Algoritmos y Estructuras de Datos III

Segundo cuatrimestre 2013

Técnicas de diseño de algoritmos

## Técnicas de diseño de algoritmos

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

- 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 $i = 1, \ldots, n$.
- Beneficio $b_i \in \mathbb{R}_+$ del objeto $i$, para $i = 1, \ldots, n$.

**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 |

- 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 $i = 1, \ldots, n$, 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 $I = (i_1, i_2, \ldots, i_n)$ es una permutación de los clientes que representa el orden de atención, entonces la suma de los tiempos de espera es

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

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

## Backtracking

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

- Habitualmente, utiliza un vector $a = (a_1, a_2, \ldots, a_n)$ para representar una solución candidata, cada $a_i$ pertenece un dominio/conjunto ordenado y finito $A_i$.
- El espacio de soluciones es el producto cartesiano $A_1 \times \ldots \times A_n$.
- En cada paso se extienden las soluciones parciales $a = (a_1, a_2, \ldots, a_k)$, $k < n$, 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 sucesores de la anterior.
- Si $S_{k+1}$ es vacío, se *retrocede* a la solución parcial $(a_1, a_2, \ldots, a_{k-1})$.

## 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 a es solución entonces
        procesar(a)
        retornar
    sino
        para cada a' ∈ Sucesores(a, k)
            BT(a', k + 1)
        fin para
    fin si
    retornar
```

## Backtracking: Esquema General - Una solución

```
algoritmo BT(a, k)
    si a es solución entonces
        sol ← a
        encontro ← true
    sino
        para cada a' ∈ 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 ($8 \times 8$) 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 $(a_1, \ldots, a_k)$, $k \leq 8$, con $a_i \in \{1, \ldots, 8\}$ 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.

  Se reduce a $8! = 40320$ combinaciones.

- ¿Cómo chequear un vector $a$ es una solución?

$$a_i - a_j \notin \{i - j, 0, j - 1\}, \forall i, j \in \{1, \ldots, 8\} \text{ e } i \neq j.$$

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

## 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 $I_1, I_2, \ldots, I_k$ 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).

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.
- Sin embargo, Strassen (1969) pateó el tablero. Particionamos:

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

## Ejemplo: Multiplicación de Strassen

Definimos:

$$\begin{aligned} M_1 &= (A_{21} + A_{22} - A_{11})(B_{22} - B_{12} + B_{11}) \\ M_2 &= A_{11} B_{11} \\ M_3 &= A_{12} B_{21} \\ M_4 &= (A_{11} - A_{21})(B_{22} - B_{12}) \\ M_5 &= (A_{21} + A_{22})(B_{12} - B_{11}) \\ M_6 &= (A_{12} - A_{21} + A_{11} - A_{22}) B_{22} \\ M_7 &= A_{22}(B_{11} + B_{22} - B_{12} - B_{21}). \end{aligned}$$

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.

- El algoritmo asintóticamente más eficiente conocido a la fecha tiene una complejidad de $O(n^{2,376})$ (Coppersmith y Winograd, 1987).

## Programación Dinámica

- Al igual que "dividir y conquistar", el problema es dividido en subproblemas de tamaños menores que son más fácil de resolver. Una vez resueltos esto 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.
- Principio de optimalidad: 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. Es condición necesaria para poder usar la técnica de programación dinámica.
  - Se cumple en camino mínimo (sin distancias negativos).
  - No se cumple en camino máximo.

## Programación Dinámica: ejemplos

- Coeficientes binomiales $\binom{n}{k}$
- Producto de matrices
- Mochila 0/1.
- Subsecuencia creciente máxima
- Comparación de secuencias de ADN

## Coeficientes binomiales

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

## Coeficientes binomiales

| | 0 | 1 | 2 | 3 | 4 | $\cdots$ | $k-1$ | $k$ |
|---|---|---|---|---|---|---|---|---|
| 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

| | 0 | 1 | 2 | 3 | 4 | $\cdots$ | $k-1$ | $k$ |
|---|---|---|---|---|---|---|---|---|
| 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

| | 0 | 1 | 2 | 3 | 4 | $\cdots$ | $k-1$ | $k$ |
|---|---|---|---|---|---|---|---|---|
| 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

| | 0 | 1 | 2 | 3 | 4 | $\cdots$ | $k-1$ | $k$ |
|---|---|---|---|---|---|---|---|---|
| 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

| | 0 | 1 | 2 | 3 | 4 | $\cdots$ | $k-1$ | $k$ |
|---|---|---|---|---|---|---|---|---|
| 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

| | 0 | 1 | 2 | 3 | 4 | $\cdots$ | $k-1$ | $k$ |
|---|---|---|---|---|---|---|---|---|
| 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

| | 0 | 1 | 2 | 3 | 4 | $\cdots$ | $k-1$ | $k$ |
|---|---|---|---|---|---|---|---|---|
| 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

| | 0 | 1 | 2 | 3 | 4 | $\cdots$ | $k-1$ | $k$ |
|---|---|---|---|---|---|---|---|---|
| 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 ("dividir y conquistar"):
  - 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

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

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 $13 \times 5$, $B$ de $5 \times 89$, $C$ de $89 \times 3$ y $D$ de $3 \times 34$. 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 $M = M_1 \times M_2 \times \ldots M_n$, estos dos subproblemas, $M_1 \times M_2 \times \ldots M_i$ y $M_{i+1} \times M_{i+2} \times \ldots M_n$ deben estar resueltos de forma óptima: se cumple el principio de optimalidad.
- Llamamos $m[i][j]$ solución del subproblema $M_i \times M_{i+1} \times \ldots M_j$, es decir la cantidad mínima de multiplicaciones necesarias para calcular $M_i \times M_{i+1} \times \ldots M_j$.

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

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

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

## Multiplicación de $n$ matrices

- ¿Algoritmo?
- ¿Complejidad?

## 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 Monte Carlo: con alta probabilidad dan una respuesta es exacta.
  - + tiempo proceso $\Rightarrow$ + probabilidad de acertar
  - ejemplo: determinar la existencia de un elemento mayor en un arreglo

## 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 8 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.
  - quicksort