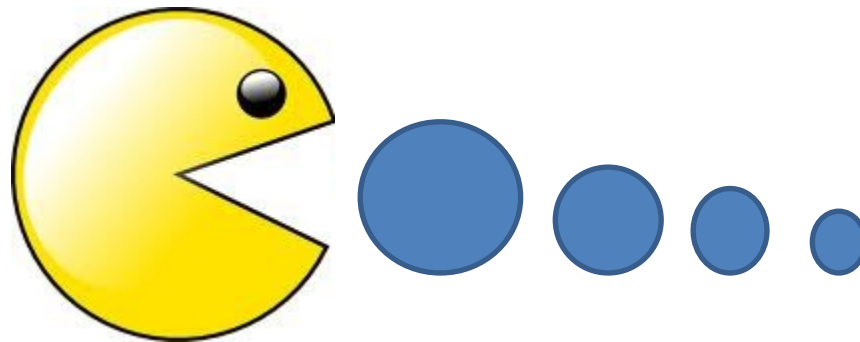
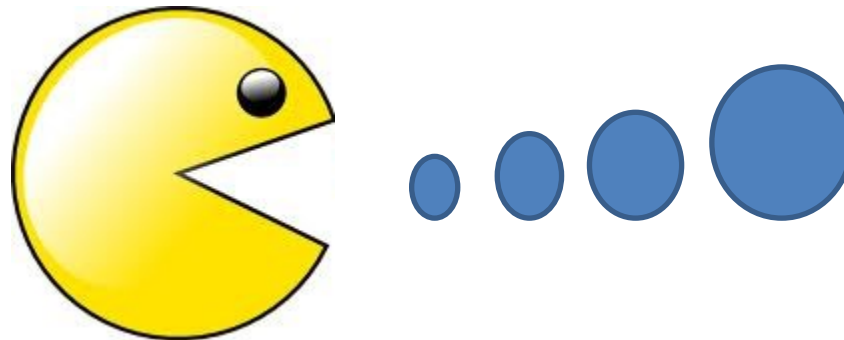


Prácticas Tema 2:

Programación Voraz



ó



1. Programación Voraz

- Todos los problemas consisten en:
 - Partir del subconjunto vacío
 - Seleccionar, **uno a uno**, un subconjunto de elementos dada cierta información de dichos elementos.
 - El elemento a seleccionar en cada paso es el **mejor posible**. Si seleccionarlo provocaría una solución **no factible**, se descarta y se considera el siguiente mejor.
 - **Se para cuando:**
 - se obtiene la solución óptima ó
 - ningún candidato hace factible a la solución

1. Programación Voraz

- **Propiedades** (o limitaciones):
 - La información disponible de cada elemento elegible:
 - Se conoce a priori.
 - Es estática.
 - Un elemento seleccionado no será descartado
 - En cada paso se selecciona el mejor elemento:
 - Más 'grande' ó
 - Menos 'costoso'

1. Programación Voraz

- En cada problema, tenemos que identificar los siguientes **elementos característicos**:
 1. Conjunto de candidatos
 2. Orden de los candidatos
 3. Función de factibilidad
 4. Función de selección
 5. Función de solución
 6. *Contraejemplo para mostrar que no siempre encuentra la solución óptima.*

2. Elementos Característicos

- Identificar los elementos característicos para el Problema I:

*Devolver un conjunto S que represente el cambio (cantidad n),
utilizando el menor número de monedas posibles.
Con monedas ilimitadas y de distintos valores.*

Instancia del problema como ejemplo:

- $n=5$.
- Monedas= {1c, 5c, 10c, 20c, 50c, 1, 2}

2. Elementos Característicos

1. Conjunto de Candidatos:

- Enumerar el conjunto de elementos disponibles antes de iniciar el proceso de búsqueda.
 - a) Si no sabemos el valor concreto de cada elemento, se describen.
 - b) Si sabemos el valor, se muestra el subconjunto.
- En el problema del cambio:
 - a) $\{x \in \text{MODENAS}\}$ ó ‘todas las monedas disponibles’
 - b) $\{1c, 5c, 10c, 20c, 50c, 1, 2\}$

2. Elementos Característicos

2. Orden de los candidatos:

- Ordenar los candidatos de mejor a peor, según la información que tenemos antes de comenzar la búsqueda.
- El sentido de *mejor* es dependiente del problema (más alto, más rápido, menos caro, más corto,...)
- En el problema del cambio:
 - a) $\{x_i \in \mathbf{MODENAS}, \mid x_i > x_{i+1}\}$ ó ‘monedas ordenadas de mayor a menor’
 - b) $\{2, 1, 50c, 20c, 10c, 5c, 1c\}$

2. Elementos Característicos

3. Función de Factibilidad

- Comprobar si la solución formada al añadir un elemento nuevo hace que ésta se convierta en errónea.
- En el problema del cambio:
 - a) $\sum_{i=1}^s s_i \leq n$ ó ‘la suma del valor de las monedas debe ser menor o igual que la cantidad a devolver’.
 - b) La suma del valor de las monedas debe ser menor o igual que 5.

2. Elementos Característicos

4. Función de selección:

- Decir qué elemento se selecciona en cada paso. Siempre será el mejor factible.
- En el problema del cambio, el mejor es la moneda de mayor valor porque reduce la cantidad de monedas necesarias:
 - a) X_i , si la solución creada es factible. Si no, X_{i+1} . Comenzando con $i=1$ y respecto al conjunto de candidatos ordenado.
 - b) 2; si no es factible, 1; si no 50c; ... hasta 1c.

2. Elementos Característicos

5. Función de solución:

- Tras añadir un nuevo elemento al conjunto de seleccionados, hay comprobar si hemos encontrado la solución óptima.
- En el problema del cambio:
 - a) $\sum_{i=1}^s s_i = n$ o 'la suma de las monedas seleccionadas es justo el cambio a devolver'.
 - b) Suma de las monedas seleccionadas = 5

2. Elementos Característicos

6. Contraejemplo:

- Solo si hay que demostrar que el algoritmo no encuentra siempre la solución óptima.
- En el problema del cambio:
 - $n=8$
 - Monedas={5,4,1}
 - devolvería $S=\{5\}$

3. Implementación

- Tras identificar los elementos característicos, escribir **pseudocódigo**.
- En el problema del cambio:

```
function Conjunto devolver(Conjunto c, int n){
    Conjunto S={};
    int suma=0;
    int i=0;
    while(suma!=n){ //mientras haya soluciones factibles
        x=c[i];
        if(suma+x)<=n then{ // es factible
             $S \leftarrow S \cup x$ ;
            suma+=x;
        }
        else i++; //no es factible
    }
}
```

4. Ejercicios

- **1. Cambio con monedas finitas:**
 - El array c contiene tantos valores repetidos como monedas hay de dicho valor.
 - Dentro del **if**, añadir $c \leftarrow c - x$
 - Cambiar while por: **while** ($s \neq n$ AND $i < \text{length}(c)$)

4. Ejercicios

- 2. (igual que el anterior)
- 3. Llenado de CD: devolver conjunto S
 - Cjto. Candidatos: $\text{Ficheros} = \{l_1, l_2, \dots, l_n\}$
 - Orden: $\text{Ficheros} = \{l_i\} / l_i \leq l_{i+1}$
 - Func. Factibi.: $\sum_{i=1}^S l_i \leq d$
 - Func. Selección: l_i , comenzando con $i=0$. Si $S \leftarrow S \cup l_i$ no es factible, $i++$ hasta que lo sea.
 - Func. Solución:
 $\sum_{i=1}^S l_i = d$, ó *no quedan l_i que hagan S factible*
 - Es óptimo porque si de cada candidato solo hay 1 ejemplar, y comenzamos por los más pequeños, siempre se mete la mayor cantidad posible de ficheros antes de llenar el CD.

4. Ejercicios

- Pseudocódigo llenarCD:

function Conjunto **llenarCD**(Conjunto ficheros, int d){

ficheros \leftarrow ordenarMenorAMayor(ficheros);

Conjunto S={};

int suma=0;

int i=0;

while(i<length(ficheros)){ *//función solución*

 x=ficheros[i];

if(suma+size(x)<=d){ *//es factible*

 S \leftarrow S U x;

 suma+=size(x)

 }

 i++;

}

return S;

}

4. Ejercicios

- **4. Problema de la mochila fraccionada**

- Cjto. Candidatos: $\text{Objetos} = \{(\text{size}_i, \text{value}_i)\}$
- Orden: mayor a menor $\text{ratio} = \text{value}_i / \text{size}_i$
- Factibilidad: $\sum_{i=1}^S \text{size}_i \leq d$
- Selección: $\text{Objetos}[i]$, comenzando con $i=0$. Si no es factible, incrementar $i++$. Si no se selecciona ninguno, coger mayor fracción posible de $\text{Objetos}[0]$.
- Solución: $\sum_{i=1}^S \text{size}_i = 0$, ó *no quedan factibles*.
- Es óptimo, por el mismo razonamiento del anterior (mayor valor, menor tamaño en todo momento)

4. Ejercicios

- Pseudocódigo Mochila:

```
function conjunto llena_Mochila(double[][2] objetos, double d){  
    objetos=ordenar_por_ratio(objetos);  
    Conjunto S={};  
    int i=0, beneficio=0;  
    while(i<length(objetos) AND d>0){ //función solución  
        if(d-objetos[i][0]>=0){ //es factible  
            S←S U objetos[i][0];  
            d-=objetos[i][0];  
            beneficio+=objetos[i][1];  
        }  
        else{ // no es factible  
            fraccion=d/objetos[i][0];  
            S← S U (fraccion x objetos[i][0]);  
            d=0;  
            beneficio+= (fraccion x objetos[i][1]);  
        }  
        i++;  
    }  
}
```

4. Ejercicios

- **5. Árbol de cubrimiento mínimo** (siguiendo **Kruskal**: de vacío, ir eligiendo aristas más pequeñas hasta conectar todos los nodos).
 - Cjto. Candidatos: $T = \{\text{arista}_i\}$
 - Orden: $T = \{\text{arista}_i\} / \text{arista}_i \leq \text{arista}_{i+1}$
 - Factib: la nueva arista conecta 2 nodos no conectados aún
 - Selección: arista factible con menor longitud
 - Solución: $N-1$ aristas seleccionadas (pq significa que hemos conectado N nodos).

4. Ejercicios

- Pseudocódigo :

```
Conjunto Kruskal(Grafo G){  
    N=numeroNodos(G);  
    Conjunto T[]=get_aristas(G);  
    T←ordenarPorLongitud(T,G);  
    Conjunto S={};  
    int i=0;  
    while(length(S)≤N-1){ //función solución  
        int a=T[i];  
        if(conecta2NodosNoConectados(S,G,a)) //es factible  
            S←S U a;  
        i++;  
    }  
    return S;  
}
```

4. Ejercicios

- **6. Camino Mínimo desde un Nodo, con DIJKSTRA:**
 - Cjto. Candidatos: : $T=\{arista_i\}$
 - Orden: $T=\{arista_i\} / arista_i \leq arista_{i+1}$
 - Selección: se añade:
 - la primera arista que conecte un nuevo nodo desde cualquier nodo ya conectado, ó
 - la arista que minimice la suma de pesos hasta un nodo ya conectado, y se descarta la antigua arista → semi-voraz
 - Factibilidad: conecta un nuevo nodo o minimiza el coste a un nodo ya conectado
 - Solución: camino construido entre origen y destino
 - Podéis ver un ejemplo animado en:
 - <http://www.youtube.com/watch?v=UG7VmPWkJmA>

4. Ejercicios II

- **7. Array ordenado I**

- Cjto. Candidatos: $\{x,y\} \in \mathbf{Z}$
- Orden: $\text{Numeros} = \{x_i\} / x_i \leq x_{i+1}$
- Factibilidad: $x*y < 0$ AND $x+y > 0$ AND $(x,y) \notin S$
- Selección: $\arg \min x, \arg \max y \ (x,y) \in \text{Numeros} /$
 $x \in \mathbf{Z}^-$ AND $y \in \mathbf{Z}^+$
- Solución: no quedan candidatos factibles

4. Ejercicios II

Pseudocódigo:

```
function void Ordenadol(int[] numeros){  
    numeros=ordenar(numeros);  
    i=0;  
    j=length(numeros);  
    S={}  
    while((i<j) AND (numeros[i]<0)){  
        if(numeros[i]+numeros[j] > 0{  
            S ← S U (numeros[i],numeros[j]);  
            j--;  
        }  
        i++; //si no se encuentra pareja solo se incrementa i  
    }  
}
```

4. Ejercicios II

- **8. Array ordenado II**

- Cjto. Candidatos: $\{x, y\} \in \mathbf{Z}$
- Orden: $\text{Numeros} = \{x_i\} / x_i \leq x_{i+1}$
- Factibilidad: $x + y < 0$ AND $(x, y) \notin \mathbf{S}$
- Selección: $\arg \min x, \arg \max y \quad (x, y) \in \text{Numeros} /$
 $x \in \mathbf{Z}^-$ AND $y \in \mathbf{Z}$ (y positivo o negativo)
- Solución: no quedan candidatos factibles

4. Ejercicios II

Pseudocódigo:

```
function void OrdenadoII(int[] numeros){  
    numeros=ordenar(numeros);  
    i=0;  
    j=length(numeros);  
    S={}  
    while((i<j) AND (numeros[i]<0)){  
        if(numeros[i]+numeros[j] < 0{  
            S ← S U (numeros[i],numeros[j]);  
            i++;  
        }  
        j--; //si no se encuentra pareja solo se incrementa j  
    }  
}
```

4. Ejercicios II

- **9. Array ordenado III**

- Cjto. Candidatos: $\{x,y\} \in \mathbf{Z}$
- Orden: $\text{Numeros} = \{x_i\} / x_i \leq x_{i+1}$
- Factibilidad: $x*y > 0$ AND $(x,y) \notin S$
- Selección: $(x,y) \in \text{Numeros} / x,y \in \mathbf{Z}^-$ OR $x,y \in \mathbf{Z}^+$
- Solución: no quedan candidatos factibles

4. Ejercicios II

Pseudocódigo:

```
function void OrdenadoIII(int[] numeros){  
    numeros=ordenar(numeros);  
    i=0;  
    S={}  
    while((i < length(numeros)){  
        if(numeros[i]*numeros[i+1] > 0{  
            S ← S U (numeros[i],numeros[i+1]);  
            i++;  
        }  
        i++;  
    }  
}
```

4. Ejercicios II

- **10. Array ordenado IV**

- Cjto. Candidatos: $\{x,y\} \in \mathbf{Z}$
- Orden: $\text{Numeros} = \{x_i\} / x_i \leq x_{i+1}$
- Factibilidad: $x*y < 0$ AND $(x,y) \notin S$
- Selección: $(x,y) \in \text{Numeros} / x \in \mathbf{Z}^-$ AND $y \in \mathbf{Z}^+$
- Solución: no quedan candidatos factibles

4. Ejercicios II

Pseudocódigo:

```
function void OrdenadoV(int[] numeros){  
    numeros=ordenar(numeros);  
    i=0;  
    j=length(numeros)  
    S={}  
    while((numeros[i]<0) AND numeros[j]>0){  
        S ← S U (numeros[i],numeros[j]);  
        i++;  
        j--;  
    }  
}
```

4. Ejercicios II

- **11. Array ordenado V**
 - Cjto. Candidatos: $\{0, y\} \in \mathbf{Z}$
 - Orden: $\text{Numeros} = \{x_i\} / x_i \leq x_{i+1}$
 - Factibilidad: $x * y = 0 \ (x, y) \notin \mathbf{S}$
 - Selección: $(x, y) / x = 0$
 - Solución: no quedan ceros en el array ordenado

4. Ejercicios II

Pseudocódigo:

```
function void OrdenadoIV(int[] numeros){  
    numeros=ordenar(numeros);  
    i=0;  
    j=length(numeros)  
    S={}  
    while((numeros[i]<=0) AND i<j ){  
        if(i*j==0) S ← S U (numeros[i],numeros[j]);  
        i++;  
        j--;  
    }  
}
```

4. Ejercicios II

• 12. Subconjuntos de medias próximas

- *Minimizar Z significa minimizar para cada subconjunto S la diferencia entre la media de los valores de S y la media del conjunto original D .*
- Candidatos: $D = \{d_1, d_2, \dots, d_n\}$
- Orden: menor a mayor $|\mu - d_i|$
- Selección: *en cada iteración, meter a cada subconjunto S el elemento más pequeño según el orden. Si quedan elementos en D , entonces meter los mayores (así se minimiza la media).*
- Factibilidad: $\arg \min_{x,y} Z$
- Solución: no quedan más elementos

4. Ejercicios II

pseudocódigo:

```
function int[][] medias(int[n] D, int k, double  $\mu$ ){ // k numero de subconjuntos a crear
    ordenar(D); // menor a mayor  $|\mu - d_i|$ 
    int[][] S;
    while(D $\neq\emptyset$ ){
        i=0, j=0;
        while(i<k AND D $\neq\emptyset$ ){
            S[i] $\leftarrow$ S[i] U D[i];
            D $\leftarrow$  D - D[i];
            i++;
        }
        while(j<k AND D $\neq\emptyset$ ){
            S[n-j] $\leftarrow$ S[n-j] U D[n-j];
            D $\leftarrow$  D-D[n-j]
            j++;
        }
    }
    return S;
}
```