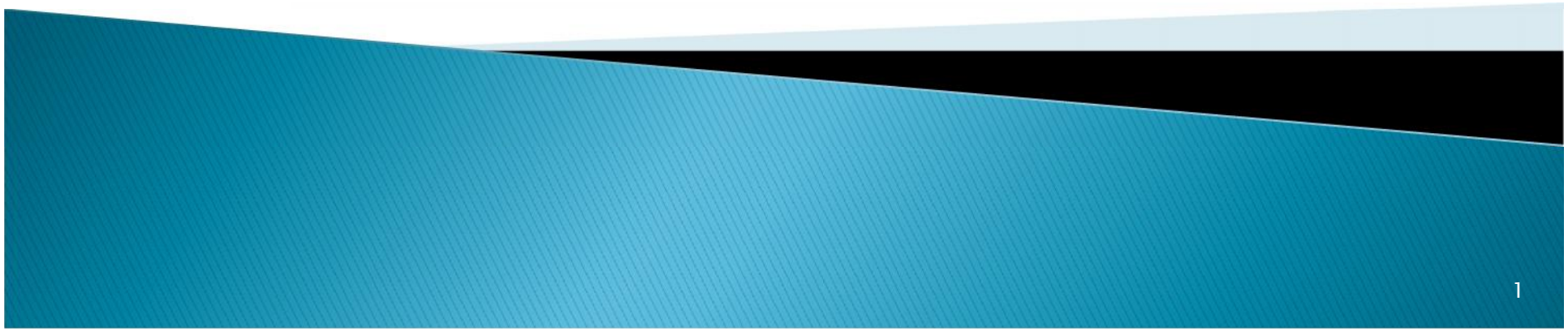
Algoritmia y Complejidad

Tema 2. Algoritmos Voraces.



1.- Introducción.

- Es uno de los esquemas más simples y al mismo tiempo de los más utilizados.
- Típicamente se emplea para resolver problemas de optimización:
  - Existe una entrada de tamaño n que son los candidatos a formar parte de la solución;
  - Existe un subconjunto de esos n candidatos que satisface ciertas restricciones: se llama solución factible;
  - Hay que obtener la solución factible que maximice o minimice una cierta función objetivo: se llama solución óptima.

1.- Introducción.

Esquema genérico:

```
función voraz(C: conjunto): conjunto
    {C es el conjunto de candidatos}
        S:= ∅
        mientras que C≠∅ y no solución(S)
                x:=elemento de x que maximiza seleccionar(C)
                C:=C-{x}
                si factible{SU{x}} entonces S:= SU{x}
        si solución(S) entonces devolver S
        sino devolver “no hay soluciones”
```

2.- Ejemplo. Problema del cambio de monedas.

- Se trata de devolver una cantidad de euros con el menor número posible de monedas.
- Se parte de:
  - Un conjunto de tipos de monedas válidas, de las que se supone que hay cantidad suficiente para realizar el desglose, y de
  - Un importe a devolver.

2.- Ejemplo. Problema del cambio de monedas.

Elementos fundamentales del esquema:

- Conjunto de candidatos: cada una de las monedas de los diferentes tipos que se pueden usar para realizar el desglose del importe dado.
- Solución: un conjunto de monedas devuelto tras el desglose y cuyo valor total es igual al importe a desglosar.
- Completable: la suma de los valores de las monedas escogidas en un momento dado no supera el importe a desglosar.
- Función de selección: elegir si es posible la moneda de mayor valor de entre las candidatas.
- Función objetivo: número total de monedas utilizadas en la solución (debe minimizarse).

2.- Ejemplo. Problema del cambio de monedas.

- Solución:

función devolver cambio(n): conjunto de monedas

{Da el cambio de n unidades utilizando el menor número posible de monedas. La constante C especifica las monedas disponibles}

const C=[100,25,10,5,1}

S≠Ø {S es un conjunto que contendrá la solución}

s=0 {s es la suma de los elementos de S}

mientras s≠n hacer

x:=el mayor elemento de C tal que s+x≤n

si no existe ese elemento entonces devolver “no encuentro solución”

S=SU{una moneda de valor x}}

s=s+x

devolver S

3.- Ejemplo. Problema de la mochila.

- Descripción (*the knapsack problem*):
  - disponemos de una mochila que soporta un peso máximo W
  - existen n objetos que podemos cargar en la mochila, cada objeto tiene un peso w_i y un valor v_i
  - el objetivo es llenar la mochila maximizando el valor de los objetos que transporta

    suponemos que los objetos se pueden romper, de forma que podemos llevar una fracción x_i ($0 \leq x_i \leq 1$) de un objeto
- Matemáticamente:
  - Maximizar $\Sigma x_i v_i$ (valor de la carga)con la restricción $\Sigma x_i w_i \leq W$ (peso de la carga menor que el peso total)
  - donde $v_i > 0$, $w_i > 0$ y $0 \leq x_i \leq 1$ para $1 \leq i \leq n$
- Aplicaciones: empresa de transporte, ...

3.- Ejemplo. Problema de la mochila.

- El pseudocódigo de nuestro algoritmo será:

```
función mochila (w[1..n],v[1..n],W):matriz[1..n]
    {Inicialización}
    para i=1 hasta n hacer x[i]:=0
    peso=0
    {bucle voraz}
    mientras peso<N hacer
            i:=el mejor elemento restante
            si peso+w[i]≤W entonces
                    x[i]:=1
                    peso:=peso+w[i]
            sino
                    x[i]:=(W-peso)/w[i]
                    peso:=W
     devolver x
```

3.- Ejemplo. Problema de la mochila.

- ¿Mejor objeto restante? → Ejemplar de problema: n = 5,W = 100

	1	2	3	4	5	
v_i	20	30	66	40	60	
w_i	10	20	30	40	50	$(\sum_{i=1}^{n} w_i > W)$

3.- Ejemplo. Problema de la mochila.

- Selección:
- 1. ¿Objeto más valioso? ↔ v_i max
- 2. ¿Objeto más ligero? ↔ w_i min
- 3. ¿Objeto más rentable? ↔ v_i/w_i max

	1	2	3	4	5	
v_i	20	30	66	40	60	
w_i	10	20	30	40	50	
v_i/w_i	2,0	1,5	2,2	1,0	1,2	Objetivo $(\sum_{i=1}^{n} x_i v_i)$
x_i (v_i max)	0	0	1	0,5	1	146
x_i (w_i min)	1	1	1	1	0	156
x_i (v_i/w_i max)	1	1	1	0	0,8	164

3.- Ejemplo. Problema de la mochila.

Cálculo del ritmo de crecimiento

Implementación directa:

- El bucle requiere como máximo n (num. obj.) iteraciones: O(n) (una para cada posible objeto en el caso de que todos quepan en la mochila)
- La función de selección requiere buscar el objeto con mejor relación valor/peso: O(n)
- Coste del algoritmo: $O(n) \cdot O(n) = O(n^2)$

Implementación preordenando los objetos:

- Ordena los objetos de mayor a menor relación valor/peso (existen algoritmos de ordenación O(n log n))
- Con los objetos ordenados la función de selección es O(1)
- Coste del algoritmo: $O(\max(O(n \log n), O(n) \cdot O(1))) = O(n \log n)$

4.- Árboles de recubrimiento mínimo

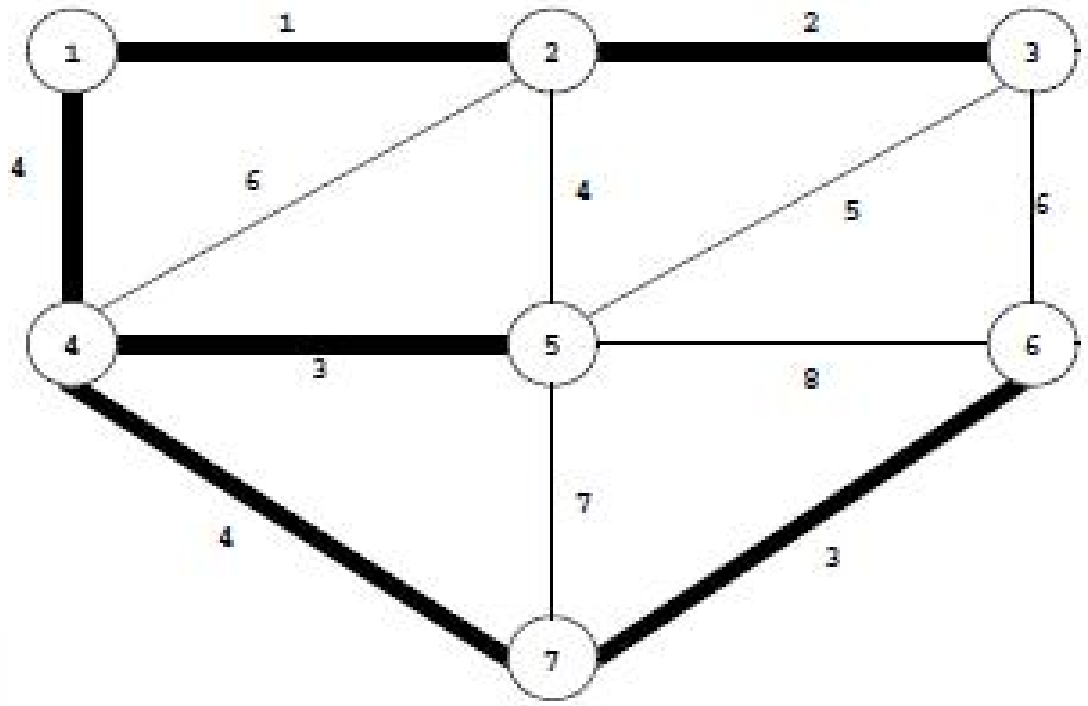
- Objetivo: dado un grafo, obtener un nuevo grafo que sólo contenga las aristas imprescindibles para una optimización global de las conexiones entre todos los nodos
  - "optimización global": algunos pares de nodos pueden no quedar conectados entre ellos con el mínimo coste posible en el grafo original
- Aplicación: problemas que tienen que ver con distribuciones geográficas
  - conjunto de computadores distribuidos geográficamente en diversas ciudades de diferentes países a los que se quiere conectar para intercambiar datos, compartir recursos, etc.;
  - se pide a las compañías telefónicas respectivas los precios de alquiler de líneas entre ciudades
  - asegurar que todos los computadores pueden comunicar entre sí, minimizando el precio total de la red

4.- Árboles de recubrimiento de coste mínimo

- Algoritmo voraz:
  - Función objetivo: minimizar la longitud del árbol de recubrimiento.
  - Candidatos: las aristas del grafo.
  - Función solución: árbol de recubrimiento de longitud mínima.
  - Función factible: Conjunto de aristas que no contiene ciclos.
  - Función de selección: varía con el algoritmo;
    -Seleccionar la arista de menor peso que aún no ha sido seleccionada y que no forme un ciclo (Algoritmo Kruskal).
    - Seleccionar la arista de menor peso que aún no ha sido seleccionada y que forme un árbol junto con el resto de aristas seleccionadas (Algoritmo Prim).

4.1- Algoritmo de Kruskal

- Seleccionar la arista de menor peso que:
  - aún no haya sido seleccionada
  - y que no conecte dos nudos de la misma componente conexa, es decir, que no forme un ciclo



4.1- Algoritmo de Kruskal

Paso	Arista considerada	Componentes conexas
-	-	{1} {2} {3} {4} {5} {6} {7}
1	(1,2)	{1,2} {3} {4} {5} {6} {7}
2	(2,3)	{1,2,3} {4} {5} {6} {7}
3	(4,5)	{1,2,3} {4,5} {6} {7}
4	(6,7)	{1,2,3} {4,5} {6,7}
5	(1,4)	{1,2,3,4,5} {6,7}
6	(2,5)	forma ciclo
7	(4,7)	{1,2,3,4,5,6,7}

4.1– Algoritmo de Kruskal

función Kruskal (G=<N,A>: grafo, longitud: $A \rightarrow R^+$): conjunto de aristas

{Inicialización}
Ordenar A por longitudes crecientes
n:= el número de nodos que hay en N
T≠∅ {contendrá las aristas del árbol de recubrimiento mínimo}
Iniciar n conjuntos, cada uno de los cuales contiene un elemento distinto de N
{bucle voraz}
repetir
e:= (u,v) {arista más corta, aún no considerada}
compu:=buscar(u)
compv:=buscar(v)
si compu≠compv entonces
fusionar(compu,compv)
T:=T∪{e}
hasta que T contenga n-1 aristas
devolver T

4.1- Algoritmo de Kruskal

Kruskal calcula el árbol expandido mínimo.

Análisis con $|N| = n$ ^ $|A| = a$

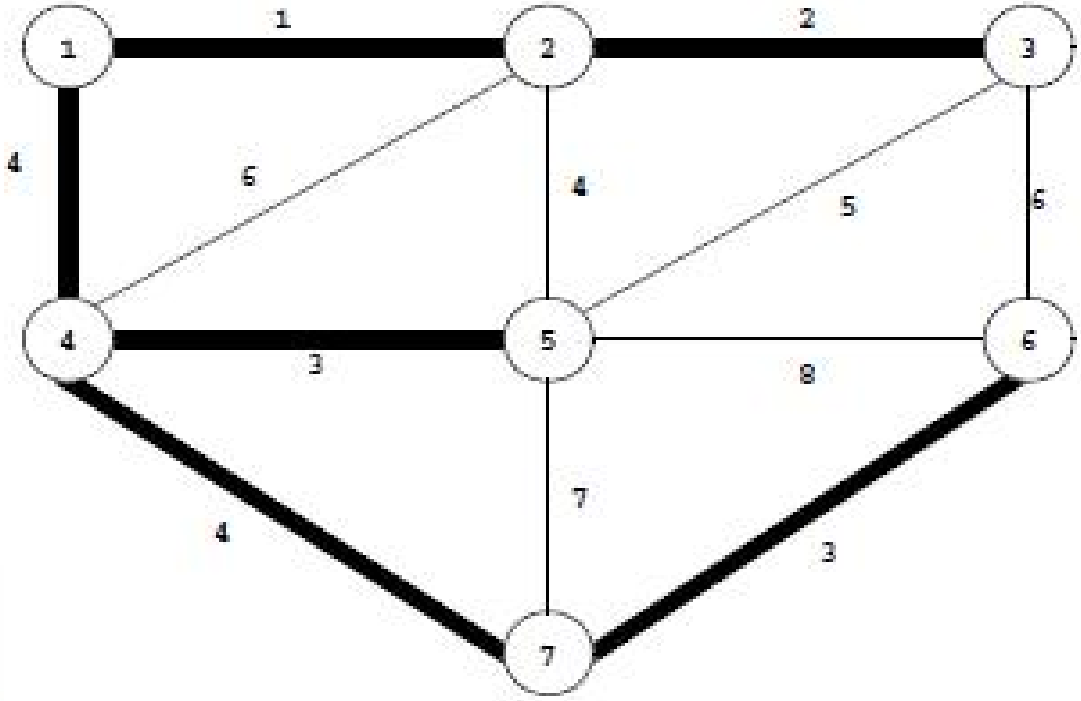
- Ordenar A: $O(a \log a) \equiv O(a \log n)$ ya que $n-1 \leq a \leq n(n-1)/2$
- Inicializar n conjuntos disjuntos: O(n)
- Para las operaciones buscar y fusionar: O(a log n)
- Para las demás operaciones: O(a).

$$T(n) = O(a \log n)$$

4.2- Algoritmo de Prim

Comenzando por un nodo cualquiera, seleccionar la arista de menor peso que:

- aún no haya sido seleccionada
- y que forme un árbol junto con el resto de aristas seleccionadas



4.2- Algoritmo de Prim

- Kruskal es un bosque que crece, en cambio
- Prim es un único árbol que va creciendo hasta alcanzar todos los nodos y calcula el árbol expandido mínimo.
- Ejemplo:

paso	selección	B
ini	-	1
1	(1,2)	1,2
2	(2,3)	1,2,3
3	(1,4)	1,2,3,4
4	(4,5)	1,2,3,4,5
5	(4,7)	1,2,3,4,5,7
6	(7,6)	1,2,3,4,5,6,7 $= N$

4.2- Algoritmo de Prim

```
función Prim (L[1..n, 1..n]): conjunto de aristas
    {Inicialización: solo el nodo 1 se encuentra en B}
    T≠∅ {contendrá las aristas del árbol de recubrimiento mínimo}
    para i:=2 hasta n hacer
        más próximo[i]:=1
        distmin[i]:=L[i,1]
    {bucle voraz}
    repetir n-1 veces
        min:=∞
        para j:=2 hasta n hacer
            si 0≤distmin[j]<min entonces
                min:=distmin[j]
                k:=j
        T:=TU{más próximo[k],k}
        distmin[k]:=-1
        para j:=2 hasta n hacer
            si L[j,k]<distmin[j] entonces
                distmin[j]:=L[j,k]
                más próximo[j]:=k
        devolver T
```

4.2- Algoritmo de Prim

- Análisis:

  inicialización = O(n)

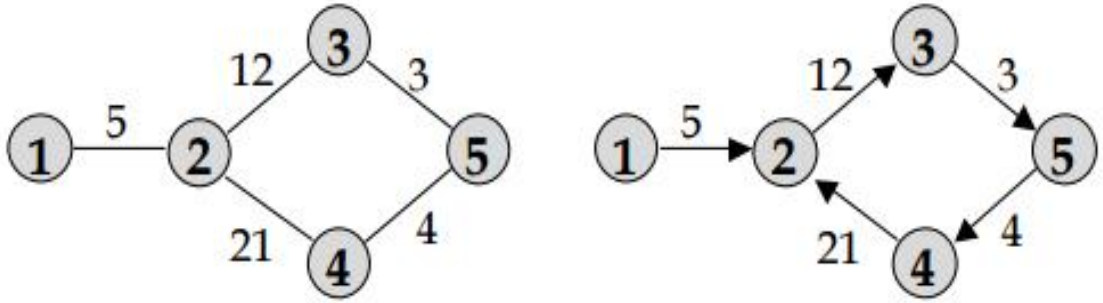
  bucle voraz: n−1 iteraciones, cada para anidado de O(n). Total: T(n) = $O(n^2)$

- Con estructuras de datos más eficientes (montículo) se podría mejorar a O (a log n), igual que Kruskal.

	Prim $\Theta(n^2)$	Kruskal $\Theta(m \log n)$
Grafo denso: $m \to n(n-1)/2$	$\Theta(n^2)$	$\Theta(n^2 \log n)$
Grafo disperso: $m \to n$	$\Theta(n^2)$	$\Theta(n \log n)$

5.- Caminos mínimos en grafos.

- Grafos etiquetados con pesos no negativos.



- Búsqueda de caminos de longitud mínima.
  - Longitud o coste de un camino: suma de los pesos de las aristas que lo componen
  - Cálculo de la longitud mínima de los caminos existentes entre dos vértices dados

5.1.- Algoritmo de Dijkstra.

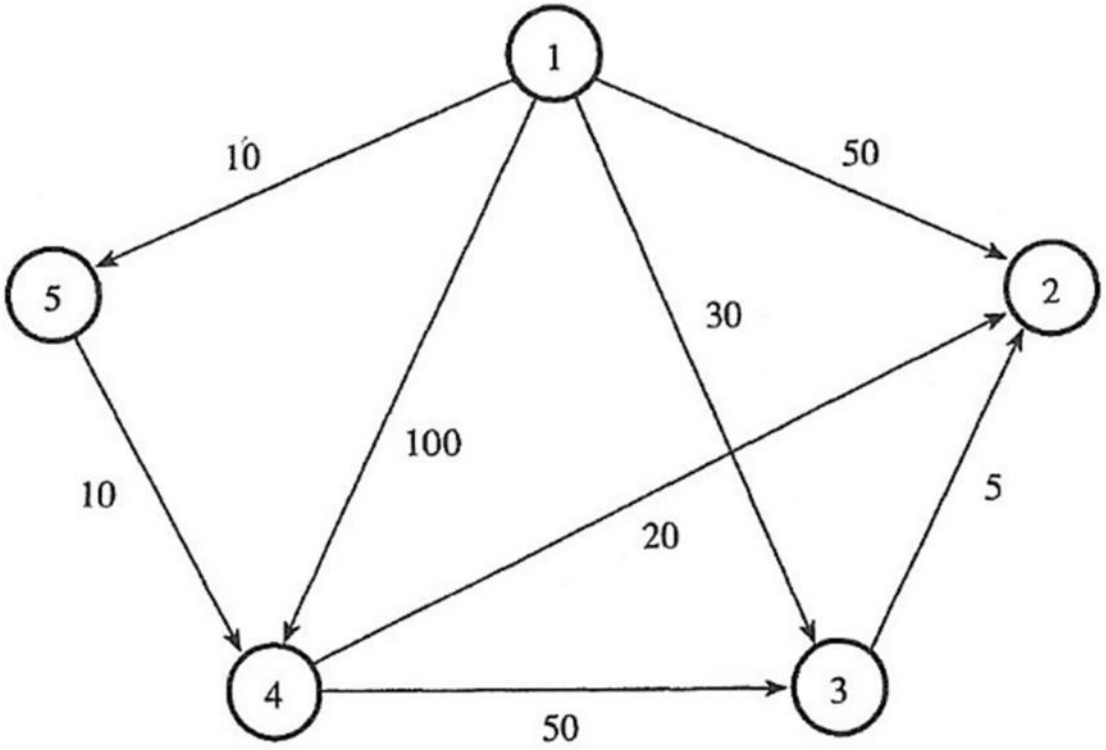
- Para grafos dirigidos (la extensión a no dirigidos es inmediata)
- Genera uno a uno los caminos de un nodo v al resto por orden creciente de longitud
- Usa un conjunto de vértices donde, a cada paso, se guardan los nodos para los que ya se sabe el camino mínimo
- Devuelve un vector indexado por vértices: en cada posición w se guarda el coste del camino mínimo que conecta v con w
- Cada vez que se incorpora un nodo a la solución se comprueba si los caminos todavía no definitivos se pueden acortar pasando por él

5.1– Algoritmo de Dijkstra.

```
función Dijkstra (L[1..n, 1..n]): matriz[2..n]
        matriz D[2..n]
    {Inicialización}
    C:={2,3,..,n}{S=N\C solo existe implícitamente}
    para i:=2 hasta n hacer D[i]:=L[1,i]
    {bucle voraz}
    repetir n-2 veces
        v:=algún elemento de C que minimiza D[v]
        C:=C\{v} {e implícitamente S:=S\{v}}
        para cada w∈C hacer
            D[w]:=min(D[w],D[v]+L[v,w])
        devolver D
```

5.1- Algoritmo de Dijkstra.

- Ejemplo:



5.1- Algoritmo de Dijkstra.

- Ejemplo:

paso	selección	C	D[2]	D[3]	D[4]	D[5]
ini	-	2, 3, 4, 5	50	30	100	10
1	5	2, 3, 4	50	30	20	10
2	4	2, 3	40	30	20	10
3	3	2	35	30	20	10

Tabla: Evolución del conjunto C y de los caminos mínimos.

5.1- Algoritmo de Dijkstra.

- Dijkstra encuentra los caminos mínimos desde el origen hacia los demás nodos del grafo.

- **Análisis**: $|N| = n$, $|A| = m$, $L[1..n, 1..n]$

  Inicialización = $O(n)$

  ¿Selección de v? = $O(n^2)$

  (Es un bucle para anidado)

  $T(n) = O(n^2)$

- Mejora: si el grafo es disperso ($m << n^2$), utilizar listas de adyacencia ya que nos ahorramos el bucle para anidado.

6- Planificación.

- Presentamos dos tipos de problemas relativos a planificar tareas en una sola máquina.
  - Minimizar el tiempo medio que invierte cada tarea en el sistema.
  - Las tareas tienen un plazo fijo de ejecución, y cada tarea aporta unos ciertos beneficios solo si esta acabada al llegar su plazo.
- Objetivo:
  - Maximizar la rentabilidad.

6.1– Minimización del tiempo en el sistema

- Un único servidor tiene que dar servicio a n clientes. El tiempo requerido por cada cliente se conoce de antemano: el cliente i requerirá un tiempo t_i para $1 \leq i \leq n$.
- El objetivo es minimizar el tiempo invertido por cada cliente en el sistema y por consiguiente minimizar el tiempo total invertido en el sistema por todos los clientes.
- $T = \sum$ (tiempo en el sistema para el cliente i)

6.1– Minimización del tiempo en el sistema

- Supongamos que tenemos tres clientes con $t_1=5$, $t_2=10$ y $t_3=3$. Existen seis ordenes de servicio posibles.

Orden	T	
1 2 3:	$5 + (5 + 10) + (5 + 10 + 3) = 38$	
1 3 2:	$5 + (5 + 3) + (5 + 3 + 10) = 31$	
2 1 3:	$10 + (10 + 5) + (10 + 5 + 3) = 43$	
2 3 1:	$10 + (10 + 3) + (10 + 3 + 5) = 41$	
3 1 2:	$3 + (3 + 5) + (3 + 5 + 10) = 29$	← óptimo
3 2 1:	$3 + (3 + 10) + (3 + 10 + 5) = 34$	

6.2- Planificación con plazo fijo.

- Tenemos que ejecutar un conjunto de n tareas, cada una de las cuales requiere un tiempo unitario. En cualquier instante podemos ejecutar únicamente una tarea. La tarea i produce unos beneficios $g_i>0$ solo en el caso de que sea ejecutada en un instante anterior a d_i.
- Por ejemplo, con n=4 y los valores siguientes:

i	1	2	3	4
g_i	50	10	15	30
d_i	2	1	2	1

- Las planificaciones que hay que considerar y los beneficios correspondientes son:

6.2- Planificación con plazo fijo.

Secuencia	Beneficio	Secuencia	Beneficio	
1	50	2,1	60	
2	10	2,3	25	
3	15	3,1	65	
4	30	4,1	80	← óptimo
1,3	65	4,3	45	

- La secuencia 3,2 no se considera porque la tarea 2 se ejecutaría en el instante t=2, después de su plazo que es $d_2=1$. Para maximizar nuestros beneficios deberíamos ejecutar la planificación 4,1.
- Se dice que un conjunto de tareas es factible si existe al menos una sucesión que permite que todas las tareas del conjunto se ejecuten antes de sus respectivos plazos.

6.2- Planificación con plazo fijo.

- Un algoritmo voraz consiste en construir la planificación paso a paso, añadiendo en cada paso la tarea que tenga el mayor valor de g_i entre las que aun no se hayan considerado, siempre y cuando el conjunto de tareas seleccionadas siga siendo factible.
- En el ejemplo seleccionamos primero la tarea 1. Después la tarea 4; el conjunto {1,4} es factible porque se puede ejecutar en el orden 4,1. El conjunto {1,3.4} no es factible, por tanto se rechaza al tarea 3. El conjunto {1,2,4} tampoco es factible y se rechaza la tarea 2. Nuestra solución, en este caso optima, es ejecutar el conjunto de tareas {1,4}, que solo se puede efectuar en el orden 4,1.
- Sea J un conjunto de k tareas. Supongamos que las tareas están numeradas de tal forma que $d_1 \leq d_2 \leq \ldots \leq d_k$. Entonces el conjunto J es factible si y solo si la secuencia 1,2,...k es factible.

6.2- Planificación con plazo fijo.

```
función secuencia (d[0..n]): k,matriz[1..k]
        matriz j[0..n]
    {La planificación se construye paso a paso en la matriz j. La
    variable k dice cuántas tareas están ya en la planificación}
    d[0]:=j[0]:=0 {centinelas}
    k:=j[1]:=1 {la tarea 1 siempre se selecciona}
    {bucle voraz}
    para i:=2 hasta n hacer {orden decreciente de g}
        r:=k
        mientras d[j[r]]>max(d[i],r) hacer r:=r-1
        si d[i]>r entonces
                para m:=k paso -1 hasta r+1 hacer j[m+1]:=j[m]
                j[r+1]:=i
                k:=k+1
        devolver k,j[1..k]
```

6.2- Planificación con plazo fijo.

- Las k tareas de la matriz j están por orden creciente de plazo. Cuando se esta considerando la tarea i, el algoritmo comprueba si se puede insertar en j en el lugar oportuno sin llevar alguna tarea que ya este en j mas allá de su plazo. De ser así, se acepta i, sino se rechaza.
- Ejemplo:

i	1	2	3	4	5	6
g_i	20	15	10	7	5	3
d_i	3	1	1	3	1	3

6.2- Planificación con plazo fijo.

- Análisis del algoritmo
- La ordenación de tareas por orden decreciente de beneficio requiere un tiempo que está en O(n logn). El caso peor es cuando clasifica las tareas por orden decreciente de plazos, y cuando todas ellas tienen cabida en la planificación. En este caso, cuando se está considerando la tarea i el algoritmo examina las k=i-1 tareas que ya están planificadas, para encontrar un lugar para el recién llegado, y después desplaza todas un lugar. Hay $\sum$ pasadas por el bucle **mientras** y $\sum$ pasadas por el bucle **para** interno
- Tiempo: $O(n^2)$

6.2- Planificación con plazo fijo.

```
función secuencia2 (d[0..n]): k,matriz[1..k]
        matriz j,F[0..n]
        p=min(n,max{d[i]|1≤i≤n})
        para i:=0 hasta p hacer
                j[i]:=0
                F[i]:=1
                iniciar el conjunto {i}
    {bucle voraz}
    para i:=1 hasta n hacer {orden decreciente de g}
        k:=buscar(min(p,d[i]))
        m:=F[k]
        si m≠0 entonces
                j[m]:=i
            l:=buscar(m-1)
            F[k]:=F[l]
            fusionar(k,l) {el conjunto resultante tiene la etiqueta k o l}
    k:=0
    para i:=1 hasta p hacer
        si j[i]>0 entonces
                k:=k+1
                j[k]:=j[i]
        devolver k,j[1..k]
```

6.2- Planificación con plazo fijo.

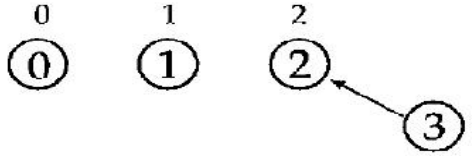
- Si se nos da el problema con las tareas ya ordenadas por beneficios decrecientes, tal que es posible obtener una secuencia optima llamando al algoritmo anterior, entonces la mayor parte del conjunto se invertirá en manipular conjuntos disjuntos.
- Hay que ejecutar como máximo 2n operaciones *buscar* y n operaciones *fusionar,* luego el tiempo requerido esta en $O(n\alpha(2n,n))$ donde α es la función de crecimiento lento.
- Si las tareas están en un orden arbitrario, primero tendremos que ordenarlas, y la obtención de la secuencia inicial requiere un tiempo $O(n \log n)$

6.2- Planificación con plazo fijo.

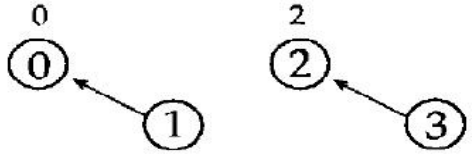
- Volviendo al ejemplo de las seis tareas:

Iniciación: $p = \text{mín}(6, \text{máx}(d_i)) = 3$

Intento 1: $d_1 = 3$, se asigna la tarea 1 a la posición 3

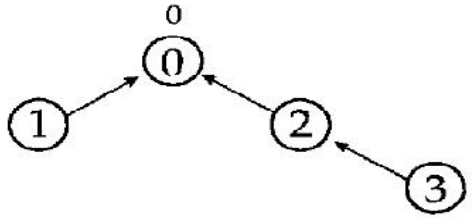


Intento 2: $d_2 = 1$, se asigna la tarea 2 a la posición 1



Intento 3: $d_3 = 1$ no hay posiciones libres disponibles porque el valor de F es 0

Intento 4: $d_4 = 3$, se asigna la tarea 4 a la posición 2



Intento 5: $d_5 = 1$, no hay posiciones libres disponibles

Intento 6: $d_6 = 3$, no hay posiciones libres disponibles

Secuencia óptima: 2, 4, 1; valor = 42

Figura 6.12. Ilustración del algoritmo rápido