

Algoritmos y Estructuras de Datos III

Primer cuatrimestre 2018

Algoritmos - Complejidad

Técnicas de diseño de algoritmos

Programa

1. Algoritmos:

- ▶ Definición de algoritmo. Máquina RAM. Complejidad. Algoritmos de tiempo polinomial y no polinomial. Límite inferior.
- ▶ Técnicas de diseño de algoritmos: *divide and conquer*, *backtracking*, algoritmos golosos, programación dinámica.
- ▶ Algoritmos aproximados y algoritmos heurísticos.

Programa

2. Grafos:

- ▶ Definiciones básicas. Adyacencia, grado de un nodo, isomorfismos, caminos, conexión, etc.
- ▶ Grafos eulerianos y hamiltonianos.
- ▶ Grafos bipartitos.
- ▶ Árboles: caracterización, árboles orientados, árbol generador.
- ▶ Planaridad. Coloreo. Número cromático.
- ▶ Matching, conjunto independiente, recubrimiento. Recubrimiento de aristas y vértices.

3. Algoritmos en grafos y aplicaciones:

- ▶ Representación de un grafo en la computadora: matrices de incidencia y adyacencia, listas.
- ▶ Algoritmos de búsqueda en grafos: BFS, DFS, A*.
- ▶ Mínimo árbol generador, algoritmos de Prim y Kruskal.
- ▶ Algoritmos para encontrar el camino mínimo en un grafo: Dijkstra, Ford, Floyd, Dantzig.
- ▶ Planificación de procesos: PERT/CPM.
- ▶ Algoritmos para determinar si un grafo es planar. Algoritmos para coloreo de grafos.
- ▶ Algoritmos para encontrar el flujo máximo en una red: Ford y Fulkerson.
- ▶ Matching: algoritmos para correspondencias máximas en grafos bipartitos. Otras aplicaciones.

4. Complejidad computacional:

- ▶ Problemas tratables e intratables. Problemas de decisión. P y NP. Máquinas de Turing no determinísticas. Problemas NP-completos. Relación entre P y NP.
- ▶ Problemas de grafos NP-completos: coloreo de grafos, grafos hamiltonianos, recubrimiento mínimo de las aristas, corte máximo, etc.

Bibliografía

1. G. Brassard and P. Bratley, *Fundamental of Algorithmics*, Prentice-Hall, 1996.
2. F. Harary, *Graph theory*, Addison-Wesley, 1969.
3. J. Gross and J. Yellen, *Graph theory and its applications*, CRC Press, 1999.
4. R. Ahuja, T. Magnanti and J. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall, 1993.
5. M. Garey and D. Johnson, *Computers and intractability: a guide to the theory of NP- Completeness*, W. Freeman and Co., 1979.

Algoritmos

- ▶ ¿Qué es un algoritmo?
- ▶ ¿Qué es un buen algoritmo?
- ▶ Dados dos algoritmos para resolver un mismo problema, ¿cuál es mejor?
- ▶ ¿Cuándo un problema está bien resuelto?

Algoritmo

- ▶ Secuencia de pasos que termina en un tiempo finito.
- ▶ Deben estar formulados en términos de pasos sencillos que sean:
 - ▶ precisos: se indica el orden de ejecución de cada paso
 - ▶ bien definidos: en toda ejecución del algoritmo se debe obtener el mismo resultado bajo los mismo parámetros
 - ▶ finitos: el algoritmo tiene que tener un número determinado de pasos
- ▶ Los describiremos mediante pseudocódigo.

Pseudocódigo

Encontrar el máximo de un vector de enteros:

algoritmo *maximo*(A, n)

entrada: un vector A con $n \geq 1$ de enteros

salida: el elemento máximo de A

$max \leftarrow A[0]$

para $i = 1$ **hasta** $n - 1$ **hacer**

si $A[i] > max$ **entonces**

$max \leftarrow A[i]$

fin si

fin para

retornar max

Análisis de algoritmos

En general, analizaremos los algoritmos utilizando como medida de eficiencia su tiempo de ejecución.

- ▶ Análisis empírico: implementarlos en una máquina determinada utilizando un lenguaje determinado, correlos para un conjunto de instancias y comparar sus tiempos de ejecución. Desventajas:
 - ▶ pérdida de tiempo y esfuerzo de programador
 - ▶ pérdida de tiempo de cómputo
 - ▶ conjunto de instancias acotado

Análisis de algoritmos

En general, analizaremos los algoritmos utilizando como medida de eficiencia su tiempo de ejecución.

- ▶ **Análisis empírico:** implementarlos en una máquina determinada utilizando un lenguaje determinado, correlos para un conjunto de instancias y comparar sus tiempos de ejecución. Desventajas:
 - ▶ pérdida de tiempo y esfuerzo de programador
 - ▶ pérdida de tiempo de cómputo
 - ▶ conjunto de instancias acotado
- ▶ **Análisis teórico:** determinar matemáticamente la cantidad de tiempo que llevará su ejecución como una función de la medida de la instancia considerada, independizándonos de la máquina sobre la cuál es implementado el algoritmo y el lenguaje para hacerlo. Para esto necesitamos definir:
 - ▶ un modelo de cómputo
 - ▶ un lenguaje sobre este modelo
 - ▶ instancias relevantes
 - ▶ tamaño de la instancia

Complejidad computacional

Definición informal: La *complejidad* de un algoritmo es una función que representa el tiempo de ejecución en función del tamaño de la entrada del algoritmo.

- ▶ Complejidad en el **peor caso**.
- ▶ Complejidad en el **caso promedio**.

Complejidad computacional

Definición informal: La *complejidad* de un algoritmo es una función que representa el tiempo de ejecución en función del tamaño de la entrada del algoritmo.

- ▶ Complejidad en el **peor caso**.
- ▶ Complejidad en el **caso promedio**.

Definición formal?

Modelo de cómputo: Máquina RAM

Definición: Máquina de registros + registro acumulador + direccionamiento indirecto.

Motivación: Modelar computadoras en las que la memoria es suficiente y donde los enteros involucrados en los cálculos entran en una palabra.

Modelo de cómputo: Máquina RAM

Definición: Máquina de registros + registro acumulador + direccionamiento indirecto.

Motivación: Modelar computadoras en las que la memoria es suficiente y donde los enteros involucrados en los cálculos entran en una palabra.

- ▶ **Unidad de entrada:** Sucesión de celdas numeradas, cada una con un entero de tamaño arbitrario.
- ▶ **Unidad de salida:** Sucesión de celdas.
- ▶ **Memoria:** Sucesión de celdas numeradas, cada una puede almacenar un entero de tamaño arbitrario.
- ▶ **Programa no almacenado** en memoria (aún así es una máquina programable!).

Modelo de cómputo: Máquina RAM

Suponemos que:

- ▶ Un programa es una secuencia de instrucciones que son ejecutadas secuencialmente, comenzando por la primera instrucción.
- ▶ Hay un contador de programa, que identifica la próxima instrucción a ser ejecutada.
- ▶ Hay tantas celdas de memoria (registros) como se necesiten.
- ▶ Se puede acceder de forma directa a cualquier celda.
- ▶ Los enteros entran en una celda de memoria.
- ▶ Hay un registro especial, llamado acumulador, donde se realizan los cálculos.

Máquina RAM - Instrucciones

- ▶ LOAD operando - Carga un valor en el acumulador
- ▶ STORE operando - Carga el acumulador en un registro
- ▶ ADD operando - Suma el operando al acumulador
- ▶ SUB operando - Resta el operando al acumulador
- ▶ MULT operando - Multiplica el operando por el acumulador
- ▶ DIV operando - Divide el acumulador por el operando
- ▶ READ operando - Lee un nuevo dato de entrada → operando
- ▶ WRITE operando - Escribe el operando a la salida
- ▶ JUMP label - Salto incondicional
- ▶ JGTZ label - Salta si el acumulador es positivo
- ▶ JZERO label - Salta si el acumulador es cero
- ▶ HALT - Termina el programa

Máquina RAM - Operandos

- ▶ **LOAD** = *a*: Carga en el acumulador el entero *a*.
- ▶ **LOAD** *i*: Carga en el acumulador el contenido del registro *i*.
- ▶ **LOAD** **i*: Carga en el acumulador el contenido del registro indexado por el valor del registro *i*.

Máquina RAM - Ejemplos

$a \leftarrow b$		LOAD	2
		STORE	1
$a \leftarrow b - 3$		LOAD	2
		SUB	=3
		STORE	1
mientras $x > 0$ hacer $x \leftarrow x - 2$ fin mientras	guarda	LOAD	1
		JGTZ	mientras
		JUMP	finmientras
	mientras	LOAD	1
		SUB	=2
		STORE	1
		JUMP	guarda
	finmientras	

Máquina RAM - Ejemplos

si $x \leq 0$ **entonces**

$y \leftarrow x + y$

sino

$y \leftarrow x$

fin si

guarda	LOAD	1
	JGTZ	sino
	LOAD	1
	ADD	2
	STORE	2
	JUMP	finsi
sino	LOAD	1
	STORE	2
finsi	...	

Programa para calcular k^k - Pseudocódigo

algoritmo *kalak*(k)

entrada: un entero k

salida: k^k si $k > 0$, 0 caso contrario

si $k \leq 0$ **entonces**

$x \leftarrow 0$

sino

$x \leftarrow k$

$y \leftarrow k - 1$

mientras $y > 0$ **hacer**

$x \leftarrow x \cdot k$

$y \leftarrow y - 1$

fin mientras

fin si

retornar x

Programa para calcular k^k - máquina RAM

	READ	1	carga en R1 la primera celda de la unidad de entrada
	LOAD	1	carga en el acumulador el valor de R1
	JGTZ	sino	si el valor del acumulador es > 0 <i>salta a</i> sino
	LOAD	= 0	carga 0 en el acumulador
	STORE	2	escribe el valor del acumulador en R2
	JUMP	fin	<i>salta a</i> fin
sino	LOAD	1	carga en el acumulador el valor de R1
	STORE	2	escribe el valor del acumulador en R2
	LOAD	1	carga en el acumulador el valor de R1
	SUB	= 1	resta 1 al valor del acumulador
	STORE	3	escribe el valor del acumulador en R3
guarda	LOAD	3	carga en el acumulador el valor de R3
	JGTZ	mientras	si el valor del acumulador es ≥ 0 <i>salta a</i> mientras
	JUMP	finmientras	<i>salta a</i> finmientras
mientras	LOAD	2	carga en el acumulador el valor de R2
	MULT	1	multiplica el valor del acumulador por el valor de R1
	STORE	2	escribe el valor del acumulador en R2
	LOAD	3	carga en el acumulador el valor de R3
	SUB	= 1	resta 1 al valor del acumulador
	STORE	3	escribe el valor del acumulador en R3
	JUMP	guarda	<i>salta a</i> guarda
finmientras fin	WRITE	2	escribe el valor de R2 en la unidad de salida
	HALT		para la ejecución

Complejidad en la Máquina RAM

- ▶ Asumimos que cada instrucción tiene un **tiempo de ejecución** asociado.

- ▶ **Tiempo de ejecución de un algoritmo A :**

$T_A(I)$ = suma de los tiempos de ejecución de las instrucciones realizadas por el algoritmo con la *instancia* I .

- ▶ **Complejidad de un algoritmo A :**

$$f_A(n) = \max_{I: |I|=n} T(I)$$

Complejidad en la Máquina RAM

- ▶ Asumimos que cada instrucción tiene un **tiempo de ejecución** asociado.
- ▶ **Tiempo de ejecución de un algoritmo A :**

$T_A(I)$ = suma de los tiempos de ejecución de las instrucciones realizadas por el algoritmo con la *instancia* I .

- ▶ **Complejidad de un algoritmo A :**

$f_A(n) = \max_{I: |I|=n} T(I)$ (pero debemos definir $|I|!$).

Operaciones básicas: tiempo de ejecución

- ▶ **Modelo uniforme:** Cada operación básica tiene un tiempo de ejecución constante.
 - ▶ Apropiado cuando los operandos entran en una palabra.
- ▶ **Modelo logarítmico:** El tiempo de ejecución de cada operación es una función del tamaño de los operandos.
 - ▶ Apropiado cuando los operandos pueden crecer arbitrariamente.

Tamaño de una instancia

Definición: Dada una instancia I , se define $|I|$ como el número de símbolos de un alfabeto finito necesarios para codificar I .

- ▶ Depende del **alfabeto** y de la **base**.
- ▶ Para almacenar $n \in \mathbb{N}$, se necesitan $L(n) = \lfloor \log_2(n) \rfloor + 1$ dígitos binarios.
- ▶ Para almacenar una lista de m enteros, se necesitan $L(m) + mL(N)$ dígitos binarios, donde N es el valor máximo de la lista (notar que se puede mejorar!).

Tamaño de una instancia

- ▶ Depende del problema que se esté analizando.
- ▶ En general, para problemas de ordenamiento, problemas sobre grafos, etc., utilizaremos como tamaño de la entrada la cantidad de elementos de la instancia de entrada. Esto modela de forma suficientemente precisa la realidad y facilita el análisis de los algoritmos.
- ▶ Para problemas sobre números, como cálculo del factorial, es más apropiado utilizar como tamaño de la entrada la cantidad de bits necesarios para representar la instancia de entrada en notación binaria.

Notación O

Dadas dos funciones $f, g : \mathbb{N} \rightarrow \mathbb{R}$, decimos que:

- ▶ $f(n) = O(g(n))$ si existen $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ tales que

$$f(n) \leq c g(n) \text{ para todo } n \geq n_0.$$

- ▶ $f(n) = \Omega(g(n))$ si existen $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ tales que

$$f(n) \geq c g(n) \text{ para todo } n \geq n_0.$$

- ▶ $f(n) = \Theta(g(n))$ si $f = O(g(n))$ y $f = \Omega(g(n))$.

Ejemplos

- ▶ Búsqueda secuencial: $O(n)$.
- ▶ Búsqueda binaria: $O(\log(n))$.

Ejemplos

- ▶ Búsqueda secuencial: $O(n)$.
- ▶ Búsqueda binaria: $O(\log(n))$.
- ▶ Ordenar un arreglo (bubblesort): $O(n^2)$.
- ▶ Ordenar un arreglo (quicksort): $O(n^2)$ en el peor caso (!).
- ▶ Ordenar un arreglo (heapsort): $O(n \log(n))$.

Ejemplos

- ▶ Búsqueda secuencial: $O(n)$.
- ▶ Búsqueda binaria: $O(\log(n))$.
- ▶ Ordenar un arreglo (bubblesort): $O(n^2)$.
- ▶ Ordenar un arreglo (quicksort): $O(n^2)$ en el peor caso (!).
- ▶ Ordenar un arreglo (heapsort): $O(n \log(n))$.

Es interesante notar que $O(n \log(n))$ es la complejidad **óptima** para algoritmos de ordenamiento basados en comparaciones.

Problemas bien resueltos

Definición: Decimos que un problema está *bien resuelto* si existe un algoritmo de complejidad polinomial para el problema.

Problemas bien resueltos

Definición: Decimos que un problema está *bien resuelto* si existe un algoritmo de complejidad polinomial para el problema.

	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
--	----------	----------	----------	----------	----------

Problemas bien resueltos

Definición: Decimos que un problema está *bien resuelto* si existe un algoritmo de complejidad polinomial para el problema.

	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
$O(n)$	0.01 ms	0.02 ms	0.03 ms	0.04 ms	0.05 ms

Problemas bien resueltos

Definición: Decimos que un problema está *bien resuelto* si existe un algoritmo de complejidad polinomial para el problema.

	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
$O(n)$	0.01 ms	0.02 ms	0.03 ms	0.04 ms	0.05 ms
$O(n^2)$	0.10 ms	0.40 ms	0.90 ms	0.16 ms	0.25 ms

Problemas bien resueltos

Definición: Decimos que un problema está *bien resuelto* si existe un algoritmo de complejidad polinomial para el problema.

	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
$O(n)$	0.01 ms	0.02 ms	0.03 ms	0.04 ms	0.05 ms
$O(n^2)$	0.10 ms	0.40 ms	0.90 ms	0.16 ms	0.25 ms
$O(n^3)$	1.00 ms	8.00 ms	2.70 ms	6.40 ms	0.12 sg

Problemas bien resueltos

Definición: Decimos que un problema está *bien resuelto* si existe un algoritmo de complejidad polinomial para el problema.

	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
$O(n)$	0.01 ms	0.02 ms	0.03 ms	0.04 ms	0.05 ms
$O(n^2)$	0.10 ms	0.40 ms	0.90 ms	0.16 ms	0.25 ms
$O(n^3)$	1.00 ms	8.00 ms	2.70 ms	6.40 ms	0.12 sg
$O(n^5)$	0.10 sg	3.20 sg	24.30 sg	1.70 min	5.20 min

Problemas bien resueltos

Definición: Decimos que un problema está *bien resuelto* si existe un algoritmo de complejidad polinomial para el problema.

	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
$O(n)$	0.01 ms	0.02 ms	0.03 ms	0.04 ms	0.05 ms
$O(n^2)$	0.10 ms	0.40 ms	0.90 ms	0.16 ms	0.25 ms
$O(n^3)$	1.00 ms	8.00 ms	2.70 ms	6.40 ms	0.12 sg
$O(n^5)$	0.10 sg	3.20 sg	24.30 sg	1.70 min	5.20 min
$O(2^n)$	1.00 ms	1.00 sg	17.90 min	12 días	35 años

Problemas bien resueltos

Definición: Decimos que un problema está *bien resuelto* si existe un algoritmo de complejidad polinomial para el problema.

	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
$O(n)$	0.01 ms	0.02 ms	0.03 ms	0.04 ms	0.05 ms
$O(n^2)$	0.10 ms	0.40 ms	0.90 ms	0.16 ms	0.25 ms
$O(n^3)$	1.00 ms	8.00 ms	2.70 ms	6.40 ms	0.12 sg
$O(n^5)$	0.10 sg	3.20 sg	24.30 sg	1.70 min	5.20 min
$O(2^n)$	1.00 ms	1.00 sg	17.90 min	12 días	35 años
$O(3^n)$	0.59 sg	58 min	6 años	3855 siglos	2×10^8 siglos!

Problemas bien resueltos

Conclusión: Los algoritmos polinomiales se consideran satisfactorios (cuanto menor sea el grado, mejor), y los algoritmos supra-polinomiales se consideran no satisfactorios.

Problemas bien resueltos

Conclusión: Los algoritmos polinomiales se consideran satisfactorios (cuanto menor sea el grado, mejor), y los algoritmos supra-polinomiales se consideran no satisfactorios.

- ▶ Si los tamaños de instancia son pequeños, ¿es tan malo un algoritmo exponencial?
- ▶ ¿Cómo se comparan $O(n^{85})$ con $O(1,001^n)$?
- ▶ ¿Puede pasar que un algoritmo de peor caso exponencial sea eficiente en la práctica? ¿Puede pasar que en la práctica sea *el mejor*?
- ▶ ¿Qué pasa si no encuentro un algoritmo polinomial?

Técnicas de diseño de algoritmos

- ▶ Algoritmos golosos
- ▶ *Divide and conquer* (dividir y conquistar)
- ▶ Recursividad
- ▶ Programación dinámica
- ▶ *Backtracking* (búsqueda con retroceso)
- ▶ Algoritmos probabilísticos

Algoritmos golosos

Idea: Construir una solución seleccionando en cada paso la mejor alternativa, sin considerar (o haciéndolo débilmente) las implicancias de esta selección.

Algoritmos golosos

Idea: Construir una solución seleccionando en cada paso la mejor alternativa, sin considerar (o haciéndolo débilmente) las implicancias de esta selección.

- ▶ Fáciles de inventar.
- ▶ Fáciles de implementar.
- ▶ Generalmente eficientes.
- ▶ Pero no siempre “funcionan”: algunos problemas no pueden ser resueltos por este enfoque.
 - ▶ Habitualmente, proporcionan **heurísticas** sencillas para **problemas de optimización**.
 - ▶ En general permiten construir soluciones razonables, pero sub-óptimas.
- ▶ Conjunto de candidatos.
- ▶ Función de selección.

Ejemplo: El problema de la mochila

Datos de entrada:

- ▶ Capacidad $C \in \mathbb{R}_+$ de la mochila (peso máximo).
- ▶ Cantidad $n \in \mathbb{N}$ de objetos.
- ▶ Peso $p_i \in \mathbb{R}_+$ del objeto i , para $i = 1, \dots, n$.
- ▶ Beneficio $b_i \in \mathbb{R}_+$ del objeto i , para $i = 1, \dots, n$.

Problema: Determinar qué objetos debemos incluir en la mochila sin excedernos del peso máximo C , de modo tal de **maximizar** el beneficio total entre los objetos seleccionados. En la versión más simple de este problema vamos a suponer que podemos poner parte de un objeto en la mochila.

Ejemplo: El problema de la mochila

Algoritmo(s) goloso(s): Mientras no se haya excedido el peso de la mochila, agregar a la mochila el objeto i que ...

- ▶ ... tenga mayor beneficio b_i .
- ▶ ... tenga menor peso p_i .
- ▶ ... maximice b_i/p_i .

Ejemplo: El problema de la mochila

Datos de entrada:

$$C = 100, n = 5$$

	1	2	3	4	5
p	10	20	30	40	50
b	20	30	66	40	60
b/p	2.0	1.5	2.2	1.0	1.2

Ejemplo: El problema de la mochila

Datos de entrada:

$$C = 100, n = 5$$

	1	2	3	4	5
p	10	20	30	40	50
b	20	30	66	40	60
b/p	2.0	1.5	2.2	1.0	1.2

- ▶ mayor beneficio b_i : $66 + 60 + 40/2 = 146$.
- ▶ menor peso p_i : $20 + 30 + 66 + 40 = 156$.
- ▶ maximice b_i/p_i : $66 + 20 + 30 + 0,8 \cdot 60 = 164$.

Ejemplo: El problema de la mochila

- ▶ ¿Qué podemos decir en cuanto a la **calidad** de las soluciones obtenidas por estos algoritmos?
- ▶ Se puede demostrar que la selección según se maximice b_i/p_i da una solución óptima.
- ▶ ¿Qué podemos decir en cuanto a su **complejidad**?
- ▶ ¿Qué sucede si los elementos se deben poner enteros en la mochila?

Ejemplo: Tiempo de espera total en un sistema

Problema: Un servidor tiene n clientes para atender, y los puede atender en cualquier orden. Para $i = 1, \dots, n$, el tiempo necesario para atender al cliente i es $t_i \in \mathbb{R}_+$. El objetivo es determinar en qué orden se deben atender los clientes para minimizar la suma de los tiempos de espera de los clientes.

Ejemplo: Tiempo de espera total en un sistema

Problema: Un servidor tiene n clientes para atender, y los puede atender en cualquier orden. Para $i = 1, \dots, n$, el tiempo necesario para atender al cliente i es $t_i \in \mathbb{R}_+$. El objetivo es determinar en qué orden se deben atender los clientes para minimizar **la suma de los tiempos de espera** de los clientes.

Si $I = (i_1, i_2, \dots, i_n)$ es una permutación de los clientes que representa el orden de atención, entonces la suma de los tiempos de espera es

$$\begin{aligned} T &= t_{i_1} + (t_{i_1} + t_{i_2}) + (t_{i_1} + t_{i_2} + t_{i_3}) + \dots \\ &= \sum_{k=1}^n (n - k + 1) t_{i_k}. \end{aligned}$$

Ejemplo: Tiempo de espera total en un sistema

Algoritmo goloso: En cada paso, atender al cliente pendiente que tenga menor tiempo de atención.

- ▶ Retorna una permutación $I_{\text{GOL}} = (i_1, \dots, i_n)$ tal que $t_{i_j} \leq t_{i_{j+1}}$ para $j = 1, \dots, n - 1$.

Ejemplo: Tiempo de espera total en un sistema

Algoritmo goloso: En cada paso, atender al cliente pendiente que tenga menor tiempo de atención.

- ▶ Retorna una permutación $I_{\text{GOL}} = (i_1, \dots, i_n)$ tal que $t_{i_j} \leq t_{i_{j+1}}$ para $j = 1, \dots, n-1$.
- ▶ ¿Cuál es la **complejidad** de este algoritmo?

Ejemplo: Tiempo de espera total en un sistema

Algoritmo goloso: En cada paso, atender al cliente pendiente que tenga menor tiempo de atención.

- ▶ Retorna una permutación $I_{\text{GOL}} = (i_1, \dots, i_n)$ tal que $t_{i_j} \leq t_{i_{j+1}}$ para $j = 1, \dots, n-1$.
- ▶ ¿Cuál es la **complejidad** de este algoritmo?
- ▶ Este algoritmo proporciona la **solución óptima**!

Divide and conquer

- ▶ Si la instancia I de entrada es pequeña, entonces utilizar un algoritmo ad hoc para el problema.
- ▶ En caso contrario:
 - ▶ **Dividir** I en sub-instancias I_1, I_2, \dots, I_k más pequeñas.
 - ▶ Resolver **recursivamente** las k sub-instancias.
 - ▶ **Combinar** las soluciones para las k sub-instancias para obtener una solución para la instancia original I .

Ejemplo: Mergesort

Algoritmo *divide and conquer* para ordenar un arreglo A de n elementos (von Neumann, 1945).

- ▶ Si n es pequeño, ordenar por cualquier método sencillo.
- ▶ Si n es grande:
 - ▶ $A_1 :=$ primera mitad de A .
 - ▶ $A_2 :=$ segunda mitad de A .
 - ▶ Ordenar recursivamente A_1 y A_2 por separado.
 - ▶ Combinar A_1 y A_2 para obtener los elementos de A ordenados (apareo de arreglos).

Ejemplo: Mergesort

Algoritmo *divide and conquer* para ordenar un arreglo A de n elementos (von Neumann, 1945).

- ▶ Si n es pequeño, ordenar por cualquier método sencillo.
- ▶ Si n es grande:
 - ▶ $A_1 :=$ primera mitad de A .
 - ▶ $A_2 :=$ segunda mitad de A .
 - ▶ Ordenar recursivamente A_1 y A_2 por separado.
 - ▶ Combinar A_1 y A_2 para obtener los elementos de A ordenados (apareo de arreglos).

Este algoritmo contiene todos los elementos típicos de la técnica *divide and conquer*.

Ejemplo: Mergesort

Algoritmo *divide and conquer* para ordenar un arreglo A de n elementos (von Neumann, 1945).

- ▶ Si n es pequeño, ordenar por cualquier método sencillo.
- ▶ Si n es grande:
 - ▶ $A_1 :=$ primera mitad de A .
 - ▶ $A_2 :=$ segunda mitad de A .
 - ▶ Ordenar recursivamente A_1 y A_2 por separado.
 - ▶ Combinar A_1 y A_2 para obtener los elementos de A ordenados (apareo de arreglos).

Este algoritmo contiene todos los elementos típicos de la técnica *divide and conquer*.

Ejemplo: Multiplicación de Strassen

- ▶ Sean $A, B \in \mathbb{R}^{n \times n}$. El algoritmo estándar para calcular AB tiene una complejidad de $\Theta(n^3)$.
- ▶ Durante muchos años se pensaba que esta complejidad era **óptima**.

Ejemplo: Multiplicación de Strassen

- ▶ Sean $A, B \in \mathbb{R}^{n \times n}$. El algoritmo estándar para calcular AB tiene una complejidad de $\Theta(n^3)$.
- ▶ Durante muchos años se pensaba que esta complejidad era **óptima**.
- ▶ Sin embargo, Strassen (1969) pateó el tablero. Particionamos:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}.$$

Ejemplo: Multiplicación de Strassen

Definimos:

$$M_1 = (A_{21} + A_{22} - A_{11})(B_{22} - B_{12} + B_{11})$$

$$M_2 = A_{11}B_{11}$$

$$M_3 = A_{12}B_{21}$$

$$M_4 = (A_{11} - A_{21})(B_{22} - B_{12})$$

$$M_5 = (A_{21} + A_{22})(B_{12} - B_{11})$$

$$M_6 = (A_{12} - A_{21} + A_{11} - A_{22})B_{22}$$

$$M_7 = A_{22}(B_{11} + B_{22} - B_{12} - B_{21}).$$

Ejemplo: Multiplicación de Strassen

Definimos:

$$M_1 = (A_{21} + A_{22} - A_{11})(B_{22} - B_{12} + B_{11})$$

$$M_2 = A_{11}B_{11}$$

$$M_3 = A_{12}B_{21}$$

$$M_4 = (A_{11} - A_{21})(B_{22} - B_{12})$$

$$M_5 = (A_{21} + A_{22})(B_{12} - B_{11})$$

$$M_6 = (A_{12} - A_{21} + A_{11} - A_{22})B_{22}$$

$$M_7 = A_{22}(B_{11} + B_{22} - B_{12} - B_{21}).$$

Entonces,

$$AB = \begin{pmatrix} M_2 + M_3 & M_1 + M_2 + M_5 + M_6 \\ M_1 + M_2 + M_4 - M_7 & M_1 + M_2 + M_4 + M_5 \end{pmatrix}.$$

Ejemplo: Multiplicación de Strassen

- ▶ Este algoritmo permite calcular el producto AB en tiempo $O(n^{\log_2(7)}) = O(n^{2,81})$ (!).
- ▶ Requiere 7 multiplicaciones de matrices de tamaño $n/2 \times n/2$, en comparación con las 8 multiplicaciones del algoritmo estándar.
- ▶ La cantidad de sumas (y restas) de matrices es mucho mayor.

Ejemplo: Multiplicación de Strassen

- ▶ Este algoritmo permite calcular el producto AB en tiempo $O(n^{\log_2(7)}) = O(n^{2,81})$ (!).
- ▶ Requiere 7 multiplicaciones de matrices de tamaño $n/2 \times n/2$, en comparación con las 8 multiplicaciones del algoritmo estándar.
- ▶ La cantidad de sumas (y restas) de matrices es mucho mayor.
- ▶ El algoritmo asintóticamente más eficiente conocido a la fecha tiene una complejidad de $O(n^{2,376})$ (Coppersmith y Winograd, 1987).

Backtracking

Idea: Técnica para recorrer sistemáticamente todas las posibles configuraciones del espacio de soluciones de un problema computacional.

Backtracking

Idea: Técnica para recorrer sistemáticamente todas las posibles configuraciones del espacio de soluciones de un problema computacional.

- ▶ Habitualmente, utiliza un **vector** $a = (a_1, a_2, \dots, a_n)$ para representar una solución candidata, cada a_i pertenece a un dominio/conjunto finito A_i .

Backtracking

Idea: Técnica para recorrer sistemáticamente todas las posibles configuraciones del espacio de soluciones de un problema computacional.

- ▶ Habitualmente, utiliza un **vector** $a = (a_1, a_2, \dots, a_n)$ para representar una solución candidata, cada a_i pertenece a un dominio/conjunto finito A_i .
- ▶ El espacio de soluciones es el producto cartesiano $A_1 \times \dots \times A_n$.

Backtracking

Idea: Técnica para recorrer sistemáticamente todas las posibles configuraciones del espacio de soluciones de un problema computacional.

- ▶ Habitualmente, utiliza un **vector** $a = (a_1, a_2, \dots, a_n)$ para representar una solución candidata, cada a_i pertenece a un dominio/conjunto finito A_i .
- ▶ El espacio de soluciones es el producto cartesiano $A_1 \times \dots \times A_n$.
- ▶ En cada paso se extienden las soluciones parciales $a = (a_1, a_2, \dots, a_k)$, $k < n$, agregando un elemento más, $a_{k+1} \in S_{k+1} \subseteq A_{k+1}$, al final del vector a . Las nuevas soluciones parciales son sucesoras de la anterior.

Backtracking

Idea: Técnica para recorrer sistemáticamente todas las posibles configuraciones del espacio de soluciones de un problema computacional.

- ▶ Habitualmente, utiliza un **vector** $a = (a_1, a_2, \dots, a_n)$ para representar una solución candidata, cada a_i pertenece a un dominio/conjunto finito A_i .
- ▶ El espacio de soluciones es el producto cartesiano $A_1 \times \dots \times A_n$.
- ▶ En cada paso se extienden las soluciones parciales $a = (a_1, a_2, \dots, a_k)$, $k < n$, agregando un elemento más, $a_{k+1} \in S_{k+1} \subseteq A_{k+1}$, al final del vector a . Las nuevas soluciones parciales son sucesoras de la anterior.
- ▶ Si S_{k+1} (conjunto de soluciones sucesoras) es vacío, esa rama no se continua explorando.

Backtracking

- ▶ Se puede pensar este espacio como un árbol dirigido, donde cada vértice representa una solución parcial y un vértice x es hijo de y si la solución parcial x se puede extender desde la solución parcial y .
- ▶ Permite descartar configuraciones antes de explorarlas (podar el árbol).

Backtracking: Esquema General - Todas las soluciones

```
algoritmo  $BT(a, k)$   
  si  $k == n + 1$  entonces  
    procesar( $a$ )  
    retornar  
  sino  
    para cada  $a' \in \text{Sucesores}(a, k)$   
       $BT(a', k + 1)$   
    fin para  
  fin si  
  retornar
```

Backtracking: Esquema General - Una solución

```
algoritmo  $BT(a, k)$   
  si  $k == n + 1$  entonces  
     $sol \leftarrow a$   
     $encontro \leftarrow \mathbf{true}$   
  sino  
    para cada  $a' \in \text{Sucesores}(a, k)$   
       $BT(a', k + 1)$   
      si  $encontro$  entonces  
        retornar  
      fin si  
    fin para  
  fin si  
  retornar
```

- ▶ sol variable global que guarda la solución.
- ▶ $encontro$ variable booleana global que indica si ya se encontró una solución. Inicialmente está en **false**.

Ejemplo: Problema de las 8 reinas

Ubicar 8 reinas en el tablero de ajedrez (8×8) sin que ninguna “amenace” a otra.

Ejemplo: Problema de las 8 reinas

Ubicar 8 reinas en el tablero de ajedrez (8×8) sin que ninguna “amenace” a otra.

- ▶ ¿Cuántas combinaciones del tablero hay que considerar?

$$64^8 \approx 10^{14}$$

Ejemplo: Problema de las 8 reinas

Ubicar 8 reinas en el tablero de ajedrez (8×8) sin que ninguna “amenace” a otra.

- ▶ ¿Cuántas combinaciones del tablero hay que considerar?

$$64^8 \approx 10^{14}$$

- ▶ Sabemos que dos reinas no pueden estar en el mismo casillero.

$$\binom{64}{8} = 442616536$$

Ejemplo: Problema de las 8 reinas

Ubicar 8 reinas en el tablero de ajedrez (8×8) sin que ninguna “amenace” a otra.

- ▶ ¿Cuántas combinaciones del tablero hay que considerar?

$$64^8 \approx 10^{14}$$

- ▶ Sabemos que dos reinas no pueden estar en el mismo casillero.

$$\binom{64}{8} = 442616536$$

- ▶ Sabemos que cada fila debe tener exactamente una reina.
Cada solución parcial puede estar representada por (a_1, \dots, a_k) , $k \leq 8$, con $a_i \in \{1, \dots, 8\}$ indicando la columna de la reina que está en la fila i .

Ejemplo: Problema de las 8 reinas

Ubicar 8 reinas en el tablero de ajedrez (8×8) sin que ninguna “amenace” a otra.

- ▶ ¿Cuántas combinaciones del tablero hay que considerar?

$$64^8 \approx 10^{14}$$

- ▶ Sabemos que dos reinas no pueden estar en el mismo casillero.

$$\binom{64}{8} = 442616536$$

- ▶ Sabemos que cada fila debe tener exactamente una reina.
Cada solución parcial puede estar representada por (a_1, \dots, a_k) , $k \leq 8$, con $a_i \in \{1, \dots, 8\}$ indicando la columna de la reina que está en la fila i .

Tenemos ahora $8^8 = 16777216$ combinaciones.

Ejemplo: Problema de las 8 reinas

- ▶ Es más, una misma columna debe tener exactamente una reina.

Ejemplo: Problema de las 8 reinas

- Es más, una misma columna debe tener exactamente una reina.

Se reduce a $8! = 40320$ combinaciones.

Ejemplo: Problema de las 8 reinas

- Es más, una misma columna debe tener exactamente una reina.

Se reduce a $8! = 40320$ combinaciones.

- ¿Cómo construir el conjunto $\text{Sucesores}(a, k)$?

Ejemplo: Problema de las 8 reinas

- Es más, una misma columna debe tener exactamente una reina.

Se reduce a $8! = 40320$ combinaciones.

- ¿Cómo construir el conjunto $\text{Sucesores}(a, k)$?

$$\text{Sucesores}(a, k) = \{(a, a_k) : a_k \in \{1, \dots, 8\}, a_k - a_j \notin \{k - j, 0, j - k\} \forall j \in \{1, \dots, k - 1\}\}.$$

Ejemplo: Problema de las 8 reinas

- Es más, una misma columna debe tener exactamente una reina.

Se reduce a $8! = 40320$ combinaciones.

- ¿Cómo construir el conjunto $\text{Sucesores}(a, k)$?

$$\text{Sucesores}(a, k) = \{(a, a_k) : a_k \in \{1, \dots, 8\}, a_k - a_j \notin \{k - j, 0, j - k\} \forall j \in \{1, \dots, k - 1\}\}.$$

- Ahora estamos en condición de implementar un algoritmo para resolver el problema!

Ejemplo: Problema de las 8 reinas

- Es más, una misma columna debe tener exactamente una reina.

Se reduce a $8! = 40320$ combinaciones.

- ¿Cómo construir el conjunto $\text{Sucesores}(a, k)$?

$$\text{Sucesores}(a, k) = \{(a, a_k) : a_k \in \{1, \dots, 8\}, a_k - a_j \notin \{k - j, 0, j - k\} \forall j \in \{1, \dots, k - 1\}\}.$$

- Ahora estamos en condición de implementar un algoritmo para resolver el problema!
- ¿Cómo generalizar para el problema de n reinas?

Programación Dinámica

- ▶ Es aplicada típicamente a problemas de optimización, donde puede haber muchas soluciones, cada una tiene un valor asociado y pretendemos obtener la solución con mejor valor.

Programación Dinámica

- ▶ Es aplicada típicamente a problemas de optimización, donde puede haber muchas soluciones, cada una tiene un valor asociado y pretendemos obtener la solución con mejor valor.
- ▶ Al igual que "dividir y conquistar", el problema es dividido en subproblemas de tamaños menores que son más fáciles de resolver. Una vez resueltos estos subproblemas, se combinan las soluciones obtenidas para generar la solución del problema original.

Programación Dinámica

- ▶ Es aplicada típicamente a problemas de optimización, donde puede haber muchas soluciones, cada una tiene un valor asociado y pretendemos obtener la solución con mejor valor.
- ▶ Al igual que "dividir y conquistar", el problema es dividido en subproblemas de tamaños menores que son más fáciles de resolver. Una vez resueltos estos subproblemas, se combinan las soluciones obtenidas para generar la solución del problema original.
- ▶ Es **bottom up** y no es recursivo.

Programación Dinámica

- ▶ Es aplicada típicamente a problemas de optimización, donde puede haber muchas soluciones, cada una tiene un valor asociado y pretendemos obtener la solución con mejor valor.
- ▶ Al igual que "dividir y conquistar", el problema es dividido en subproblemas de tamaños menores que son más fáciles de resolver. Una vez resueltos estos subproblemas, se combinan las soluciones obtenidas para generar la solución del problema original.
- ▶ Es **bottom up** y no es recursivo.
- ▶ Se guardan las soluciones de los subproblemas para no calcularlos más de una vez.

Programación Dinámica

- ▶ **Principio de optimalidad de Bellman:**

Un problema de optimización satisface el principio de optimalidad de Bellman si en una sucesión óptima de decisiones o elecciones, cada subsucesión es a su vez óptima.

Programación Dinámica

- ▶ **Principio de optimalidad de Bellman:**

Un problema de optimización satisface el principio de optimalidad de Bellman si en una sucesión óptima de decisiones o elecciones, cada subsucesión es a su vez óptima.

- ▶ Es decir, si miramos una subsolución de la solución óptima, debe ser solución del subproblema asociado a esa subsolución.

Programación Dinámica

- ▶ **Principio de optimalidad de Bellman:**

Un problema de optimización satisface el principio de optimalidad de Bellman si en una sucesión óptima de decisiones o elecciones, cada subsucesión es a su vez óptima.

- ▶ Es decir, si miramos una subsolución de la solución óptima, debe ser solución del subproblema asociado a esa subsolución.
- ▶ El principio de optimalidad es condición necesaria para poder usar la técnica de programación dinámica.

Programación Dinámica

- ▶ **Principio de optimalidad de Bellman:**

Un problema de optimización satisface el principio de optimalidad de Bellman si en una sucesión óptima de decisiones o elecciones, cada subsucesión es a su vez óptima.

- ▶ Es decir, si miramos una subsolución de la solución óptima, debe ser solución del subproblema asociado a esa subsolución.
- ▶ El principio de optimalidad es condición necesaria para poder usar la técnica de programación dinámica.
- ▶ Ejemplos:
 - ▶ Se cumple en camino mínimo (sin distancias negativas).
 - ▶ No se cumple en camino máximo.

Ejemplo: Coeficientes binomiales

- **Definición:** Si $n \geq 0$ y $0 \leq k \leq n$, se define

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

- **Teorema:** Si $n \geq 0$ y $0 \leq k \leq n$, entonces

$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0 \text{ o } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \\ 0 & \text{caso contrario} \end{cases}$$

Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	...	$k-1$	k
0								
1								
2								
3								
4								
\vdots								
$k-1$								
k								
\vdots								
$n-1$								
n								

Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	...	$k-1$	k
0	1							
1	1	1						
2	1		1					
3	1			1				
4	1				1			
\vdots	\vdots					\ddots		
$k-1$	1						1	
k	1							1
\vdots	\vdots							
$n-1$	1							
n	1							

Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	...	$k-1$	k
0	1							
1	1	1						
2	1		1					
3	1			1				
4	1				1			
\vdots	\vdots					\ddots		
$k-1$	1						1	
k	1							1
\vdots	\vdots							
$n-1$	1							
n	1							

Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	...	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1			1				
4	1				1			
\vdots	\vdots					\ddots		
$k-1$	1						1	
k	1							1
\vdots	\vdots							
$n-1$	1							
n	1							

Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	...	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1			1				
4	1				1			
\vdots	\vdots					\ddots		
$k-1$	1						1	
k	1							1
\vdots	\vdots							
$n-1$	1							
n	1							

Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	...	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3		1				
4	1				1			
\vdots	\vdots					\ddots		
$k-1$	1						1	
k	1							1
\vdots	\vdots							
$n-1$	1							
n	1							

Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	...	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3		1				
4	1				1			
\vdots	\vdots					\ddots		
$k-1$	1						1	
k	1							1
\vdots	\vdots							
$n-1$	1							
n	1							

Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	...	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1				1			
\vdots	\vdots					\ddots		
$k-1$	1						1	
k	1							1
\vdots	\vdots							
$n-1$	1							
n	1							

Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	...	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1				1			
\vdots	\vdots					\ddots		
$k-1$	1						1	
k	1							1
\vdots	\vdots							
$n-1$	1							
n	1							

Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	...	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4			1			
\vdots	\vdots					\ddots		
$k-1$	1						1	
k	1							1
\vdots	\vdots							
$n-1$	1							
n	1							

Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	...	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4			1			
\vdots	\vdots					\ddots		
$k-1$	1						1	
k	1							1
\vdots	\vdots							
$n-1$	1							
n	1							

Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	...	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6		1			
\vdots	\vdots					\ddots		
$k-1$	1						1	
k	1							1
\vdots	\vdots							
$n-1$	1							
n	1							

Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	...	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6		1			
\vdots	\vdots					\ddots		
$k-1$	1						1	
k	1							1
\vdots	\vdots							
$n-1$	1							
n	1							

Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	...	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
\vdots	\vdots					\ddots		
$k-1$	1						1	
k	1							1
\vdots	\vdots							
$n-1$	1							
n	1							

Coeficientes binomiales

algoritmo *combinatorio*(n, k)

entrada: dos enteros n y k

salida: $\binom{n}{k}$

para $i = 1$ **hasta** n **hacer**

$A[i][0] \leftarrow 1$

fin para

para $j = 0$ **hasta** k **hacer**

$A[j][j] \leftarrow 1$

fin para

para $i = 2$ **hasta** n **hacer**

para $j = 2$ **hasta** $\min(i - 1, k)$ **hacer**

$A[i][j] \leftarrow A[i - 1][j - 1] + A[i - 1][j]$

fin para

fin para

retornar $A[n][k]$

Coeficientes binomiales

- ▶ Función recursiva:
 - ▶ Complejidad

Coeficientes binomiales

- ▶ Función recursiva:
 - ▶ Complejidad $\Omega\left(\binom{n}{k}\right)$.

Coeficientes binomiales

- ▶ Función recursiva:
 - ▶ Complejidad $\Omega(\binom{n}{k})$.
- ▶ Programación dinámica:
 - ▶ Complejidad

Coeficientes binomiales

- ▶ Función recursiva:
 - ▶ Complejidad $\Omega(\binom{n}{k})$.
- ▶ Programación dinámica:
 - ▶ Complejidad $O(nk)$.

Coeficientes binomiales

- ▶ Función recursiva:
 - ▶ Complejidad $\Omega\left(\binom{n}{k}\right)$.
- ▶ Programación dinámica:
 - ▶ Complejidad $O(nk)$.
 - ▶ Espacio

Coeficientes binomiales

- ▶ Función recursiva:
 - ▶ Complejidad $\Omega(\binom{n}{k})$.
- ▶ Programación dinámica:
 - ▶ Complejidad $O(nk)$.
 - ▶ Espacio $\Theta(k)$: sólo necesitamos almacenar la fila anterior de la que estamos calculando.

Multiplicación de n matrices

$$M = M_1 \times M_2 \times \dots M_n$$

Multiplicación de n matrices

$$M = M_1 \times M_2 \times \dots M_n$$

Por la propiedad asociativa del producto de matrices esto puede hacerse de muchas formas. Queremos determinar la que minimiza el número de operaciones necesarias. Por ejemplo: las dimensiones de A es de 13×5 , B de 5×89 , C de 89×3 y D de 3×34 .

Tenemos

- ▶ $((AB)C)D$ requiere 10582 multiplicaciones.
- ▶ $(AB)(CD)$ requiere 54201 multiplicaciones.
- ▶ $(A(BC))D$ requiere 2856 multiplicaciones.
- ▶ $A((BC)D)$ requiere 4055 multiplicaciones.
- ▶ $A(B(CD))$ requiere 26418 multiplicaciones.

Multiplicación de n matrices

- ▶ La mejor forma de multiplicar todas las matrices es multiplicar las matrices 1 a i por un lado y las matrices $i + 1$ a n por otro lado y luego multiplicar estos dos resultados para algún $1 \leq i \leq n - 1$.
- ▶ En la solución óptima de $M = M_1 \times M_2 \times \dots M_n$, estos dos subproblemas, $M_1 \times M_2 \times \dots M_i$ y $M_{i+1} \times M_{i+2} \times \dots M_n$ deben estar resueltos de forma óptima: se cumple el principio de optimalidad.
- ▶ Llamamos $m[i][j]$ solución del subproblema $M_i \times M_{i+1} \times \dots M_j$, es decir la cantidad mínima de multiplicaciones necesarias para calcular $M_i \times M_{i+1} \times \dots M_j$.

Multiplicación de n matrices

Suponemos que las dimensiones de las matrices están dadas por un vector $d \in \mathbb{N}^{n+1}$, tal que la matriz M_i tiene $d[i - 1]$ filas y $d[i]$ columnas para $1 \leq i \leq n$. Entonces:

Multiplicación de n matrices

Suponemos que las dimensiones de las matrices están dadas por un vector $d \in \mathbb{N}^{n+1}$, tal que la matriz M_i tiene $d[i - 1]$ filas y $d[i]$ columnas para $1 \leq i \leq n$. Entonces:

- Para $i = 1, 2, \dots, n$, $m[i][i] =$

Multiplicación de n matrices

Suponemos que las dimensiones de las matrices están dadas por un vector $d \in \mathbb{N}^{n+1}$, tal que la matriz M_i tiene $d[i - 1]$ filas y $d[i]$ columnas para $1 \leq i \leq n$. Entonces:

- ▶ Para $i = 1, 2, \dots, n$, $m[i][i] = 0$

Multiplicación de n matrices

Suponemos que las dimensiones de las matrices están dadas por un vector $d \in N^{n+1}$, tal que la matriz M_i tiene $d[i - 1]$ filas y $d[i]$ columnas para $1 \leq i \leq n$. Entonces:

- ▶ Para $i = 1, 2, \dots, n$, $m[i][i] = 0$
- ▶ Para $i = 1, 2, \dots, n - 1$, $m[i][i + 1] =$

Multiplicación de n matrices

Suponemos que las dimensiones de las matrices están dadas por un vector $d \in N^{n+1}$, tal que la matriz M_i tiene $d[i-1]$ filas y $d[i]$ columnas para $1 \leq i \leq n$. Entonces:

- ▶ Para $i = 1, 2, \dots, n$, $m[i][i] = 0$
- ▶ Para $i = 1, 2, \dots, n-1$, $m[i][i+1] = d[i-1]d[i]d[i+1]$

Multiplicación de n matrices

Suponemos que las dimensiones de las matrices están dadas por un vector $d \in N^{n+1}$, tal que la matriz M_i tiene $d[i-1]$ filas y $d[i]$ columnas para $1 \leq i \leq n$. Entonces:

- ▶ Para $i = 1, 2, \dots, n$, $m[i][i] = 0$
- ▶ Para $i = 1, 2, \dots, n-1$, $m[i][i+1] = d[i-1]d[i]d[i+1]$
- ▶ Para $s = 2, \dots, n-1$, $i = 1, 2, \dots, n-s$,

$$m[i][i+s] =$$

Multiplicación de n matrices

Suponemos que las dimensiones de las matrices están dadas por un vector $d \in N^{n+1}$, tal que la matriz M_i tiene $d[i-1]$ filas y $d[i]$ columnas para $1 \leq i \leq n$. Entonces:

- ▶ Para $i = 1, 2, \dots, n$, $m[i][i] = 0$
- ▶ Para $i = 1, 2, \dots, n-1$, $m[i][i+1] = d[i-1]d[i]d[i+1]$
- ▶ Para $s = 2, \dots, n-1$, $i = 1, 2, \dots, n-s$,

$$m[i][i+s] = \min_{i \leq k < i+s} (m[i][k] + m[k+1][i+s] + d[i-1]d[k]d[i+s])$$

Multiplicación de n matrices

Suponemos que las dimensiones de las matrices están dadas por un vector $d \in N^{n+1}$, tal que la matriz M_i tiene $d[i-1]$ filas y $d[i]$ columnas para $1 \leq i \leq n$. Entonces:

- ▶ Para $i = 1, 2, \dots, n$, $m[i][i] = 0$
- ▶ Para $i = 1, 2, \dots, n-1$, $m[i][i+1] = d[i-1]d[i]d[i+1]$
- ▶ Para $s = 2, \dots, n-1$, $i = 1, 2, \dots, n-s$,

$$m[i][i+s] = \min_{i \leq k < i+s} (m[i][k] + m[k+1][i+s] + d[i-1]d[k]d[i+s])$$

La solución del problema es $m[1][n]$.

Algoritmos probabilísticos

- ▶ Cuando un algoritmo tiene que hacer una elección a veces es preferible elegir al azar en vez de gastar mucho tiempo tratando de ver cual es la mejor elección.
- ▶ Algoritmos numéricos: dan una respuesta aproximada.
 - ▶ + tiempo proceso \Rightarrow + precisión
 - ▶ Ejemplo: cálculo de integral
- ▶ Algoritmos de Montecarlo: con alta probabilidad dan una respuesta correcta.
 - ▶ + tiempo proceso \Rightarrow + probabilidad de acertar
 - ▶ Ejemplo: determinar la existencia en un arreglo de un elemento mayor a un valor dado

Algoritmos probabilísticos

- ▶ Algoritmos Las Vegas: si da respuesta es correcta pero puede no darla.
 - ▶ + tiempo proceso \Rightarrow + probabilidad de obtener la respuesta
 - ▶ Ejemplo: problema de las n reinas
- ▶ Algoritmos Sherwood: randomiza un algoritmo determinístico donde hay una gran diferencia entre el peor caso y caso promedio. Elimina la diferencia entre buenas y malas instancias.
 - ▶ Ejemplo: quicksort con pivote seleccionado aleatoriamente

Heurísticas

- ▶ Dado un problema Π , un algoritmo heurístico es un algoritmo que intenta obtener soluciones para el problema que intenta resolver pero no necesariamente lo hace en todos los casos.
- ▶ Sea Π un problema de optimización, I una instancia del problema, $x^*(I)$ el valor óptimo de la función a optimizar en dicha instancia. Un algoritmo heurístico obtiene una solución con un valor que se espera sea cercano a ese óptimo pero no necesariamente el óptimo.
- ▶ Si H es un algoritmo heurístico para un problema de optimización llamamos $x^H(I)$ al valor que devuelve la heurística.

Algoritmos aproximados

Decimos que H es un algoritmo ϵ – aproximado para el problema Π si para algún $\epsilon > 0$

$$|x^H(I) - x^*(I)| \leq \epsilon |x^*(I)|$$