bmatrix} 0 & 35 & 25 \\ 35 & 0 & 30 \\ 25 & 30 & 0 \end{bmatrix}$$

El algoritmo de Kruskal para MST(2,3,4):

1. Ordenar las aristas por peso:

- 2-4: 25
- 3-4: 30
- 2-3: 35

2. Construir el MST:

- Paso 1: Agregar arista 2-4 (peso 25)
- Paso 2: Agregar arista 3-4 (peso 30)
- Paso 3: No se agrega 2-3 porque formaría un ciclo

3. Total MST(2,3,4) = 25 + 30 = 55

Por lo tanto, la cota inferior para el estado {1} es:

- Costo actual: 10 (de 1 a 2)
- Más MST(2,3,4): 55
- Más costo mínimo para volver a 1: 10
- Total: 10 + 55 + 10 = 75

Esta cota inferior es válida porque:

- Cualquier tour debe visitar todas las ciudades
- El MST representa el costo mínimo para conectar las ciudades restantes
- Se debe volver a la ciudad inicial

Anotaciones del Árbol:

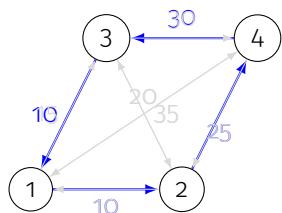
- **Nivel 0:** Estado inicial {1}
 - Cota inferior: 75
 - Cota superior: 100
- **Nivel 1:** Estados con dos ciudades
 - {1,2}: cota inferior = 65 (mejor)
 - {1,3}: cota inferior = 70
 - {1,4}: cota inferior = 80
- **Nivel 2:** Estados con tres ciudades
 - {1,2,3}: cota inferior = 85
 - {1,2,4}: cota inferior = 75
 - {1,3,2}: cota inferior = 90
 - {1,3,4}: cota inferior = 80
 - {1,4,2}: cota inferior = 85
 - {1,4,3}: cota inferior = 85

Orden de Exploración:

1. Se explora primero $\{1,2\}$ por tener la menor cota inferior (65)
2. Luego se exploran sus hijos $\{1,2,3\}$ y $\{1,2,4\}$
3. Se encuentra la solución óptima en $\{1,2,4\}$ con costo 75
4. Se podan los demás estados por tener cotas inferiores mayores

Solución Óptima: La solución óptima encontrada es el recorrido: $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$

- Costo total: 75
- Desglose del recorrido:
 - $1 \rightarrow 2$: 10
 - $2 \rightarrow 4$: 25
 - $4 \rightarrow 3$: 30
 - $3 \rightarrow 1$: 10
 - Total: $10 + 25 + 30 + 10 = 75$

**14.1.3. Implementación de Cotas**

1. **Cota Inferior:**
 - Costo actual del camino parcial
 - Más el costo del MST de las ciudades restantes
 - Más el costo mínimo para volver a la ciudad inicial
2. **Cota Superior:**
 - Algoritmo del vecino más cercano
 - Algoritmo de inserción más barata
 - Algoritmo de 2-opt

14.1.4. Optimizaciones

1. **Ordenamiento de Ciudades:**
 - Ordenar ciudades por distancia a la ciudad actual
 - Probar primero las ciudades más cercanas
 - Aumenta la probabilidad de encontrar buenas soluciones temprano
2. **Poda por Dominancia:**
 - Si dos estados tienen las mismas ciudades visitadas
 - Mantener solo el de menor costo actual

- Reduce el espacio de búsqueda

3. Memorización:

- Guardar estados ya explorados
- Evitar recalcular cotas
- Mejora el rendimiento en problemas pequeños

SUBSECCIÓN 14.2

Problema de la Mochila o/1

14.2.1. Análisis del Problema

1. Espacio de estados:

- Cada estado representa una selección parcial de items
- Variables: items seleccionados, peso actual, valor actual
- Restricciones: peso máximo

2. Cotas:

- Cota inferior: valor actual + valor de items restantes
- Cota superior: solución de la mochila fraccionaria
- Las cotas son admisibles

3. Complejidad:

- Sin cotas: $O(2^n)$
- Con cotas: $O(nW)$ en el mejor caso
- W es la capacidad de la mochila

SECCIÓN 15

Ventajas y Desventajas

■ Ventajas:

- Más eficiente que backtracking para problemas de optimización
- Garantiza encontrar la solución óptima
- Permite encontrar cotas de la solución óptima

■ Desventajas:

- Requiere calcular cotas en cada paso
- Necesita memoria para mantener la cola de estados
- La efectividad depende de la calidad de las cotas

SECCIÓN 16

Cuándo Usar Branch and Bound

1. Cuando se busca optimizar una función objetivo
2. Cuando se pueden calcular cotas efectivas
3. Cuando el problema tiene subestructura óptima
4. Cuando se necesita una solución exacta

SECCIÓN 17

Técnicas de Optimización

1. Selección de estados:

- Mejor primero: prioriza estados con mejor cota
- Amplitud: explora todos los estados del mismo nivel
- Profundidad: explora hasta encontrar una solución

2. Mejora de cotas:

- Usar heurísticas para cotas superiores
- Aplicar relajaciones del problema
- Mantener un registro de la mejor solución

3. Poda:

- Eliminar estados con cotas peores que la mejor solución
- Usar dominancia entre estados
- Aplicar restricciones adicionales

Problema del viajante de comercio

El problema del viajante de comercio (TSP, por sus siglas en inglés) es uno de los problemas más estudiados en la teoría de la computación y la optimización combinatoria. Se plantea de la siguiente manera: dado un conjunto de ciudades y las distancias entre cada par de ellas, el objetivo es encontrar el recorrido más corto que visite cada ciudad exactamente una vez y regrese a la ciudad de origen.

Clasificación del TSP

El TSP es un problema *NP*-completo. Esto significa que:

- **Verificación en Tiempo Polinómico:** Si se proporciona una solución (es decir, un recorrido específico), es posible verificar en tiempo polinómico si este recorrido es válido y si su longitud es menor o igual a un valor dado. Esto lo clasifica como un problema en *NP*.
- **Dificultad de Resolución:** No se conoce ningún algoritmo que pueda resolver todas las instancias del TSP en tiempo polinómico. La dificultad radica en que el número de posibles recorridos crece factorialmente con el número de ciudades, lo que hace que la búsqueda exhaustiva sea impracticable para grandes conjuntos de datos.
- **Reducción de Problemas:** El TSP es *NP*-completo porque cualquier problema en *NP* puede ser transformado (reducido) en tiempo polinómico a una instancia del TSP. Esto implica que resolver el TSP eficientemente permitiría resolver cualquier problema *NP* eficientemente.

Importancia del TSP

El TSP no solo es un problema teórico interesante, sino que también tiene aplicaciones prácticas en áreas como la logística, la planificación de rutas, y la fabricación de circuitos impresos. Debido a su complejidad, se han desarrollado numerosos algoritmos aproximados y heurísticos para encontrar soluciones cercanas al óptimo en un tiempo razonable.

SECCIÓN 18

Cuestiones y problemas

18.1 Indique el comportamiento del algoritmo cuya ecuación de recurrencia es:

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

18.2 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?

- 18.3** 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.
- 18.4** Determine el algoritmo para multiplicar enteros grandes. Encontrar su coste. Este es el algoritmo de Karatsuba y Ofmann.
- 18.5** 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.
- 18.6** Hacer un algoritmo para encontrar un subvector de longitud m cuya suma sea máxima dentro de un vector de longitud n , con $m \leq n$.
- 18.7** Usando DyV, encontrar la suma de todos los numeros en un arreglo $A[1 \dots n]$ de enteros. Indique el costo temporal. Justifique.
- 18.8** 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.
- 18.9** 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 .
- 18.10** 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.
- 18.11** 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.
- 18.12** 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.
- 18.13** 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.
- 18.14** 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 algoritmos para computar el camino mas largo en G desde s a t . Indique la complejidad de su algoritmo.

Repaso Matemáticas para Ciencias de la Computación

SUBSECCIÓN 18.1

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 18.2

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.

18.2.1. 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 18.3

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:

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

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

18.3.1. Identidades y propiedades

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

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

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

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

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

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

SUBSECCIÓN 18.4

Factorial y coeficiente Binomial

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

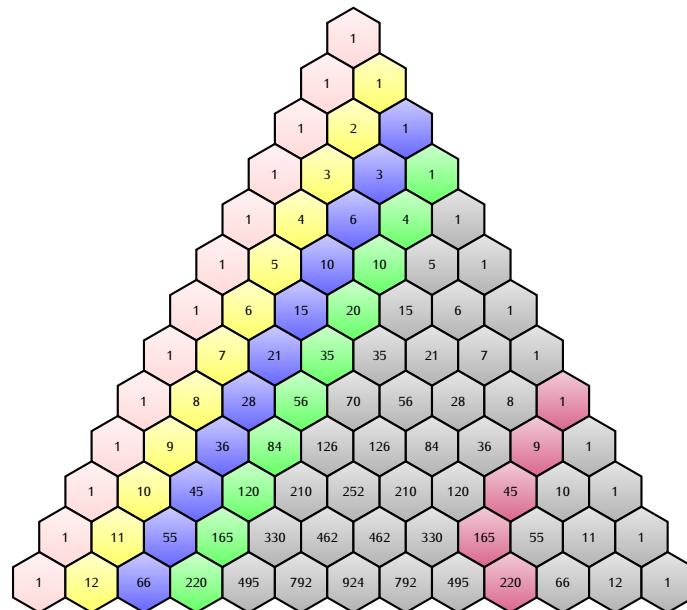
$$30! = 2652528598121910586363084800000000.$$

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

$$30! \approx 264517095922964306151924784891709$$

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

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



18.4.1. Identidades y propiedades

$$\begin{aligned}
 \binom{n}{k} &= \binom{n}{n-k} & \binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{n} &= 2^n \\
 \binom{n}{n} &= \binom{n}{0} = 1 & 1 + 2 + \cdots + n &= \binom{1}{1} + \binom{2}{1} + \cdots + \binom{n}{1} = \binom{n+1}{2} \\
 \binom{n}{k} &= \binom{n-1}{k} + \binom{n-1}{k-1} & \sum_{j \text{ par}} \binom{n}{j} &= \sum_{j \text{ impar}} \binom{n}{j} \\
 \binom{n}{k} &= \frac{n}{k} \binom{n-1}{n-k} & \sum_{k \leq n} \binom{r+k}{k} &= \binom{r+n+1}{n} \\
 (1+x)^n &= \sum_{j=0}^n \binom{n}{j} x^j & \sum_{m \leq k \leq n} \binom{k}{m} &= \binom{n+1}{m+1} \\
 (x+y)^n &= \sum_{k=0}^n \binom{n}{k} x^k y^{n-k}
 \end{aligned}$$

SUBSECCIÓN 18.5

Sumatorias

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

18.5.1. Identidades y propiedades

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

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

$$\begin{aligned}\sum_{j=1}^n j &= \frac{n(n+1)}{2} \\ \sum_{j=1}^n j^2 &= \frac{n(n+1)(2n+1)}{6} \\ \sum_{j=1}^n j^3 &= \frac{n^2(n+1)^2}{4} \\ \sum_{k=1}^n k(k+1) &= \frac{n(n+1)(n+2)}{3} \\ \sum_{k=1}^n \frac{1}{k(k+1)} &= \frac{n}{n+1} \\ \sum_{j=1}^n c^j &= \frac{c^{n+1}-1}{c-1}, c \neq 1\end{aligned}$$

$$\begin{aligned}\sum_{k=1}^n k(k+1)(k+2) &= \frac{n(n+1)(n+2)(n+3)}{4} \\ \sum_{k=1}^n \frac{1}{k(k+1)(k+2)} &= \frac{n(n+3)}{4(n+1)(n+2)} \\ \sum_{k=1}^n (2k-1) &= n^2 \\ \sum_{j=0}^{\infty} c^j &= \frac{1}{1-c}, |c| < 1 \\ \sum_{j=0}^n jc^j &= \sum_{j=1}^n jc^j = \frac{nc^{n+2} - nc^{n+1} - c^{n+1} + c}{(c-1)^2}, c \neq 1 \\ \sum_{j=0}^{\infty} jc^j &= \frac{c}{(1-c)^2}, |c| < 1 \\ \frac{\log(n+1)}{\log e} &\leq \sum_{j=1}^n \frac{1}{j} \leq \frac{\log n}{\log e} + 1 \\ H_n &= \sum_{j=1}^n \frac{1}{j}\end{aligned}$$

SUBSECCIÓN 18.6

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 $a_i = ar^{i-1}$:

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

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

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

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

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

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

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

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

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

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

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

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

18.6.1. Matrices

Determinante: Regla de Sarrus,

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

$$= a_{11}a_{22}a_{33} + a_{21}a_{32}a_{13} + a_{31}a_{12}a_{23} - a_{13}a_{22}a_{31} - a_{23}a_{32}a_{11} - a_{33}a_{12}a_{21}$$

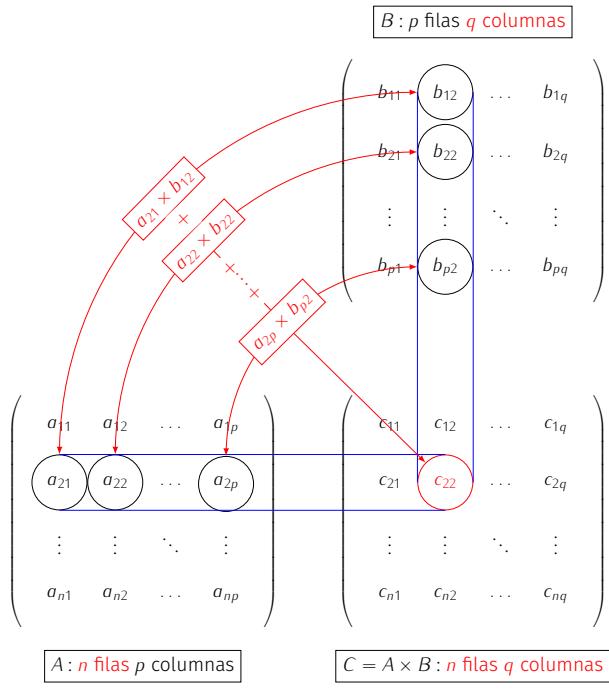
Dadas las matrices de la misma dimensión $A = (a_{ij})$ y $B = (b_{ij})$, entonces:

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

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

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

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



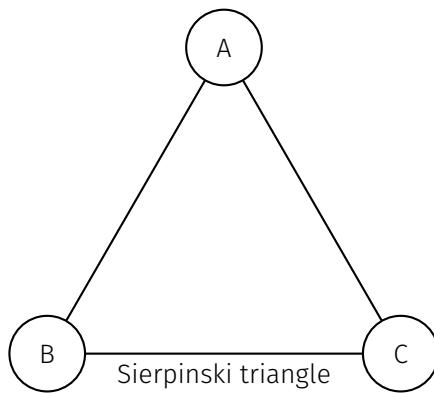
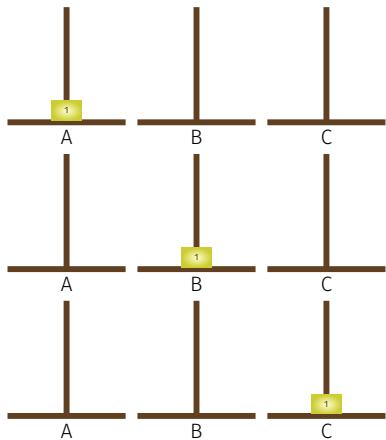
Transpuesta:

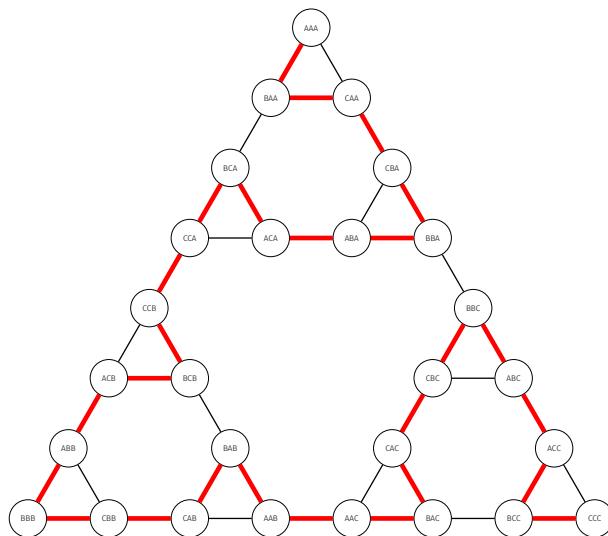
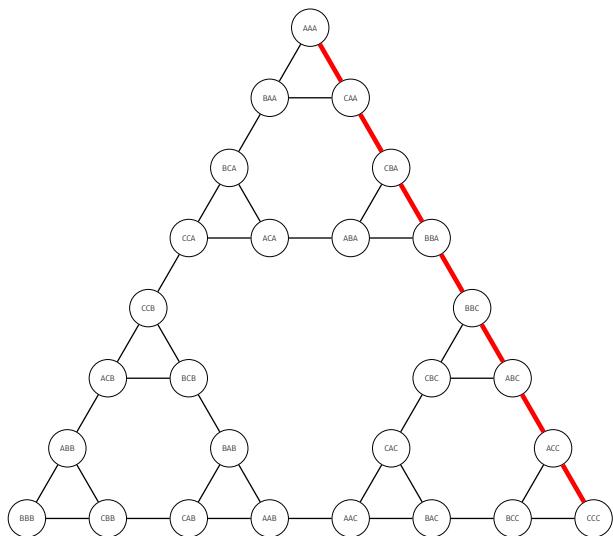
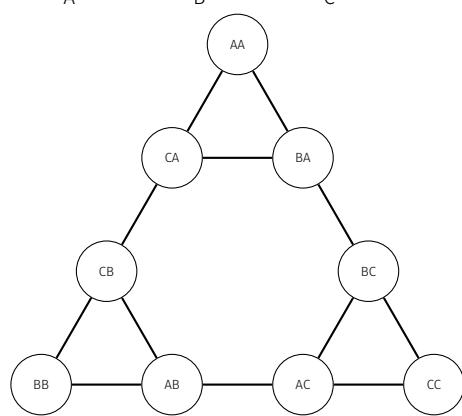
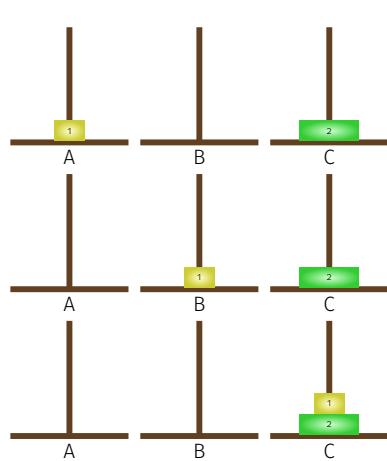
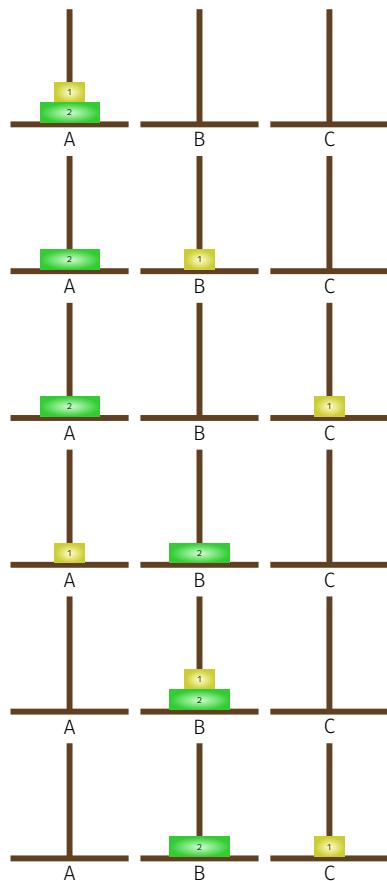
$$(A^T)_{ij} = A_{ji}, 1 \leq i \leq n, 1 \leq j \leq m$$

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

αA	$v N$
βB	$\xi \Xi$
$\gamma \Gamma$	$o O$
$\delta \Delta$	$\pi \Pi$
$\epsilon \varepsilon E$	$\rho \varrho P$
ζZ	$\sigma \Sigma$
ηH	τT
$\theta \vartheta \Theta$	$v Y$
$l l$	$\phi \varphi \Phi$
κK	$\chi \chi$
$\lambda \Lambda$	$\psi \Psi$
μM	$\omega \Omega$

SUBSECCIÓN 18.7

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.