



# ALGORÍTMICA

## Capítulo 3. Algoritmos Greedy

### Ejercicios prácticos

# Objetivos de las prácticas

- Con estas prácticas se persigue:
  1. Apreciar la utilidad de los algoritmos greedy para resolver problemas de forma muy eficiente, en algunos casos obteniendo soluciones óptimas y en otros soluciones cercanas a las óptimas.
  2. Constatar la utilidad del enfoque greedy en problemas que se planteen sobre grafos.
  3. Comprobar la utilidad de las heurísticas en Algorítmica
  4. Trabajar comprometidamente en equipo
  5. Aprender a expresar en público las ventajas, inconvenientes y alternativas empleadas, para lograr la solución alcanzada

# Contenedores en un barco

- Se tiene un buque mercante cuya capacidad de carga es de  $K$  toneladas y un conjunto de contenedores  $c_1, \dots, c_n$  cuyos pesos respectivos son  $p_1, \dots, p_n$  (expresados también en toneladas).
- Teniendo en cuenta que la capacidad del buque es menor que la suma total de los pesos de los contenedores:
  - Diseñar un algoritmo que maximice el número de contenedores cargados. Demostrar su optimalidad.
  - Diseñar un algoritmo que intente maximizar el número de toneladas cargadas.

# Contenedores en un barco

- Algoritmo maximizando el número de contenedores cargados
- El algoritmo greedy para el primer caso simplemente escoge en cada momento el contenedor todavía no cargado en el barco que tenga el menor peso. O dicho de otra forma, selecciona los contenedores en orden no decreciente de peso.
- El algoritmo sería pues:

```
Ordenar de menor a mayor el vector de pesos p[];  
i=1;  
fin=false;  
sum=0;  
while (not fin && i<=n) {  
    if sum+p[i] <= k {  
        seleccionar c[i];  
        i=i+1;  
        sum=sum+p[i];  
    }  
    else fin=true;  
}
```

Your company name!

# Contenedores en un barco

- Demostración de la optimalidad del algoritmo
- Sea  $T = \{c_1, \dots, c_n\}$  y supongamos sin pérdida de generalidad que  $p_1 \leq p_2 \leq \dots \leq p_n$ .
- La solución que proporciona el algoritmo greedy está formada por el conjunto  $S = \{c_1, c_2, \dots, c_m\}$  ( $|S| = m$ ) de modo que

$$\sum_{c_i \in S} p_i = \sum_{i=1}^m p_i \leq K \text{ y } \sum_{i=1}^{m+1} p_i > K.$$

- En consecuencia tenemos también que

$$\sum_{c_i \in S} p_i + \sum_{c_i \in Q} p_i > K, \forall Q \subseteq T \setminus S, Q \neq \emptyset$$

Your company name

(puesto que  $p_{m+1} \leq p_j, \forall j > m$ ).



# Contenedores en un barco

- Veamos que cualquier otro subconjunto  $U \subseteq T$  de items  $c_i$  que contenga un número mayor de items que  $S$  no es una solución válida, por lo que la solución optima (con mayor número de items) es  $S$ :

Sea  $U \subseteq T$  tal que  $|U| = m' > m$ . Se demostrará que  $\sum_{c_i \in U} p_i > K$ , por lo que el conjunto  $U$  no es una solución válida. El valor de  $\sum_{c_i \in U} p_i$  puede descomponerse mediante  $\sum_{c_i \in U} p_i = \sum_{c_i \in S \cap U} p_i + \sum_{c_i \in U \setminus S} p_i$ .

Tenemos que  $|U| = m' = |S \cap U| + |U \setminus S|$ , y que  $|S| = m = |S \cap U| + |S \setminus U|$ , luego  $|U \setminus S| = m' - |S \cap U| > m - |S \cap U| = |S \setminus U|$ , porque  $m' > m$ . Por tanto en el conjunto  $U \setminus S$  hay más elementos que en  $S \setminus U$ . Así que podemos sustituir los elementos de  $U \setminus S$  por elementos de  $S \setminus U$  (que tienen menor valor) y aun nos quedarán algunos elementos sobrantes (exactamente  $|U \setminus S| - |S \setminus U|$ ). Como todos los elementos de  $U \setminus S$  son de mayor valor que todos los de  $S \setminus U$  ( $\forall c_j \in U \setminus S, \forall c_i \in S \setminus U, p_j \geq p_i$ ), entonces escojamos cualquier subconjunto  $R \subseteq U \setminus S$  tal que  $|R| = |S \setminus U| < |U \setminus S|$  (de forma que el subconjunto  $U \setminus S \setminus R \neq \emptyset$ ). Entonces

$$\begin{aligned} \sum_{c_i \in U} p_i &= \sum_{c_i \in S \cap U} p_i + \sum_{c_i \in U \setminus S} p_i = \sum_{c_i \in S \cap U} p_i + \sum_{c_i \in R} p_i + \sum_{c_i \in U \setminus S \setminus R} p_i \\ &\geq \sum_{c_i \in S \cap U} p_i + \sum_{c_i \in S \setminus U} p_i + \sum_{c_i \in U \setminus S \setminus R} p_i = \sum_{c_i \in S} p_i + \sum_{c_i \in U \setminus S \setminus R} p_i > K. \end{aligned}$$

- y por tanto  $U$  no es una solución válida

# Contenedores en un barco

- Veamos ahora el algoritmo maximizando el número de toneladas cargadas.
- En este caso parece intuitivo que el algoritmo greedy debe de intentar cargar en cada momento el contenedor de mayor peso de los que queden aun sin cargar. En otras palabras, cargar los contenedores en orden no creciente de peso.
- El algoritmo (que no da siempre el óptimo) sería pues:

```
Ordenar de mayor a menor el vector de pesos p[];  
i=1;  
sum=0;  
while (i<=n) {  
    if sum+p[i] <= k {  
        seleccionar c[i];  
        i=i+1;  
        sum=sum+p[i];  
    }  
    else i=i+1;  
}
```

Your compai

# Minimizar el número de visitas al proveedor

- Un granjero necesita disponer siempre de un determinado fertilizante. La cantidad máxima que puede almacenar la consume en  $r$  días, y antes de que eso ocurra necesita acudir a una tienda del pueblo para abastecerse.
- El problema es que dicha tienda tiene un horario de apertura muy irregular (solo abre determinados días). El granjero conoce los días en que abre la tienda, y desea minimizar el número de desplazamientos al pueblo para abastecerse.
  - Diseñar un algoritmo greedy que determine en qué días debe acudir al pueblo a comprar fertilizante durante un periodo de tiempo determinado (por ejemplo durante el siguiente mes).
  - Demostrar que el algoritmo encuentra siempre la solución óptima.



# Minimizar el número de visitas al proveedor

- La idea del algoritmo (greedy) es muy simple: aguantar lo máximo posible sin ir al pueblo, es decir acudir al pueblo en un día de apertura de la tienda, de modo que si no fuese ese día y acudiese en un día de apertura posterior, se quedaría sin fertilizante.
- Para formalizar el problema, sean  $d_1, d_2, \dots, d_n$  los días en que abre la tienda durante el periodo de interés a partir del día de inicio ( $d_0 = 0$ ), y  $d_{n+1}$  el día final del periodo de interés, de modo que si  $i < j$  entonces  $d_i < d_j$ .
- Para que el problema tenga solución, el número de días entre aperturas sucesivas de la tienda no puede ser mayor el número de días que tardamos en consumir el fertilizante (en caso contrario siempre nos quedaríamos sin fertilizante antes de la siguiente apertura), o sea  $d_{i+1} - d_i \leq r \quad \forall i = 0, \dots, n$ .
- Una solución factible es una lista ordenada (de los días en que iremos al pueblo),  $d_{j_0}, d_{j_1}, \dots, d_{j_m}, d_{j_{m+1}}$ , tal que
$$d_{j_0} = d_0, d_{j_{m+1}} = d_{n+1}, \text{ y } d_{j_{i+1}} - d_{j_i} \leq r$$

# Minimizar el número de visitas al proveedor

- La estrategia greedy es la siguiente: si nos encontramos en el día de visita  $d_i$ , entonces la siguiente visita será en el día  $d_{j^*}$  tal que  $j^* = \max_{j>i} \{j/d_j - d_i \leq r\}$  eso equivale a decir que  $j^*$  es el día tal que  $d_{j^*} - d_i \leq r$  pero  $d_{j^*+1} - d_i > r$
- Así puede probarse que la solución que se obtiene es optimal.

Sea  $d_{p_0}, d_{p_1}, \dots, d_{p_m}, d_{p_{m+1}}$  la solución propuesta por el algoritmo greedy. Supongamos que hay otra solución factible  $d_{q_0}, d_{q_1}, \dots, d_{q_k}, d_{q_{k+1}}$  que es preferible, es decir que realiza menos visitas,  $k < m$ . Vamos a demostrar por inducción en primer lugar que  $\forall j = 1, \dots, k, d_{p_j} \geq d_{q_j}$ .

Para el caso  $j = 1$  el resultado es evidente por la propia definición del algoritmo greedy: el granjero espera al último día de apertura al que puede llegar sin agotar el fertilizante (formalmente, si  $d_{p_1} < d_{q_1}$ , como debe ser  $d_{p_1} \leq r$  y  $d_{p_1+1} > r$ , entonces  $d_{q_1} \geq d_{p_1+1} > r$ , y  $q$  no sería una solución factible).

Supongamos (hipótesis de inducción) que es cierto que  $d_{p_{i-1}} \geq d_{q_{i-1}}$ . Entonces  $d_{q_i} - d_{p_{i-1}} \leq d_{q_i} - d_{q_{i-1}}$ , y como  $q$  es factible tenemos también que  $d_{q_i} - d_{q_{i-1}} \leq r$ . Por tanto  $d_{q_i} - d_{p_{i-1}} \leq r$ . Esto significa que  $d_{q_i}$  está dentro del alcance de  $d_{p_{i-1}}$ , por lo que el algoritmo greedy pudo escoger  $d_{q_i}$  y no lo hizo, así que  $d_{q_i}$  no puede ser el día más lejano dentro del alcance, y por tanto  $d_{p_i} \geq d_{q_i}$ . Queda pues probado que  $\forall j = 1, \dots, k, d_{p_j} \geq d_{q_j}$ .

Finalmente, tenemos por un lado que  $d_{p_k} \geq d_{q_k}$ , por tanto  $d_{n+1} - d_{p_k} \leq d_{n+1} - d_{q_k}$ . Además puesto que  $q$  es una solución factible, tiene que ser  $d_{n+1} - d_{q_k} \leq r$ . Combinando ambas desigualdades tenemos  $d_{n+1} - d_{p_k} \leq r$ . Esto significa que el día del final del periodo de interés está dentro del alcance del día  $d_{p_k}$ , por lo que no tendría sentido ir más días y llegamos a una contradicción (formalmente,  $r \geq d_{n+1} - d_{p_k} > d_{p_{k+1}} - d_{p_k} \geq d_{p_{k+1}} - d_{p_k} > r$ , la última desigualdad por definición de la estrategia greedy).

# Minimizar el tiempo medio de acceso

- Sean  $n$  programas  $P_1, P_2, \dots, P_n$  que hay que almacenar en una cinta. El programa  $P_i$  requiere  $s_i$  kilobytes de espacio y la cinta es suficientemente larga para almacenar todos los programas.
- Se sabe con qué frecuencia se utiliza cada programa: una fracción  $\pi_i$  de las solicitudes afecta al programa  $P_i$  (y por tanto  $\sum_{i=1}^n \pi_i = 1$ ).
- Los datos se almacenan en la cinta con densidad constante y la velocidad de la cinta también es constante.
- Una vez que se carga el programa, la cinta se rebobina hasta el principio.

# Minimizar el tiempo medio de acceso

- Si los programas se almacenan por orden  $i_1, i_2, \dots, i_n$  el tiempo medio requerido para cargar un programa es, por tanto:

$$\hat{T} = c \sum_{j=1}^n \left[ \pi_{i_j} \sum_{k=1}^j s_{i_k} \right]$$

- donde la constante  $c$  depende de la densidad de grabación y de la velocidad de la cinta.
- Se desea minimizar  $\hat{T}$  empleando un algoritmo greedy. Demostrar la optimalidad del algoritmo o encontrar un contraejemplo que demuestre que el algoritmo no es óptimo para los siguientes criterios de selección:
  - Programas en orden no decreciente de  $s_i$ .
  - Programas en orden no creciente de  $\pi_i$ .
  - Programas en orden no creciente de  $\pi_i/s_i$ .

# Minimizar el tiempo medio de acceso

- Para demostrar que las dos primeras opciones para seleccionar de forma greedy el orden de almacenamiento de los programas no son óptimas, basta con encontrar un ejemplo en el que los tiempos de acceso obtenidos sean mayores que los obtenidos por alguna ordenación de los programas.
- Consideremos el caso en que  $n = 3$  y los valores de  $s_i$  y  $\pi_i$  son los de la siguiente tabla:

| $i$     | 1   | 2   | 3   |
|---------|-----|-----|-----|
| $s_i$   | 10  | 20  | 40  |
| $\pi_i$ | 0.1 | 0.6 | 0.3 |

- En este caso el criterio (a) origina el orden 1, 2, 3, con un valor  $T_a = 40$  ( $0.1 \cdot 10 + 0.6 \cdot 30 + 0.3 \cdot 70$ ). El criterio (b) da el orden 2, 3, 1 con valor  $T_b = 37$  ( $0.6 \cdot 20 + 0.3 \cdot 60 + 0.1 \cdot 70$ ). Pero el orden 2, 1, 3 (que es el generado por el criterio (c), pues  $\pi_2/s_2 = 0.03 > \pi_1/s_1 = 0.01 > \pi_3/s_3 = 0.0075$ ) da un valor  $T_b = 0.6 \cdot 20 + 0.1 \cdot 30 + 0.3 \cdot 70 = 36$ .
- Por tanto los criterios (a) y (b) no producen un algoritmo greedy óptimo.



# Minimizar el tiempo medio de acceso

- Demostraremos que el criterio de selección c) si produce el óptimo
- Consideremos una ordenación cualquiera (por simplicidad en la notación supongamos que es)  $1, 2, \dots, n$  y la ordenación resultante de intercambiar las posiciones  $i$  e  $i + 1$  (una trasposición), o sea  $1, \dots, i - 1, i + 1, i, i + 2, \dots, n$ .
- El tiempo medio de acceso para la primera ordenación es (obviando la constante c)

$$\hat{T}_o = \sum_{j=1}^n \left[ \pi_j \sum_{k=1}^j s_k \right] = \sum_{j=1}^{i-1} \left[ \pi_j \sum_{k=1}^j s_k \right] + \pi_i \sum_{k=1}^i s_k + \pi_{i+1} \sum_{k=1}^{i+1} s_k + \sum_{j=i+2}^n \left[ \pi_j \sum_{k=1}^j s_k \right]$$

- Y el tiempo requerido para la segunda ordenación es

$$\hat{T}_t = \sum_{j=1}^{i-1} \left[ \pi_j \sum_{k=1}^j s_k \right] + \pi_{i+1} \left( \sum_{k=1}^{i-1} s_k + s_{i+1} \right) + \pi_i \sum_{k=1}^i s_k + \sum_{j=i+2}^n \left[ \pi_j \sum_{k=1}^j s_k \right]$$

# Minimizar el tiempo medio de acceso

- El primer y el último término en esas expresiones es el mismo en ambos casos, la única diferencia está en los dos sumandos intermedios.
- Si restamos obtenemos

$$\hat{T}_o - \hat{T}_t = \pi_i \sum_{k=1}^i s_k + \pi_{i+1} \sum_{k=1}^{i+1} s_k - \pi_{i+1} \left( \sum_{k=1}^{i-1} s_k + s_{i+1} \right) - \pi_i \sum_{k=1}^{i+1} s_k = -\pi_i s_{i+1} + \pi_{i+1} s_i$$

- Luego

$$\hat{T}_o \leq \hat{T}_t \iff \pi_i / s_i \geq \pi_{i+1} / s_{i+1}$$

- Es decir, que el tiempo medio de acceso es mejor (menor) cuando un programa con cociente  $\pi/s$  se coloca antes que otro programa cuyo cociente  $\pi/s$  sea menor.

# Minimizar el tiempo medio de acceso

- Consideremos entonces la ordenación resultante del criterio (c).
- Por simplicidad en la notación supongamos que dicha ordenación es tal que  $\pi_1/s_1 \geq \pi_2/s_2 \geq \dots \geq \pi_n/s_n$  (sino renombramos los índices).
- Cualquier otra ordenación se puede obtener a partir de esta como una permutación la misma, y toda permutación se puede obtener como una sucesión de trasposiciones.
- En cada una de esas trasposiciones el tiempo medio de acceso se va incrementando, según lo visto antes.
- Por tanto ninguna de las otras ordenaciones tiene un tiempo de acceso menor que la del criterio (c), por lo que este criterio genera de forma greedy la ordenación óptima.

# Recubrimiento de un grafo no dirigido

- Sea  $G = (V, E)$  un grafo no dirigido. Un conjunto  $U$  se dice que es un recubrimiento de  $G$  si  $U \subseteq V$  y cada arista en  $E$  incide en, al menos, un vértice o nodo de  $U$ , es decir  $\forall (x, y) \in E$ , o bien  $x \in U$  o  $y \in U$ . Un conjunto de nodos es un recubrimiento minimal de  $G$  si es un recubrimiento con el menor número posible de nodos.
- Diseñar un algoritmo greedy para intentar obtener un recubrimiento minimal de  $G$ . Demostrar que el algoritmo es correcto, o dar un contraejemplo.
- Diseñar un algoritmo greedy que obtenga un recubrimiento minimal para el caso particular de grafos que sean árboles y opcionalmente, realizar un estudio experimental de las diferencias entre los dos algoritmos anteriores cuando ambos se aplican a árboles.

# Recubrimiento de un grafo no dirigido

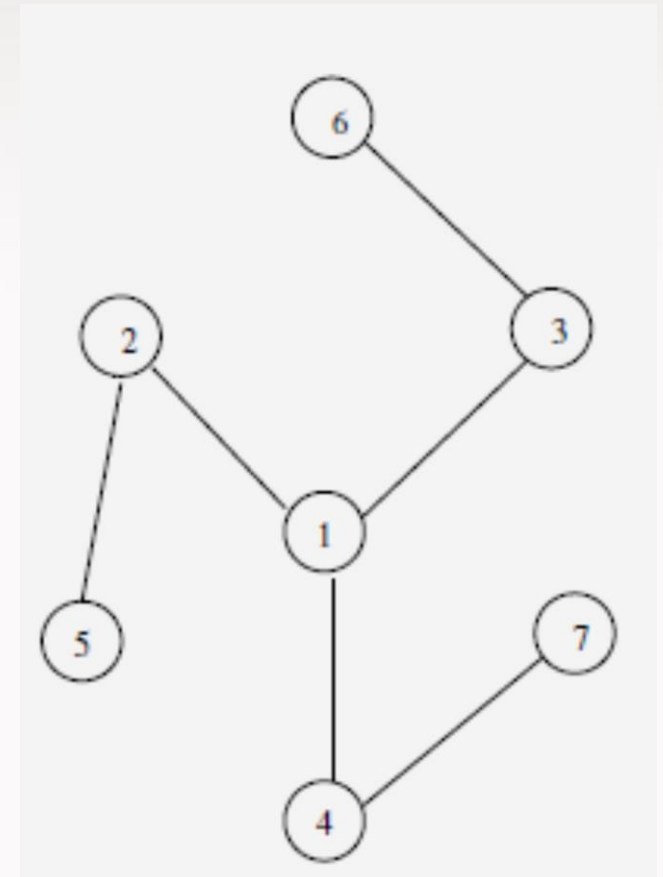
- La estrategia greedy consistirá en ir seleccionando de uno en uno los nodos que formarán el recubrimiento.
- Una función de selección razonable sería escoger en cada paso el nodo con mayor grado de incidencia de entre los nodos aún no seleccionados (el grado de incidencia de un vértice es el número de aristas que inciden en él). De este modo se intenta recubrir el mayor número posible de aristas con el menor número de nodos.
- El algoritmo greedy para el caso de grafos cualesquiera sería así,

```
recubrimiento(V,E) {  
    U=vacio;  
    while (E not vacio) {  
        escoger el nodo v de V de maximo grado de incidencia;  
        U=U union {v};  
        V=V-{v},  
        E=E-{(u,w) | u=v ó w=v};  
    }  
    return (U)  
}
```



# Recubrimiento de un grafo no dirigido

- Este algoritmo no garantiza encontrar un recubrimiento minimal.
- Por ejemplo, para el grafo  $G = (V, E)$  de la figura:  $V = \{1, 2, 3, 4, 5, 6, 7\}$  y  $E = \{(1, 2), (1, 3), (1, 4), (2, 5), (3, 6), (4, 7)\}$ , el algoritmo anterior da como resultado  $U = \{1, 2, 3, 4\}$  o también, cambiando 5 por 2, 6 por 3 y 7 por 4. El tamaño del conjunto es 4.
- Sin embargo, la solución óptima es  $U^* = \{2, 3, 4\}$ , de tamaño 3.



# Recubrimiento de un grafo no dirigido

- En un grafo que sea un árbol (o un bosque de arboles) siempre podemos encontrar al menos un nodo hoja (un nodo con una única arista incidente en el).
- Para poder recubrir esa arista es necesario que alguno de sus nodos extremos esté en el recubrimiento. Seleccionaremos el único nodo adyacente al nodo hoja (que tiene la posibilidad de recubrir otras aristas, mientras que el nodo hoja solo puede recubrir a su arista incidente).
- Si eliminamos del árbol todas las aristas incidentes en el nodo seleccionado, el resultado sigue siendo un bosque. Por tanto sigue existiendo al menos un nodo hoja en ese grafo.
- Repetimos el proceso hasta que hayamos eliminado todas las aristas del árbol (cuando tengamos solo una arista aislada, podemos escoger cualquiera de sus nodos extremos). El resultado es necesariamente un recubrimiento minimal, porque todos los nodos seleccionados son imprescindibles para poder recubrir al menos una de las aristas.

# Recubrimiento de un grafo no dirigido

- Para poder comparar ambos algoritmos, la idea es generar aleatoriamente muchos árboles, aplicar a cada árbol ambos algoritmos y calcular la diferencia entre el tamaño del algoritmo óptimo y el del aproximado.
- Para generar árboles aleatoriamente, se describe a continuación un algoritmo que toma como entradas el tamaño del conjunto de vértices y el máximo número de hijos permitido.

```
genera_arbol(numnodes, maxh) {  
  
    n=1; //cuenta el número de nodos generados hasta ahora  
    i=0; //etiqueta del nodo  
    encolar(q,i); //q es una cola FIFO  
    while (n < numnodes) {  
        i=desencolar(q);  
        ch=random(1..maxh); //entero aleatorio entre 1 y maxh  
        if ((ch+n) > numnodes) ch=numnodes-n; //para no generar más de numnodes nodos  
        for (j=n; j<ch+n; j++) {  
            v[i,j]=1; //v es la matriz de adyacencia del árbol  
            encolar(q,j);  
        }  
        n=n+ch  
    }  
    while (not vacia(q)) k=desencolar(q);  
}
```

# Reparaciones

- Un electricista necesita hacer  $n$  reparaciones urgentes, y sabe de antemano el tiempo que le va a llevar cada una de ellas: en la tarea  $i$ -ésima tardará  $t_i$  minutos.
- Como en su empresa le pagan dependiendo de la satisfacción del cliente y esta es inversamente proporcional al tiempo que tardan en atenderles, necesita decidir el orden en el que atenderá los avisos para minimizar el tiempo medio de atención de los clientes (desde el inicio hasta que termine la reparación)
- Diseñar un algoritmo greedy para resolver esta tarea. Demostrar que el algoritmo obtiene la solución óptima.
- Modificar el algoritmo anterior para el caso de una empresa en la que se disponga de los servicios de más de un electricista.

# Reparaciones

- Minimizar el tiempo medio de atención es lo mismo que minimizar el tiempo total de atención.
- Como un cliente  $j$  espera mientras atienden a todos los clientes anteriores a él  $i_1, i_2, \dots, i_m$ , el incremento en tiempo que se obtiene al añadir  $P$  al cliente  $j$  es igual a la suma de los tiempos de servicio de esos  $m$  clientes,

$$\sum_{k=1}^m t_{i_k}$$

- (porque eso es lo que el cliente  $j$  tiene que esperar antes de recibir servicio), mas  $t_j$  (el tiempo necesario para servir a ese cliente). Por tanto si se quiere minimizar el tiempo total, parece que lo mejor es atender a continuación al cliente con menor tiempo de servicio.
- Así la estrategia greedy es atender en cada momento al cliente que requiera menor tiempo de servicio de entre los pendientes. Dicho de otra manera, atender a los cliente en orden no decreciente de tiempos de servicio.



# Reparaciones

- Sea  $P = p_1 p_2 \dots p_m$  cualquier permutación de los enteros de 1 a  $n$ , y sea  $s_i = t_{p_i}$ . Si se sirve a los clientes en el orden  $P$ , entonces el tiempo requerido para servir a todos los clientes será:

$$T(P) = s_1 + (s_1 + s_2) + (s_1 + s_2 + s_3) + \dots + (s_1 + s_2 + \dots + s_n) =$$
$$ns_1 + (n-1)s_2 + (n-2)s_3 + \dots + 2s_{n-1} + s_n = \sum_{k=1}^n (n-k+1)s_k$$

- Si  $P$  no ordena a los clientes en orden creciente de tiempos de servicio, entonces podemos encontrar dos enteros  $a$  y  $b$  tal que  $a < b$  y  $s_a > s_b$ . Es decir, se sirve al cliente  $a$ -ésimo antes que al  $b$ -ésimo, aunque el primero necesite más tiempo de servicio que el segundo. Si intercambiamos las posiciones de esos dos clientes, obtenemos un nuevo orden de servicio  $P'$  (el orden  $P$  después de intercambiar los enteros  $p_a$  y  $p_b$ ). Con esta nueva planificación, el tiempo total en el sistema de todos los clientes es:

$$T(P') = (n-a+1)s_b + (n-b+1)s_a + \sum_{k=1, k \neq a, b}^n (n-k+1)s_k$$

# Reparaciones

- Esto es así porque para todos los  $s_i$  que no sean  $s_a$  y  $s_b$ , todos continúan apareciendo la misma cantidad de veces que antes. En cambio  $s_a$  que antes aparecía  $n - a + 1$  veces ahora se retrasa a la posición  $b$ , por lo que aparece  $b - a$  veces menos, es decir  $s_a$  aparece  $(n - a + 1) - (b - a) = n - b + 1$  veces.
- Por otro lado  $s_b$  que aparecía  $n - b + 1$  veces, ahora se adelanta a la posición de  $a$ , por lo que aparece  $b - a$  veces más, en total  $(n - b + 1) + (b - a) = n - a + 1$  veces. La diferencia entre los tiempos de ambas planificaciones es:

$$\begin{aligned} T(P) - T(P') &= (n - a + 1)s_a + (n - b + 1)s_b - ((n - a + 1)s_b + (n - b + 1)s_a) = \\ &= ((n - a + 1) - (n - b + 1))s_a + ((n - b + 1) - (n - a + 1))s_b = (b - a)(s_a - s_b) > 0 \end{aligned}$$

- Por tanto la planificación  $P'$  es preferible a  $P$  (consume menos tiempo total). De este modo cualquier planificación que no ordene de forma no decreciente se puede mejorar sucesivamente intercambiando el orden de clientes que estén ordenados en orden decreciente de tiempos. Así las únicas planificaciones que no se pueden mejorar son las que ordenan de forma no decreciente todos sus elementos.

# Reparaciones

- Si suponemos que hay  $M$  electricistas, el criterio no cambia esencialmente:
- Se ordenan las tareas de forma no decreciente de tiempo de servicio y vamos asignando las tareas al primer electricista que quede desocupado.
- En definitiva, al electricista  $k$ -ésimo le asignamos las tareas (ordenadas)  $k, k + M, k + 2M, \dots$