

# Algoritmos y Estructuras de Datos II

Algoritmos voraces

9 de mayo de 2016

# Clase de hoy

- 1 Organización de la materia
- 2 Algoritmos voraces
  - Forma general
  - Problema de la moneda
  - Problema de la mochila
  - Árboles generadores de costo mínimo

# Organización de la materia

- cómo vs. qué
- 3 partes
  - 1 análisis de algoritmos
  - 2 tipos de datos
  - 3 técnicas de resolución de problemas
    - divide y vencerás
    - [algoritmos voraces](#)
    - backtracking
    - programación dinámica
    - recorrida de grafos

## Algoritmos Voraces (o Glotones, Golosos) (Greedy)

- Es la técnica más sencilla de resolución de problemas.
- Normalmente se trata de algoritmos que resuelven problemas de **optimización**, es decir, tenemos un problema que queremos resolver de manera **óptima**:
  - el camino más corto que une dos ciudades,
  - el valor máximo alcanzable entre ciertos objetos,
  - el costo mínimo para proveer un cierto servicio,
  - el menor número de billetes para pagar un cierto importe,
  - el menor tiempo necesario para realizar un trabajo, etc.
- Los algoritmos voraces intentan construir la solución óptima buscada **paso a paso**,
- **eligiendo** en cada paso
- la **componente** de la solución
- que **parece** más apropiada.

## Características

- Nunca revisan una **elección** ya realizada,
- confían en haber elegido bien las componentes anteriores.
- Por ello, lamentablemente, no siempre funcionan,
- pero cuando funcionan son muy eficientes.
- Existen varios problemas interesantes que admiten soluciones voraces,
- que como acabamos de decir, resultan muy eficientes.

## Problemas con solución voraz

- Problema de la moneda.
- Problema de la mochila.
- Problema del camino de costo mínimo en un grafo.
- Problema del árbol generador de costo mínimo en un grafo.

## Ingredientes comunes de los algoritmos voraces

- se tiene un problema a resolver de manera **óptima**,
- un conjunto de **candidatos** a integrar la solución,
- los candidatos se van clasificando en 3: los aún no considerados, los **incorporados** a la solución parcial, y los **descartados**,
- existe una función que chequea si los candidatos incorporados ya forman una **solución** del problema (sin preocuparse por si la misma es o no óptima),
- una segunda función que comprueba si un conjunto de candidatos es **factible** de crecer hacia una solución (sin preocuparse por cuestiones de optimalidad),
- finalmente, una tercer función que **selecciona** de entre los candidatos aún no considerados, el más promisorio.

## Receta general de los algoritmos voraces

- Inicialmente ningún candidato ha sido considerado, es decir, ni incorporado ni descartado.
- En cada paso se utiliza la función de **selección** para elegir cuál candidato considerar.
- Se utiliza la función **factible** para evaluar si el candidato considerado se incorpora a la solución o no.
- Se utiliza la función **solución** para comprobar si se ha llegado a una solución o si el proceso de construcción debe continuar.



## Forma general

```
fun greedy(C) ret S
    {C: conjunto de candidatos, S: solución a construir}
    S:= {}
    do S no es solución → c:= seleccionar de C
        C:= C-{c}
        if S∪{c} es factible → S:= S∪{c} fi
    od
end fun
```

Lo más importante es el criterio de selección.

## Problema de la moneda

- Tenemos una cantidad infinita de monedas de cada una de las siguientes denominaciones:
  - 1 peso,
  - 50 centavos,
  - 25 centavos,
  - 10 centavos,
  - 5 centavos
  - y 1 centavo.
- Se desea pagar un cierto monto de manera exacta.
- Se debe determinar la manera de pagar dicho importe exacto con la menor cantidad de monedas posible.

## Solución al problema de la moneda

- Seleccionar una moneda de la mayor denominación posible que no exceda el monto a pagar,
- utilizar exactamente el mismo algoritmo para el importe remanente.

Criterio de selección claramente establecido.

## Algoritmo voraz

```
fun cambio(m: monto) ret S: conjunto de monedas
  var c,s: monto
  C:= {100, 50, 25, 10, 5, 1}
  S:= {}
  s:= 0
  do s  $\neq$  m  $\rightarrow$  c:= mayor elemento de C tal que s+c  $\leq$  m
    S:= S $\cup$ {una moneda de denominación c}
    s:= s+c
  od
end fun
```

# Algoritmo voraz

Versión más detallada

```
fun cambio(m: monto) ret S: array[1..6] of nat
  S[1]:= m div 100
  m:= m mod 100
  S[2]:= m div 50
  m:= m mod 50
  S[3]:= m div 25
  m:= m mod 25
  S[4]:= m div 10
  m:= m mod 10
  S[5]:= m div 5
  m:= m mod 5
  S[6]:= m
end fun
```

# Algoritmo voraz

Detallado pero genérico

{Pre:  $d[1] \geq d[2] \geq \dots \geq d[n]$  }

```
fun cambio(d:array[1..n] of nat, m: monto) ret S: array[1..n] of nat
  for i:= 1 to n do
    S[i]:= m div d[i]
    m:= m mod d[i]
  od
end fun
```

## Sobre este algoritmo

- El orden del algoritmo es  $n$ , es decir, el número de denominaciones.
- Si el arreglo de denominaciones no está ordenado requiere  $n \log n$  ordenarlo y luego  $n$  más el algoritmo, en total es  $n \log n$ .
- No siempre funciona, depende del conjunto de denominaciones.
- Para un conjunto razonable, funciona.

## Conjunto de denominaciones para el que no funciona

- Sean 4, 3 y 1 las denominaciones y sea 6 el monto a pagar.
- El algoritmo voraz intenta pagar con una moneda de denominación 4, queda un saldo de 2 que solamente puede pagarse con 2 monedas de 1, en total, 3 monedas.
- Pero hay una solución mejor: dos monedas de 3.
- De todas formas, el algoritmo anda bien para todas las denominaciones de uso habitual.



## Problema de la mochila

- Tenemos una mochila de capacidad  $W$ .
- Tenemos  $n$  objetos de valor  $v_1, v_2, \dots, v_n$  y peso  $w_1, w_2, \dots, w_n$ .
- Se quiere encontrar la mejor selección de objetos para llevar en la mochila.
- Por mejor selección se entiende aquélla que totaliza el mayor valor posible sin que su peso exceda la capacidad  $W$  de la mochila.
- Para que el problema no sea trivial, asumimos que la suma de los pesos de los  $n$  objetos excede la capacidad de la mochila, obligándonos entonces a seleccionar cuáles cargar en ella.

## Criterio de selección

¿Cómo conviene seleccionar un objeto para cargar en la mochila?

- El más valioso de todos.
- El menos pesado de todos.
- Una combinación de los dos.

## Análisis del primer criterio de selección

El más valioso primero

- Razonabilidad: el objetivo es cargar la mochila con el mayor valor posible, escogemos los objetos más valiosos.
- Falla: puede que al elegir un objeto valioso dejemos de lado otro apenas menos valioso pero mucho más liviano.
- Ejemplo: Mochila de capacidad 10, objetos de valor 12, 11 y 9, y peso 7, 5 y 5.
- De elegir primero el de mayor valor (12) ocuparíamos 7 de los 10 kg de la mochila, no quedando lugar para otro objeto.
- En cambio, de elegir el de valor 11, ocuparíamos solamente 5 kg quedando 5 kg para el de valor 9, totalizando un valor de 20.

## Análisis del segundo criterio de selección

El menos pesado primero

- Razonabilidad: hay que procurar aprovechar la capacidad de la mochila, escogemos los objetos más livianos.
- Falla: puede que al elegir un objeto liviano dejemos de lado otro apenas más pesado pero mucho más valioso.
- Ejemplo: Mochila de capacidad 13, objetos de valor 12, 11 y 7, y peso 6, 6 y 5.
- De elegir primero el de menor peso (5) obtendríamos su valor (7) más, en el mejor de los casos, 12, totalizando  $12+7=19$ .
- En cambio, de elegir los dos de peso 6, no se excede la capacidad de la mochila y se totaliza un valor de 23.

## Análisis del tercer criterio de selección

### Combinando ambos criterios

- Debemos asegurarnos de que cada kg utilizado de la mochila sea aprovechado de la mejor manera posible: que cada kg colocado en la mochila valga lo más posible.
- Criterio: elegir el de mayor valor relativo (cociente entre el valor y el peso): dicho cociente expresa el valor promedio de cada kg de ese objeto.
- Falla: puede que al elegir un objeto dejemos de lado otro de peor cociente, pero que aprovecha mejor la capacidad.
- Ejemplo: Mochila de capacidad 10, objetos de valor 12, 11 y 8, y peso 6, 5 y 4.
- El criterio elige al que pesa 5, ya que cada kg de ese objeto vale más de 2. Pero convenía elegir los otros dos.

# Problema de la mochila

## Versión simplificada

- El problema de la mochila no admite solución voraz.
- Se simplifica permitiendo **fraccionar** objetos.
- Ahora sí el tercer criterio funciona.
- (En el ejemplo anterior, elegimos primero el que vale 11 y luego  $5/6$  del que vale 12 obteniendo como valor total  $11 + 10 = 21$ ).

# Algoritmo voraz

{Pre:  $\sum_{i=1}^n w[i] \geq W$ }

**fun** mochila(v: **array**[1..n] **of** valor, w: **array**[1..n] **of** peso, W: peso)  
**ret** s: **array**[1..n] **of** real

**var** weight: peso; c: **nat**

**for** i:= 1 **to** n **do** s[i]:= 0 **od**

weight:= 0

**do** weight  $\neq$  W  $\rightarrow$  c:= tal que s[c] = 0  $\wedge$  v[c]/w[c] máximo

**if** weight + w[c]  $\leq$  W  $\rightarrow$  s[c]:= 1

weight:= weight + w[c]

weight + w[c]  $>$  W  $\rightarrow$  s[c]:= (W-weight)/w[c]

weight:= W

**fi**

**od**

**end fun**

## Sobre este algoritmo

- Si los objetos ya están ordenados según su cociente valor/peso, el orden del algoritmo es  $n$ , es decir, el número de objetos.
- Si los objetos no están ordenado según su cociente valor/peso, requiere  $n \log n$  ordenarlo y luego  $n$  más el algoritmo, en total es  $n \log n$ .
- Si los objetos en total exceden muy largamente la capacidad de la mochila, en vez de ordenar puede convenir utilizar una cola de prioridades, en cuyo caso el orden es  $n$ .
- funciona siempre que esté permitido fraccionarse objetos.



# Algoritmo voraz si los objetos están ordenados

```
{Pre:  $\sum_{i=1}^n w[i] \geq W \wedge \forall i. v[i]/w[i] \geq v[i+1]/w[i+1]}$ 
fun mochila(v: array[1..n] of valor, w: array[1..n] of peso, W: peso)
    ret s: array[1..n] of real

    var weight: peso; c: nat
    for i:= 1 to n do s[i]:= 0 od
    weight:= 0
    c:= 1
    do weight + w[c]  $\leq$  W  $\rightarrow$  s[c]:= 1
        weight:= weight + w[c]
        c:= c+1
    od
    s[c]:= (W-weight)/w[c]
end fun
```

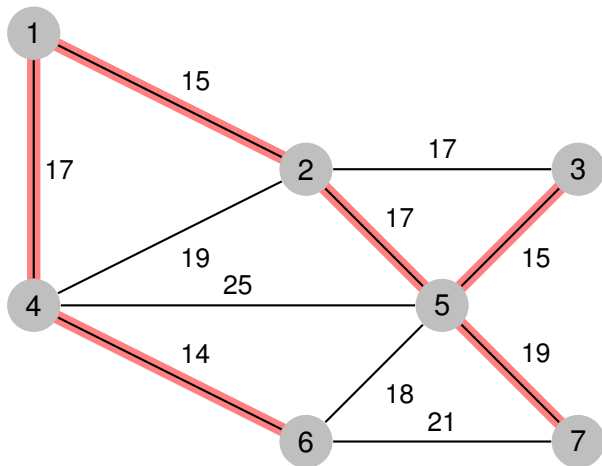
## Árbol generador de costo mínimo

- Sea  $G = (V, A)$  un grafo conexo no dirigido con un costo no negativo asociado a cada arista.
- Se dice que  $T \subseteq A$  es un árbol generador (intuitivamente, un tendido) si el grafo  $(V, T)$  es conexo y no contiene ciclos.
- Su costo es la suma de los costos de sus aristas.
- Se busca  $T$  tal que su costo sea mínimo.

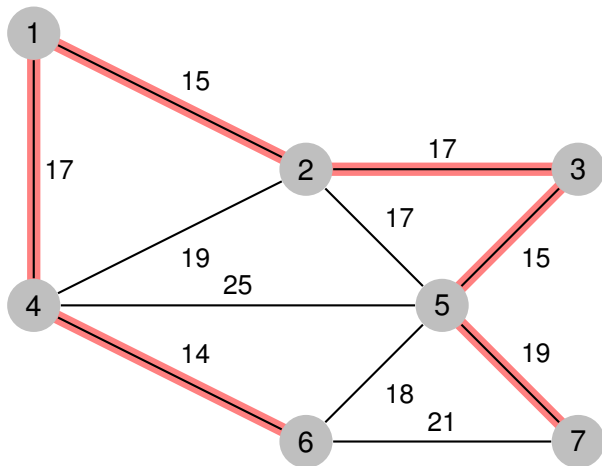
## Árbol generador de costo mínimo

- El problema de encontrar un árbol generador de costo mínimo tiene numerosas aplicaciones en la vida real.
- Cada vez que se quiera realizar un tendido (eléctrico, telefónico, etc) se quieren unir distintas localidades de modo que requiera el menor costo en instalaciones (por ejemplo, cables) posible.
- Se trata de realizar el tendido siguiendo la traza de un árbol generador de costo mínimo.

# Ejemplo



# Ejemplo

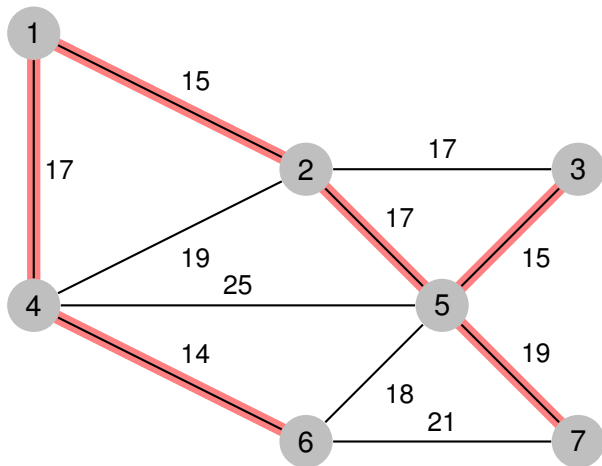


## Dos estrategias

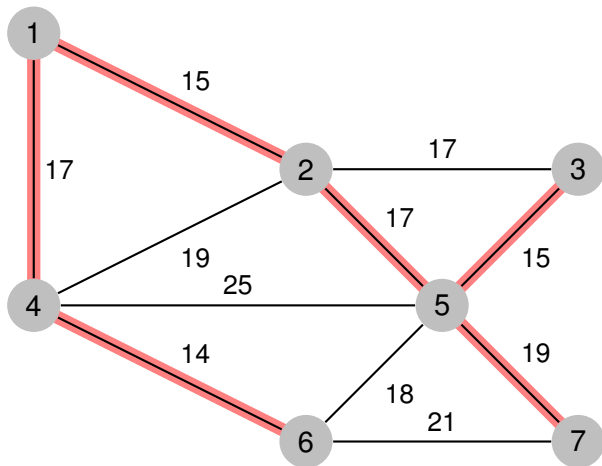
Hay dos grandes ideas de cómo resolverlo:

- La de Prim: se parte desde un vértice origen y se va extendiendo el tendido a partir de ahí:
  - en cada paso se une el tendido ya existente con alguno de los vértices aún no alcanzados, seleccionando la arista de menor costo capaz de incorporar un nuevo vértice
- La de Kruskal: se divide el grafo en distintas componentes (originariamente una por cada vértice) y se van uniendo componentes,
  - en cada paso se selecciona la arista de menor costo capaz de unir componentes.

# Algoritmo de Prim



# Algoritmo de Kruskal





## Implementación del Algoritmo de Prim

```
fun Prim( $G=(V,A)$  con costos en las aristas,  $k: V$ )  
    ret  $T$ : conjunto de aristas  
  
    var  $c$ : arista  
     $C := V - \{k\}$   
     $T := \{\}$   
    do  $n-1$  times  $\rightarrow$   
         $c :=$  arista  $\{i, j\}$  de costo mínimo tal que  $i \in C$  y  $j \notin C$   
         $C := C - \{i\}$   
         $T := T \cup \{c\}$   
    od  
end fun
```

donde  $n = |V|$ . La condición del ciclo podría reemplazarse por  $|T| < n - 1$  o  $C \neq \{\}$ , entre otras.