Cambio de monedas y Algoritmos voraces

Cambio de monedas

Dado un sistema monetario, cómo descomponer una cantidad *M* en la menor cantidad de monedas posibles. Más formalmente:

Entrada

- Serie de denominaciones (valores enteros positivos) $d_1 < d_2 < \ldots < d_N$ que compone el sistema monetario.
- Una cantidad entera positiva *M*.

Salida

Serie r_1, r_2, r_N tal que $\sum_{i=1}^{n} r_i * d_i = M$ minimizando $\sum_{i=1}^{N} r_i$

Ejemplo: En el sistema colombiano, *M* = $1850

1 de 1000 + 1 de 500 + 1 de 200 + 1 de 100 + 1 de 50

Algoritmos voraces

Aunque no es una definición formal, podríamos decir que un algoritmo puede ser considerado como ‘voraz’ si involucra un proceso iterativo que va tomando decisiones miopes con un criterio *greedy* previendo que de esa manera se llega al final a la decisión deseada.

En este contexto miope significa que una decisión se toma con la información que se tiene en ese momento y que, una vez tomada, es irrevocable.

Generalmente, no es complejo definir un algoritmo ‘greedy’. Este no siempre garantiza soluciones óptimas pero en algunos casos si lo hace y con una eficiencia mejor que otras aproximaciones.

Cambio de monedas

¿Cuál sería el criterio *greedy* en este caso?

Tomar la mayor cantidad posible de la mayor denominación

Y ¿Cómo sería el proceso iterativo?

Repetir el procedimiento con la siguiente denominación y la cantidad restante

```
reverse(d)
r = M
for each d
    if (r ≥ d)
        c += r / d
        r = r % d
```

¿Cuál es la eficiencia de este algoritmo? $O(N)$

Cambio de monedas

¿Esta solución es siempre óptima?

No. ¿Qué pasa por ejemplo si d = {11, 5, 1} y M = 15?

¿Para que sistemas monetarios es óptima entonces?

Caso 1: cuando $d = p^i$ para i : 0, 1, 2, … siendo p un numero natural mayor que 1

¿Por qué? Resulta que una propiedad de los números naturales es que si x es un natural entonces se puede expresar como:

$x = r_0 p^0 + r_1 p^1 + r_2 p^2 + \ldots + r_n p^n$ con $n = log_p x$

Para cualquier p natural mayor que 1

Cambio de monedas

Caso 2a: Sistema numérico colombiano

¿Será que nuestros padres de la patria pensaron en el algoritmo greedy a la hora de escoger las denominaciones?

Caso 2b: Sistema numérico estadounidense (mucho más común en plataformas de programación competitiva)

¿Por qué? ¿Qué tienen en común estos sistemas? ¿Se podría generalizar alguna regla diferente al caso 1?

Minimización de sumas

Minimización de sumas

Dados un arreglo *A* de *N* valores numéricos, ¿Cuál es el valor de *X* que minimiza la siguiente suma?

$S = |A_1 - X|^c + |A_2 - X|^c + \ldots + |A_N - X|^c$

Concentrémonos en los casos *c* = 1 y *c* = 2

Ejemplo: *c* = 2 y *A* = [3, 7, 2, 5, 13]

X = 6, *S* = $(3-6)^2 + (7-6)^2 + (2-6)^2 + (5-6)^2 + (13-6)^2 = 76$

¿Cuál sería el criterio *greedy* en este caso?

Escoger la media de *A* para *X*, ¿Por qué?

Minimización de sumas

Como *c* es par, puede quitarse el valor absoluto y expresar *S* como:

$$S = (A_1 - X)^2 + (A_2 - X)^2 + \ldots (A_N - X)^2$$

$$= NX^2 - 2X(A_1 + A_2 + \cdots + A_N) + (A_1^2 + A_2^2 + \cdots A_N^2)$$

Donde la última parte no depende de *X* por lo que puede ignorarse, mientras que el restante $NX^2 - 2XB$ con $\mathrm{B} = (A_1 + A_2 + \cdots + A_N)$ es una parábola hacia arriba cuyas raíces son $r_1 = 0$ y $r_2 = 2B/N$ con lo que por simetría el valor mínimo está ubicado en el punto medio B/N que es precisamente la media de *A*.

¿Cuál es la eficiencia de este algoritmo? $O(N)$

Minimización de sumas

¿Qué sucede cuando *c* = 1?

$$S = |A_1 - X| + |A_2 - X| + \cdots + |A_N - X|$$

Ejemplo: *c* = 1 y *A* = [3, 7, 2, 5, 13]

¿Funciona la media?

X = 5, S = |6-5| + |4-5| + |7-5| + |3-5| + |5-5| = 6

¿Cuál sería entonces el criterio *greedy* en este caso?

Escoger la mediana de *A* para *X* y, en caso que *N* sea par, cualquiera de los valores medios, incluso cualquier valor entre ellos, producirá el mismo *S*

¿Cuál es la eficiencia de este algoritmo? $O(N * \log_2 N)$

Alineación de cadenas binarias

Alineación de cadenas binarias

Dados dos cadenas *A* y *B*, ambas de *N* valores binarios, y una función *f* que le aplica la operación NOT a una sub-cadena (cambia los 0's por 1's y viceversa), ¿cuál es la mínima cantidad de veces que se debe aplicar *f* para transformar *B* en *A*?

Ejemplo:

A = 0 1 1 1 0 1
B = 1 1 0 0 0 1

f(1,4) + *f(2,2)*, ó
f(1,1) + *f(3,4)*

Alineación de cadenas binarias

¿Cuál sería el criterio *greedy* en este caso?

Comenzando a la izquierda de la cadena, aplicar *f* a la sub-cadena más larga de elementos desalineados

Y ¿Cómo sería el proceso iterativo?

Repetir el procedimiento a partir del índice donde terminó el anterior

Alineación de cadenas binarias

```
read A, B
N = |A|
desalineado = False
min = 0
for i = 0 to N-1:
   if A_i ≠ B_i:
      if not desalineado:
         desalineado = True
         min = min + 1
   else:
      desalineado = False
print min
```

¿Cuál es la eficiencia de este algoritmo? $O(N)$

Knapsack fraccional

Problema de la mochila fraccional

Fuente:
https://openclipart.org/detail/313728/thief

También conocido por *Knapsack* por su nombre en inglés consiste en que, dada una mochila con una capacidad máxima de carga W y un conjunto de N objetos cada uno con un valor monetario v_i y un peso w_i, ¿Cómo debe llenarse de forma que se maximice el valor total cargado sin exceder la capacidad W y considerando que los objetos se pueden fraccionar?

Ejemplo: Para $W = 50$

	A	B	C	D
w_i	10	20	30	40
v_i	55	100	75	120

Solución:

A + B + $\frac{1}{2}$ D, con una ganancia de 215

Knapsack fraccional

Preguntémonos primero: Si todos los objetos tuvieran el mismo peso w_i, ¿Cuáles deberían escogerse primero, los de mayor valor v_i o los de menor?

→ Los de mayor valor v_i

Y si todos los objetos tuvieran el mismo valor v_i, ¿Cuáles deberían escogerse primero, los de mayor peso o las de menor?

→Los de menor peso w_i

A partir de la generalización de estos casos especiales, un posible criterio greedy es escoger los objetos con mayor valor y menor peso o, lo que es lo mismo, aquellas con mayor relación valor – peso. Cuando no sea posible un objeto completo, tomar la mayor fracción posible.

¿Cuál es la eficiencia de este algoritmo? $\boldsymbol{O(N * log(N))}$

Cobertura de intervalo

Cobertura de intervalo

Caso particular del problema NP *Set covering*.

Entrada: Dado un intervalo objetivo [*a*, *b*] sobre la recta real y un conjunto de *N* subintervalos [s_i, f_i]

Salida: Mínimo número de subintervalos que, unidos, cubran completamente [*a, b*]

Ejemplo: para el intervalo [1, 10]

	A	B	C	D	E
s	1	2	3	5	6
f	4	6	7	8	10

{A, C, E}

{A, B, E}

Cobertura de intervalo

Dados *N* sub-intervalos, ¿cuántos subconjuntos diferentes pueden haber? $\mathbf{2^N - 1}$

Es decir, una solución por búsqueda exhaustiva sería generar un arreglo de binario *N* elementos y evaluar en $O(N)$ cada una de las $2^N - 1$ posibilidades

¿Se puede hacer mejor?

Para este problema ¿cuál sería la decisión greedy?

Partiendo desde *a*, ¿Qué tal escoger el intervalo que expanda el alcance lo más posible en cada paso, es decir, aquel con un límite derecho mayor?

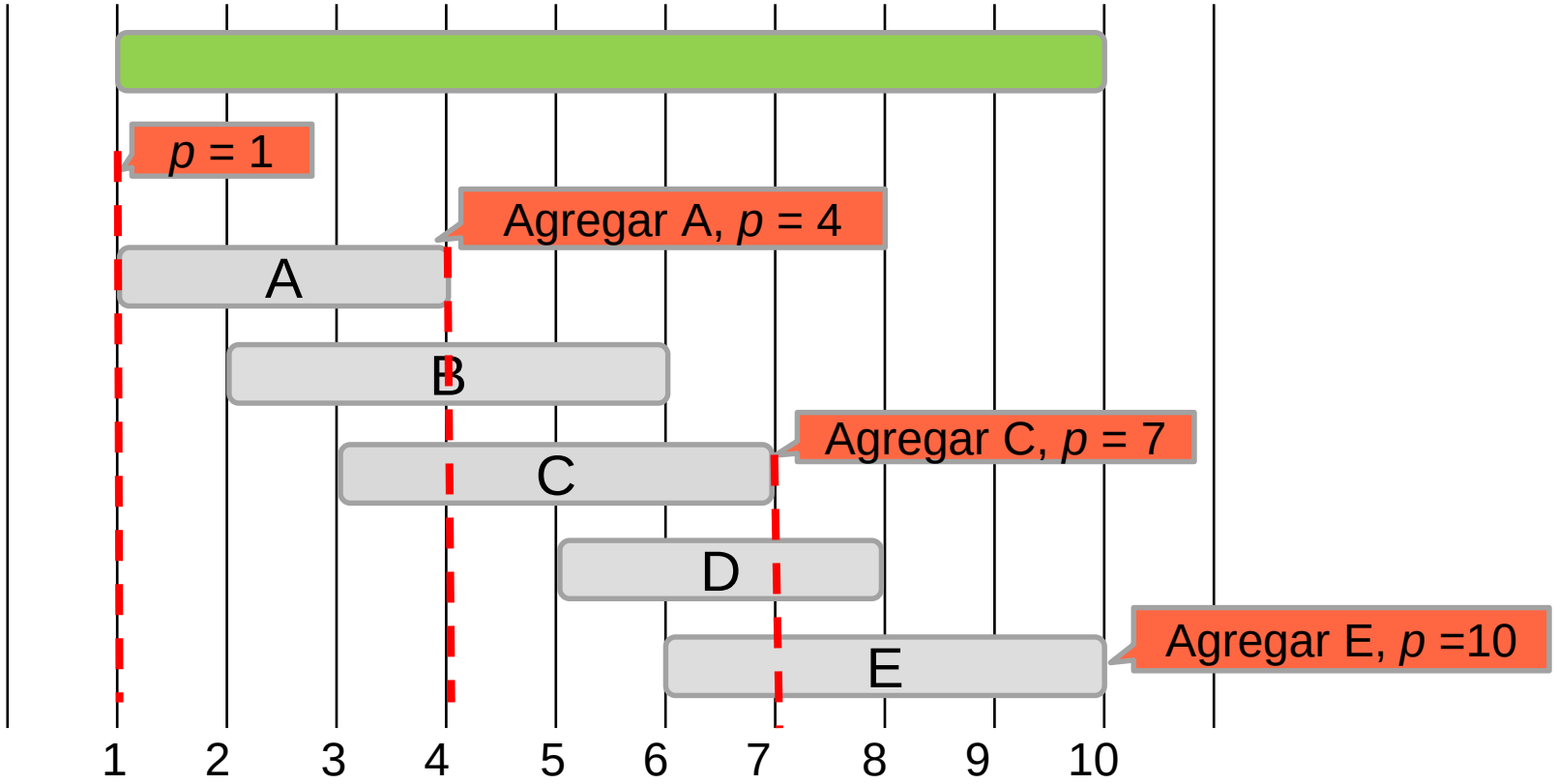
Cobertura de intervalo

Para utilizar esta estrategia debemos:

Ordenar los intervalos por su inicio *si*
p = a
minimo = 0
Mientras *p* < *b*
    Buscar los intervalos tales que $s_i \leq p$
    Si hay por lo menos uno, elegir aquel con mayor f_i
        minimo += 1
        $p = f_i$
    En caso contrario no hay solución
Si *p* < *b*, no hay solución, en caso contrario, imprimir *minimo*

Volviendo al ejemplo para [1, 10]:

	A	B	C	D	E
s	1	2	3	5	6
f	4	6	7	8	10



¿Cuál es la eficiencia de este algoritmo? $\boldsymbol{O(N * log(N))}$

Programación de actividades

Programación de actividades

También conocido como ISMP por *Interval Scheduling Maximization Problem*.

Entrada: Un conjunto de *N* actividades que necesitan de un recurso para ser llevadas a cabo, el cual solo puede atender una al tiempo. Cada actividad tiene un tiempo de inicio s_i y uno de finalización e_i donde $0 \leq s_i < e_i < \infty$

Salida: Un subconjunto actividades mutuamente compatibles que maximiza la cantidad de actividades llevadas a cabo.

Ejemplo:

	A	B	C	D	E	F	G	H
s	1	0	5	3	6	8	8	12
e	4	6	7	9	10	11	12	16

{A, C, F, H}

{A, C, G, H}

Programación de actividades

Dados *N* actividades, ¿cuántos subconjuntos diferentes pueden haber? $\mathbf{2^N - 1}$

Es decir, una solución por búsqueda exhaustiva sería generar un arreglo de binario *N* elementos y evaluar en $O(N)$ cada una de las $2^N - 1$ posibilidades

¿Se puede hacer mejor?

Para este problema ¿cuál sería la decisión greedy?

¿Qué tal escoger la actividad que deje la mayor cantidad
de recursos disponibles para las actividades siguientes,
es decir, aquella con un menor tiempo de finalización?

Programación de actividades

Para utilizar esta estrategia debemos:

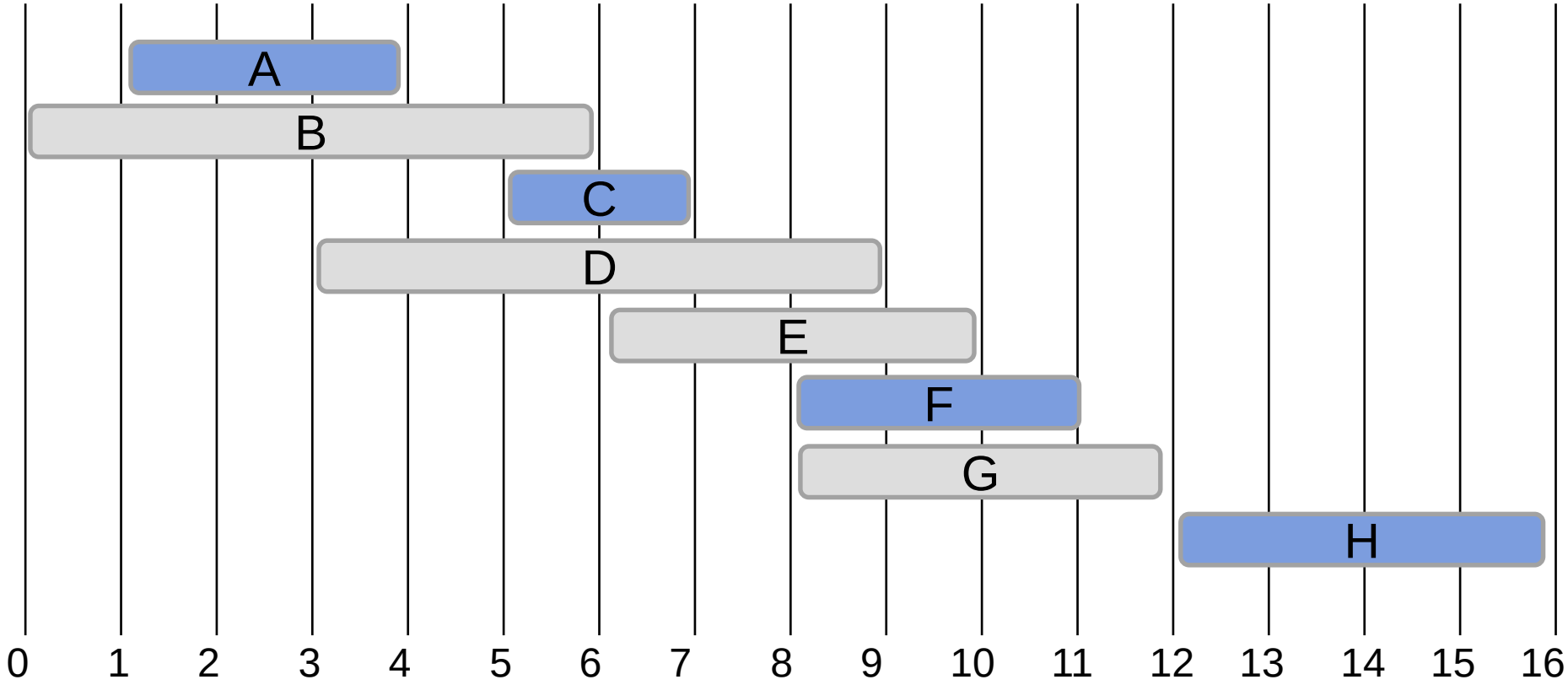
- En caso que ya no lo estén, ordenar las actividades de forma ascendente por tiempo de finalización.
- Escoger la primera actividad
- Escoger sistemáticamente la siguiente actividad compatible según la decisión greedy hasta llegar a la *N*-ésima.

Pero para este problema ¿la intuición empleada es correcta?, es decir, ¿la decisión greedy utilizada guía la estrategia definida hacia una solución óptima?

Resulta que si (ver teorema 16.1 y la prueba correspondiente en el libro guía)

Volviendo al ejemplo:

	A	B	C	D	E	F	G	H
s	1	0	5	3	6	8	8	12
e	4	6	7	9	10	11	12	16



¿Cuál es la eficiencia de este algoritmo? $\boldsymbol{O(N * log(N))}$