

# Algoritmos y Complejidad

## Algoritmos *greedy*

Pablo R. Fillottrani

Depto. Ciencias e Ingeniería de la Computación  
Universidad Nacional del Sur

Primer Cuatrimestre 2017



## Generalidades

- ▶ los algoritmos *greedy* son algoritmos que toman decisiones de corto alcance, basadas en información inmediatamente disponible, sin importar consecuencias futuras.
- ▶ se usan generalmente para resolver problemas de *optimización*.
- ▶ en general son algoritmos eficientes y fáciles de implementar, si es que funcionan (**no siempre son correctos!!**).



## Algoritmos *greedy*

Generalidades

Problema de la mochila

Scheduling de procesos

Códigos de Huffman



## Ejemplo

**Problema:** Dado un conjunto de monedas, ¿cuál es la mínima cantidad de monedas necesarias para pagar  $n$  centavos?.

**Solución *greedy*:** Dar en lo posible monedas de denominación grande.



## Características generales de todo algoritmo *greedy*

- ▶ se dispone de un conjunto  $C$  de **candidatos** de los cuales se debe seleccionar un subconjunto que optimice alguna propiedad.
- ▶ a medida que avanza el algoritmo, se van seleccionando candidatos y se los coloca en el conjunto  $S$  de **candidatos aceptados**, o  $R$  de **candidatos rechazados**.



## Características generales de todo algoritmo *greedy*

- ▶ existe una función `esSolución()` que determina si un conjunto de candidatos es una **solución**, no necesariamente optimal, del problema.
- ▶ existe una función `esViable()` que determina si un conjunto de candidatos es **posible de ser extendido para formar una solución**, no necesariamente optimal, del problema.
- ▶ existe una función `selección()` que devuelve el **candidato más promisorio** del conjunto de aquellos que todavía no han sido considerados.



## Esquema general para un algoritmo *greedy*

```
C ::= conjunto de candidatos; S ::= {}  
WHILE (C != {} and ! esSolución(S))  
  x ::= selección(C); C ::= C - {x}  
  IF esViable(S + {x})  
    S ::= S + {x}  
  ENDIF  
ENDWHILE  
IF esSolución(S)  
  RETURN S  
ELSE  
  RETURN "No encontré soluciones"  
ENDIF
```



## Problema de la mochila

**Problema:** se tienen  $n$  objetos (cada objeto  $i$  tiene un peso  $w_i$  y un valor  $v_i$ ); y una mochila con capacidad máxima de  $W$ . Se pretende encontrar la manera de cargar la mochila de forma que se maximice el valor de lo transportado y se respete su capacidad máxima.

- ▶ se quiere encontrar valores  $x_i, 0 \leq x_i \leq 1$  de forma que

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

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



- ▶ claramente, si  $\sum_{i=1}^n w_i \leq W$  entonces  $x_i = 1$  es optimal.
- ▶ los casos interesantes aparecen cuando  $\sum_{i=1}^n w_i > W$ .
- ▶ se puede implementar un algoritmo *greedy* con diversas estrategias de selección.



## Algoritmo *greedy*

- ▶ datos de entrada: arreglos  $w[1..n]$ , y  $v[1..n]$  contienen con los pesos y valores de los objetos.
- ▶ datos de salida: arreglo  $x[1..n]$  con la porción de cada elemento que se carga en la mochila.
- ▶  $x[i]=1$  significa que el objeto  $i$  se lleva completo;  $x[i]=0$  que nada se lleva del objeto  $i$ ; y  $x[i]=r$ ,  $0 < r < 1$  significa que el elemento  $i$  se lleva fraccionado.



## Algoritmo *greedy*

```
FOR i ::=1 TO n
  x[i] ::= 0
ENDFOR
peso ::= 0
WHILE peso < W
  i ::= seleccion() //no definido cómo
  IF peso+w[i] < W
    x[i] ::= 1; peso ::= peso+w[i]
  ELSE
    x[i] ::= (W-peso)/w[i]; peso ::= W
  ENDIF
ENDWHILE; RETURN x
```



- ▶ la función `seleccion()` no está especificada.
- ▶ para definirla se pueden considerar tres estrategias diferentes:
  1. seleccionar el elemento de mayor valor
  2. seleccionar el elemento de menor peso
  3. seleccionar el elemento que tenga mayor valor por unidad de peso



## Ejemplo de aplicación

- ▶ sea  $n = 5$ ,  $W = 100$  y objetos con los siguientes pesos y valores:

	obj. 1	obj. 2	obj. 3	obj. 4	obj. 5
w	10	20	30	40	50
v	20	30	66	40	60

- ▶ las soluciones con las tres estrategias de selección son:

	obj. 1	obj. 2	obj. 3	obj. 4	obj. 5	Valor
Max $v_i$	0	0	0 1	0 0,5	0 1	146
Min $w_i$	0 1	0 1	0 1	0 1	0	156
Max $v_i/w_i$	0 1	0 1	0 1	0	0 0,8	164



- ▶ el ejemplo anterior demuestra que las dos primeras estrategias resultan en algoritmos que no son correctos.
- ▶ ¿es correcta la tercer estrategia?



## Correctitud

### Teorema 1 (Correctitud del algoritmo greedy para la mochila)

El algoritmo greedy para el problema de la mochila con selección por mayor  $v_i/w_i$  siempre encuentra una solución optimal.

#### Prueba.

Sea  $X = (x_1, x_2, \dots, x_n)$  la solución que encuentra el algoritmo, y  $Y = (y_1, y_2, \dots, y_n)$  cualquier otra solución viable (o sea tal que  $\sum_{i=1}^n y_i w_i \leq W$ ). Se prueba que  $valor(X) - valor(Y) \geq 0$ , luego  $X$  es una solución optimal. □



## Análisis del tiempo de ejecución

- ▶ si se ordenan los elementos antes del ciclo greedy, la selección en cada iteración puede hacerse en tiempo constante y el algoritmo es entonces de  $\Theta(n \log n)$ , determinado por el ordenamiento.
- ▶ si se usa un heap ordenado por la estrategia de selección, el tiempo de inicialización cae a  $\Theta(n)$ , pero cada selección obliga a mantener la estructura (heapify) por lo que el algoritmo también resulta de  $\Theta(n \log n)$ .
- ▶ ¿Cuál de estas dos implementaciones es más conveniente?



## Definición del problema

- ▶ se tiene un servidor (que puede ser un procesador, un cajero en un banco, un surtidor de nafta, etc.) que tiene  $n$  clientes que servir.
- ▶ el tiempo de servicio requerido por cada cliente es conocido previamente:  $t_i, 1 \leq i \leq n$ .
- ▶ **Problema:** **SCHEDULING** se quiere encontrar una secuencia de atención al cliente que minimice el tiempo total de espera de los clientes:

$$\text{Tiempo de espera} = \sum_{i=1}^n (\text{tiempo del cliente } i \text{ en el sistema})$$



## Ejemplo

- ▶ tres clientes numerados 1, 2, 3 con tiempos  $t_1 = 5, t_2 = 10, t_3 = 3$

Scheduling	Tiempo de espera
123:	$5 + (5+10) + (5+10+3) = 38$
132:	$5 + (5+3) + (5+3+10) = 31$
213:	$10 + (10+5) + (10+5+3) = 43$
231:	$10 + (10+3) + (10+3+5) = 41$
312:	$3 + (3+5) + (3+5+10) = 29$
321:	$3 + (3+10) + (3+10+5) = 34$

← optimal



## Algoritmo *greedy*

- ▶ el ejemplo anterior sugiere un algoritmo *greedy* en donde la selección se hace en base el menor tiempo de servicio restante siempre devuelve un algoritmo optimal.

### Teorema 2 (Correctitud del algoritmo *greedy* para scheduling)

El algoritmo *greedy* para **SCHEDULING** es correcto.

#### Prueba.

**ejercicio** Ayuda: se prueba por el absurdo, suponiendo que existe una mejor solución que la encontrada por el algoritmo *greedy*, y se llega a una contradicción. □



### ▶ Implementación:

- ▶ se ordenan los procesos por orden creciente de tiempo de servicio, y se implementa el ciclo *greedy*.
- ▶ como el cuerpo del ciclo *greedy* es de  $\Theta(1)$ , y el ciclo no se repite más de  $n$  veces, el tiempo del algoritmo en general estará dominado por el tiempo del ordenamiento:  $\Theta(n \log n)$ .

(ejercicio)

- ▶ existen numerosas variantes de este problema (con más de un procesos, con límites a la espera de los procesos, con ganancia por la ejecución del proceso antes del límite, etc.), la mayoría de las cuales tienen algoritmos *greedy* correctos.



## Definición del problema

- ▶ los **códigos de Huffman** se usan para comprimir información eficientemente, logrando una reducción del 20% al 90% (dependiendo de los información original)
- ▶ la información es representada como una secuencia de caracteres, donde cada caracter tiene una **frecuencia** conocida
- ▶ suponemos que cada caracter es representado en **binario**



## Definición del problema

- ▶ no todas las codificaciones variables son aceptables. Para que la **decodificación** no produzca ambigüedades, se debe asegurar que *ningún código debe ser prefijo de otro*
- ▶ se puede demostrar siempre se puede alcanzar una compresión optimal usando estos **códigos prefijos**
- ▶ es muy fácil representar los códigos prefijos con un árbol binario en donde cada camino representa el código de la hoja
- ▶ toda codificación optimal es representada por un **árbol completo**, donde cada nodo interno tiene exactamente dos hijos
- ▶ esto significa que si  $C$  son los caracteres, se necesita un árbol binario de  $|C|$  hojas y  $|C| - 1$  nodos internos (**ejercicio**)



## Ejemplo

- ▶ se tienen 6 caracteres:  $a, \dots, f$ . Si se quiere codificar información escrita en estos caracteres con **códigos de longitud fija** se necesitan  $\lceil \log_2 6 \rceil = 3$  bits por código.
- ▶ un archivo de 1.000 caracteres necesitará 3.000 bits
- ▶ si se permiten **códigos de longitud variable**, se pueden asignar códigos cortos a caracteres frecuentes y códigos más largos a caracteres poco frecuentes. Es necesario conocer de antemano la frecuencia con la que aparecen los caracteres
- ▶ por ejemplo:

	a	b	c	d	e	f
Frecuencia (%)	45	13	12	16	9	5
Código variable	0	101	100	111	1101	1100

- ▶ sólo  $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 10 = 2,240$  bits son necesarios con esta codificación



## Definición del problema

- ▶ sea para cada  $c \in C$ ,  $c.freq$  la frecuencia de  $c$ . Si usamos una codificación representada por un árbol  $T$ , entonces sea  $\delta_T(c)$  la profundidad en  $T$  de la hoja con caracter  $c$ , o lo que es lo mismo la cantidad de dígitos de la codificación de  $c$
- ▶ el número de bits necesarios para codificar usando  $T$  es

$$B(T) = \sum_{c \in C} c.freq * \delta_T(c)$$

- ▶ tomaremos a  $B(T)$  como el costo de la codificación  $T$
- ▶ el problema algorítmico **HUFFMAN** consiste entonces en dada una serie de caracteres  $C$  con sus frecuencias, encontrar una codificación  $T$  optimal en  $B(T)$



## Algoritmo de codificación de Huffman

- ▶ Huffman diseñó un algoritmo *greedy* de  $O(n \log n)$  (ejercicio) basado en la E.D. heap

```
n ::= |C|; Q.construirHeap(C)
FOR i ::=1 TO n
  x ::= Q.extraerMin(); y ::= Q.extraerMin()
  z ::= nuevo Nodo
  z.left ::= x; z.right ::= y
  z.freq ::= x.freq+y.freq
  Q.insertar(z)
ENDFOR
RETURN Q
```



## Correctitud

- ▶ sea  $C'$  el alfabeto basado en  $C$  que se obtiene eliminando  $x$  e  $y$ , y agregando un nuevo caracter  $z$  tal que  $z.freq = x.freq + y.freq$ .

### Lema 4

Sea  $T'$  una codificación optimal para  $C'$ . Entonces se puede obtener una codificación optimal  $T$  para  $C$  reemplazando en  $T'$  el nodo hoja de  $z$  por un nodo interno con dos hijos  $x$  e  $y$ .

### Prueba.

Se muestra por contradicción que a partir de  $T''$  obtenido de aplicar el lema 3 a una codificación mejor que  $T$  en  $C$ , se obtiene una codificación mejor que  $T'$ .



## Correctitud

- ▶ sea  $C$  un alfabeto en donde cada  $c \in C$  tiene frecuencia  $c.freq$ ; y sean  $x, y \in C$  los caracteres con menores frecuencias en  $C$ .

### Lema 3

$C$  tiene una codificación prefija optimal en la cual  $x$  e  $y$  son los hermanos de máxima profundidad.

### Prueba.

Sea  $T$  una codificación optimal y  $a, b \in C$  los caracteres hermanos de máxima profundidad en  $T$ . Suponemos  $a.freq \leq b.freq$  y  $x.freq \leq y.freq$ , luego  $x.freq \leq a.freq$  y  $y.freq \leq b.freq$ . Construimos  $T'$  intercambiando en  $T$   $a$  con  $x$ ; y  $T''$  intercambiando en  $T'$   $b$  con  $y$ . Se muestra  $B(T) - B(T') \geq 0$  y  $B(T') - B(T'') \geq 0$ . Como  $T$  es optimal, entonces  $T''$  también.



## Correctitud

### Teorema 5

El algoritmo de Huffman produce una codificación optimal.

### Prueba.

Por inducción en las iteraciones aplicando el lema 4. □

