

TEMA 7:

ALGORITMOS VORACES

COMPUTABILIDAD Y ALGORITMIA

M. Colebrook Santamaría

J. Riera Ledesma

J. Hernández Aceituno

Objetivos

- Características generales de los algoritmos voraces
- Esquema general de los algoritmos voraces
- Funciones asociadas
- Ejemplo del cambio de moneda
- Ejemplo del problema de la mochila

Introducción

- Los algoritmos **voraces** (*greedy*: codicioso, ávido) son los más fáciles dentro de la familia de técnicas algorítmicas.
- Su enfoque es *miope*: **toman decisiones** basándose en la información disponible en cada momento, sin tener en cuenta el efecto que pueden tener en el futuro.
- Por ello son fáciles de diseñar y desarrollar, y suelen ser **eficientes**.
- Se suelen utilizar para resolver problemas de **optimización**. Ejemplos: búsqueda de la ruta más corta entre dos puntos, búsqueda de la mejor planificación de tareas en un ordenador, etc.

Ejemplo de algoritmo voraz: Cambio de monedas



- Monedas disponibles:
 - 1, 2, 5, 10, 20, 50 céntimos de €.
 - 1 y 2 € (equivalente a 100c y 200c)
- El problema consiste en diseñar un algoritmo para pagar una cierta cantidad usando el menor número posible de monedas.
- Ejemplo: $2,89\text{€} = 2\text{ €} + 50\text{c} + 20\text{c} + 10\text{c} + 5\text{c} + 2\text{c} + 2\text{c}$.
- En total son 7 monedas, que es el menor número posible de monedas para esa cantidad.

Algoritmo para el cambio de monedas

```
función devolver_cambio(n: cantidad): conjunto de monedas {  
  M <- { 200, 100, 50, 20, 10, 5, 2, 1 }  
  S <-  $\emptyset$       // conjunto de la solución  
  suma <- 0      // suma de los elementos de S  
  mientras suma  $\neq$  n hacer {  
    v <- mayor elemento de M tal que suma + v  $\leq$  n  
    si no existe v entonces  
      devolver NO_EXISTE_SOLUCIÓN  
    S <- S  $\cup$  {una moneda de valor v}  
    suma <- suma + v  
  }  
  devolver S  
}
```

Análisis del algoritmo

- Con los valores correctos de las monedas y disponiendo de un suministro ilimitado de cada moneda, el algoritmo produce una **solución óptima**.
- En otro caso, el algoritmo puede **no funcionar** o **no encontrar el valor óptimo**. Por ejemplo: $M=\{1,3,4\}$ y $n=6$, el algoritmo hubiera cogido $S=\{4,1,1\}$, siendo la **solución óptima** $S=\{3,3\}$.
- El algoritmo es **voraz** porque en cada paso selecciona la mayor de las monedas posibles, y **nunca** cambia de opinión (no hace **backtracking**): una vez que una moneda se incluye en la solución, se queda para siempre.
- **Complejidad**: siendo $m=|M|$, tenemos **$O(n \cdot m)$** , aunque se puede reducir a **$O(m)$** si se cambia el esquema.

Algoritmo alternativo para el cambio de monedas

```
función devolver_cambio(n: cantidad): conjunto de monedas {  
  M <- { 200, 100, 50, 20, 10, 5, 2, 1 }  
  S <-  $\emptyset$       // conjunto de la solución  
  suma <- 0      // suma de los elementos de S  
  para v  $\in$  M (de mayor a menor valor) hacer {  
    c <- (n - suma) / v // división entera  
    si c > 0 entonces {  
      S <- S  $\cup$  {c monedas de valor v} // c items de valor v  
      suma <- suma + c * v  
    }  
  }  
  devolver S  
}
```

Características generales de los algoritmos voraces

Los algoritmos voraces disponen de tres conjuntos:

- Un **conjunto (o lista) de candidatos** viables.
- Los candidatos considerados y **seleccionados**.
- Los candidatos considerados y **rechazados**.

y cuatro funciones que calculan:

- si un conjunto de candidatos constituye una **solución**.
- si un cierto conjunto de candidatos es **factible**.
- cuál es el mejor de los candidatos que no han sido ni seleccionados ni rechazados (función de **selección**).
- el valor de la solución (función **objetivo**).

Esquema general de un algoritmo voraz

```
función voraz(C: conjunto): conjunto
{ // C es el conjunto de candidatos
  S <-  $\emptyset$  // conjunto de la solución
  mientras C  $\neq \emptyset$  y no solución(S) hacer {
    x <- seleccionar(C)
    C <- C - {x} // quitamos x de C
    si factible(S  $\cup$  {x}) entonces
      S <- S  $\cup$  {x}
  }
  si solución(S) entonces devolver S
  en otro caso devolver NO_HAY_SOLUCIÓN
}
```

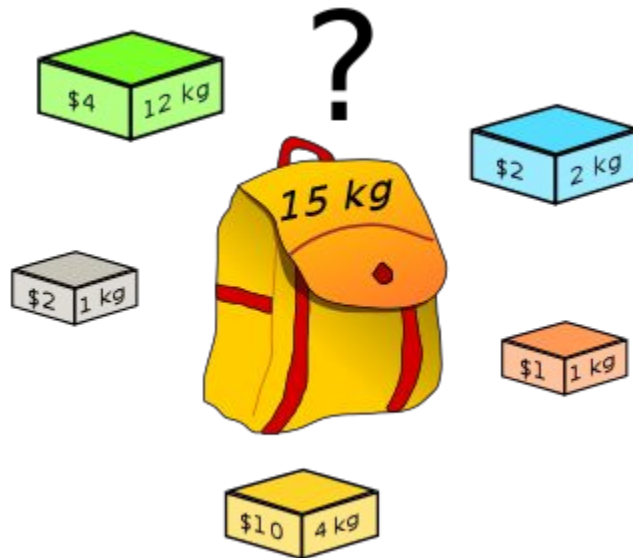
Esquema general aplicado al cambio de monedas

- **Conjunto de candidatos:** es el conjunto de monedas $M=\{1c, 2c, 5c, \dots, 200c\}$, con una cantidad ilimitada de cada moneda.
- **Función solución:** comprueba si el valor de las monedas seleccionadas hasta el momento es *exactamente* el valor buscado.
- **Función de factibilidad:** un conjunto de monedas es factible si su valor total no sobrepasa la cantidad establecida.
- **Función de selección:** escoge la moneda de valor más alto que quede en el conjunto de candidatos.
- **Función objetivo:** cuenta el número de monedas en la solución.

El problema de la mochila (1)

(knapsack problem)

- Tenemos **n objetos** y una mochila.
- Cada objeto $i=1,\dots,n$ tiene un **peso p_i** y un **valor v_i** .
- El peso máximo que la mochila soporta es **P** .
- Objetivo: Llenar la mochila maximizando el valor total de todos los objetos, y respetando la limitación de peso máximo **P** .



El problema de la mochila (2)

- Para simplificar, permitimos que para cada **objeto i** podemos incluir una fracción x_i del mismo, $0 \leq x_i \leq 1$.
- Por tanto, el **objeto i** contribuye al **peso** de la mochila en $x_i \cdot p_i$ unidades, y al **valor** en $x_i \cdot v_i$ unidades.
- El problema se puede definir de esta forma:

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

$$s.a. \sum_{i=1}^n x_i p_i \leq P$$

$$v_i > 0, \quad p_i > 0, \quad 0 \leq x_i \leq 1, \quad i=1, \dots, n$$

Funciones de selección

- Tenemos al menos tres funciones de selección:
 - a) Seleccionar el objeto **más valioso**.
 - b) Seleccionar el objeto **menos pesado**.
 - c) Seleccionar el objeto cuyo **valor por unidad de peso** sea el mayor.
- Las opciones (a) y (b) no generan la solución óptima, pero (c) sí.

Ejemplo

$P = 100$ kg, $n = 5$ objetos					
i:	1	2	3	4	5
v_i:	20 €	30 €	66 €	40 €	60 €
p_i:	10 kg	20 kg	30 kg	40 kg	50 kg
v_i / p_i:	2.0 €/kg	1.5 €/kg	2.2 €/kg	1.0 €/kg	1.2 €/kg

Selección:		x_i					Peso total	Valor total
(a)	Max v_i	0	0	1	0.5	1	100	146
(b)	Min p_i	1	1	1	1	0	100	156
(c)	Max v_i / p_i	1	1	1	0	0.8	100	164

Algoritmo de la mochila

```
función mochila(P, p[1..n], v[1..n]): vector[1..n] {  
  para i <- 1 hasta n hacer x[i] <- 0  
  peso <- 0  
  mientras peso < P hacer {  
    i <- seleccionar el mejor objeto restante  
    si peso + p[i] <= P entonces {  
      x[i] <- 1  
      peso <- peso + p[i]  
    } en otro caso {  
      x[i] <- (P - peso) / p[i]  
      peso <- P  
    }  
  devolver x  
}
```

Análisis del algoritmo

- Si los objetos ya están en orden decreciente de v_i / p_i , el algoritmo voraz requiere un tiempo **$O(n)$** .
- De otro modo, en general la ordenación requerirá un tiempo de cómputo mayor (**$O(n \log n)$** para *heapsort*) y predominará sobre la del algoritmo voraz (reglas de simplificación).

Referencias

- ★ Brassard, G. and Bratley, P. (1998) “Fundamentos de Algoritmia”, *Prentice-Hall*. [**Capítulo 6**]
- ★ Shaffer, C. A. (2013) “Data Structures and Algorithm Analysis”, Edition 3.2 (C++ Version), *Dover Publications*. **Freely** available for **educational** and **non-commercial use** at: people.cs.vt.edu/shaffer/Book/C++3elatest.pdf