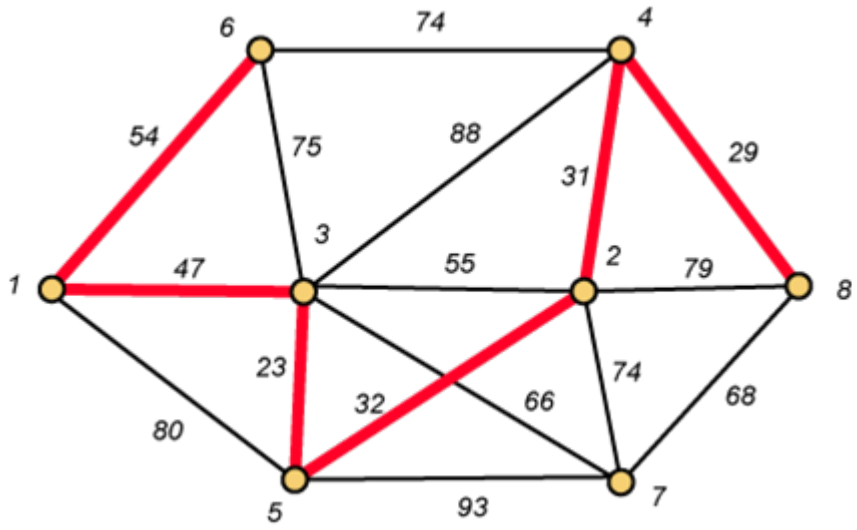
Práctica 2
Algorítmica
Algoritmos Greedy



Manuel Ariza Ortiz
Javier Bueno López
Carlos Enríquez López
Sergio Cruz Pérez

Índice

Problema 1: Contenedores en un barco

En el problema que se nos presenta se tiene un buque mercante cuya capacidad de carga es de K toneladas y un conjunto de contenedores $c_1, \ldots, c_n$ cuyos pesos respectivos son $p_1, \ldots, p_n$ (expresados también en toneladas).

Teniendo en cuenta que la capacidad del buque es menor que la suma total de los pesos de los contenedores se nos piden una serie de objetivos:
– Diseñar un algoritmo que maximice el número de contenedores cargados. Demostrar su optimalidad.
– Diseñar un algoritmo que intente maximizar el número de toneladas cargadas.

Si observamos con detenimiento el problema es fácil observar que en realidad se trata del problema de la mochila pero con ligeras modificaciones que tendremos que abordar para demostrar los algoritmos que son óptimos y los que no.
Al tratarse de un problema de la mochila que debemos afrontar con heurísticas greedy nuestro objetivo general en cada algoritmo será :

-Maximizar la función de beneficio para cada algoritmo:
$f(c)=\sum^{n}_{i=0} b_i c_i$ Donde b_i es el beneficio que nos aporta cada contenedor c_i.

-Sin superar la carga máxima que puede soportar nuestro recipiente. En este caso el barco.
$g(c)=\sum^{n}_{i=0} p_i c_i \leq K$ Donde p_i es el peso de un contenedor y K la carga máxima del barco.

Solución al problema con enfoque greedy:

Para resolver estos problemas con algoritmos voraces primero necesitaremos que consten de una serie de elementos básicos que serán muy parecidos en los dos diseños que se nos piden:

-Conjunto de candidatos: Los elementos seleccionables .En nuestro caso los contenedores que podemos cargar.
-Una solución parcial: los contenedores que hayamos seleccionado hasta el momento en un instante.
-Una función de selección: que determina el mejor candidato del conjunto de candidatos. Variará dependiendo del diseño pedido. Se especificará más adelante.
-Función de factibilidad: determina si es posible completar la solución parcial para alcanzar una solución del problema. En nuestro caso no sobrepasar la carga máxima del barco, es decir,
$g(c)=\sum^{n}_{i=0} p_i c_i \leq K$
-Criterio que define lo que es una solución: indica si la solución parcial obtenida resuelve el problema. Cumplirá el concepto de solución los contenedores cargados que en cada momento cumplan la función de factibilidad.
-Función objetivo: valor de la solución alcanzada que en nuestro caso queremos optimizar. Dependerá de la variante del problema que se pida. Se especificará más adelante.

El esquema general que seguiremos en las implementaciones será el propio de los algoritmos voraces que explicamos ahora:

-En primer lugar se parte de un conjunto solución vacío: $S = \varnothing$.
-De la lista de candidatos, se elige el mejor (de acuerdo con la función de selección).
-Comprobamos si se puede llegar a una solución con el candidato seleccionado (función de factibilidad). Si no es así, lo eliminamos de la lista de candidatos posibles y nunca más lo consideraremos. En caso contrario lo añadimos al conjunto de soluciones.

-Si aún no hemos llegado a una solución, seleccionamos otro candidato y repetimos el proceso hasta llegar a una solución (o quedarnos sin posibles candidatos).

Algoritmo que maximice el número de contenedores cargados:

Para implementar este primer caso (usando como lenguaje C++) simplemente escogemos en cada momento el contenedor todavía no cargado en el barco que tenga el menor peso. O dicho de otra forma, seleccionamos los contenedores en orden no decreciente de peso.
Para simplificar el problema nuestros contenedores abarcarán entre 1 y 100 unidades de peso.

```
//Entrada: vector de contenedores y la capacidad maxima del barco.
//Devuelve un vector con los contenedores del barco.
vector<int> llenaBarcoContenedores(multiset<int, less<int> > c ,int peso_max, int &peso_actual) {
    vector<int> barco;
    int peso_contenedor;
    int num_contenedor;
    peso_actual = 0;
    bool lleno = false;

    while( !c.empty() && !lleno ) {
        peso_contenedor = *(c.begin());//Seleccionamos el contenedor más ligero.

        if((peso_actual + peso_contenedor) <= peso_max) { //Funcion de factibilidad
            barco.push_back(peso_contenedor);       //Introducimos el contenedor
            peso_actual += peso_contenedor;     //en el barco.

            c.erase(c.begin());  //Eliminamos el contenedor de la lista.
        }
        else
            lleno = true;
    }
    return barco;
}
```

(Código completo al final de la memoria)

La función adjuntada es el algoritmo greedy encargado de cargar los contenedores en el barco. Esta recibe una bolsa con el conjunto de candidatos y devuelve un vector que contiene los candidatos seleccionados.
El funcionamiento es sencillo. Seleccionamos el contenedor más ligero. Si cumple la función de factibilidad lo cargamos en el barco, incrementamos el peso del barco por haberlo admitido y eliminamos de la lista de candidatos el contenedor cargado. Si la función de factibilidad no se cumple para algún contenedor quiere decir que no lo podemos cargar y tampoco a los sucesivos ya que todos tienen un peso mayor que el actual.
Repetimos el proceso hasta que alguno no cumpla la función de factibilidad o nos quedemos sin contenedores.

Este algoritmo da para un conjunto de candidatos aleatorio de 10 contenedores una salida como la siguiente, donde las cargas del barco muestran el peso de cada contenedor:

```
Para 10 contenedores y una carga maxima de 300
en el barco sobran 16 unidades de peso.
Contenedores cargados: 8

Carga del barco:
[1]9    [2]21   [3]26   [4]30   [5]34   [6]46   [7]55   [8]63

Contenedores a cargar:
9 - 21 - 26 - 30 - 34 - 46 - 55 - 63 - 75 - 96 -
```

Demostrando la optimalidad:

Como habíamos dicho anteriormente trataremos este problema como un problema de la mochila fraccional con algunos matices.

Para demostrar que siguiendo la ordenación dada el algoritmo encuentra la solución óptima, vamos a demostrar que los elementos están ordenados por densidad (beneficio en función del peso) , es decir, que $b_i/p_i \geq b_j/p_j$ si $i < j$:

> -En nuestro caso concreto la función objetivo, es decir, la función que queremos maximizar es esta:
> $f(c)=\sum^{n}_{i=0} b_i c_i$ donde b_i es el beneficio que nos aporta cada contenedor c_i. Sin embargo como lo que buscamos maximizar es el numero de contenedores y cada contenedor podemos cargarlo únicamente una vez entonces $b_i = 1, \forall i, 1 \leq i \leq n$ donde n es el número de contenedores.
>
> -Por otro lado nuestra función de factibilidad es esta: $g(c)=\sum^{n}_{i=0} p_i c_i \leq K$ donde p_i es el peso de cada contenedor y por lo tanto el costo de subirlo al barco.
>
> -Por lo tanto si $b_i = 1, \forall i, 1 \leq i \geq n$ tenemos que la densidad de cada contenedor es de $b_i/p_i = 1/p_i$. Al estar nuestros contenedores ordenados en orden **no decreciente** según peso también lo están en orden **no creciente** según densidad. $p_i \leq p_j$ si $i < j$ entonces $1/p_i \geq 1/p_j$ si $i < j$.

Ahora que sabemos que nuestros contenedores están ordenados de manera no creciente por densidad podemos continuar con la demostración de la optimalidad según el problema de la mochila.

Sea $C = (c_1,c_2,...,c_n)$ la solución encontrada por el algoritmo. Si $c_i = 1$ para todo i, la solución es óptima(algo que se cumple ya que cargamos contenedores completos). Consideremos entonces $Y = (y_1,y_2,...,y_n)$ otra solución, y sea $f(y)=\sum^{n}_{i=0} b_i y_i$ su beneficio. Por ser solución cumple que $\sum^{n}_{i=0} p_i y_i \leq M$. Entonces, restando ambas capacidades, podemos afirmar que $\sum^{n}_{i=0}(p_i c_i - p_i y_i) \geq 0$.

Si $i < j$ entonces $c_i = 1$, y por tanto $(c_i - y_i) \geq 0$. Además, $(b_i/p_i) \geq (b_j/p_j)$ por la ordenación escogida (decreciente en densidad). En consecuencia, podemos afirmar que $(c_i - y_i)(b_i/p_i) \geq (c_i - y_i)(b_j/p_j)$ para todo i, y por tanto:

$$f(c) - f(y) = \sum^{n}_{i=0} c_i (b_i/p_i) - \sum^{n}_{j=0} y_j (b_j/p_j) \geq 0$$

Esto es, $f(c) \geq f(y)$, como queríamos demostrar.

Algoritmo que maximice el número de toneladas cargadas:

Tras haber solucionado el primer problema este se vuelve más sencillo.
Para implementar este segundo caso simplemente escogemos en cada momento el contenedor todavía no cargado en el barco que tenga el mayor peso (justo al contrario que en algoritmo anterior). O dicho de otra forma, seleccionamos los contenedores en orden no creciente de peso.

La función usada para este algoritmo greedy encargado de cargar los contenedores en el barco es la misma que en el primer diseño solo que en este caso recibe una bolsa con los contenedores ordenados en orden no creciente de peso. Con su funcionamiento ocurre lo mismo.

```
//Entrada: vector de contenedores y la capacidad maxima del barco.
//Devuelve un vector con los contenedores del barco.
vector<int> llenaBarcoPeso(multiset<int, greater<int> > c ,int peso_max, int &peso_actual) {
```

Refutando la optimalidad:

En esta ocasión este problema también se comporta como un problema de la mochila pero en este caso es un problema de mochila 0/1.
Esto se debe a que la función objetivo para este problema ($f(c)=\sum^{n}_{i=0} b_i c_i$) busca maximizar el peso mientras que la función de factibilidad ($g(c)=\sum^{n}_{i=0} p_i c_i$) sigue teniendo el peso como coste. Por lo tanto: $b_i = p_i$ y $f(c) = g(c)$ lo que da lugar a que el problema se comporte igual que el de la mochila 0/1.

No obstante el camino más sencillo para demostrar que el enfoque greedy no da solución óptima en este caso es un contra-ejemplo. En este la captura representa una ejecución del algoritmo voraz utilizado para resolver el problema.

```
Para 10 contenedores y una carga maxima de 400
en el barco sobran 15 unidades de peso.
Peso cargado: 385/400

Carga del barco:
[1]80   [2]79   [3]72   [4]58   [5]55   [6]41

Contenedores a cargar:
80 - 79 - 72 - 58 - 55 - 41 - 33 - 26 - 25 - 3
```

Como podemos ver el algoritmo consigue cargar 385 unidades de peso de un total de 400 posibles utilizando el criterio de meter en primer lugar el contenedor más pesado disponible.
Sin embargo existe otra solución mejor a la dada por el algoritmo:

-Se trataría de no cargar el contenedor n.º 6 de 41 unidades de peso y cargar en su lugar los contenedores de pesos 26, 25 y 3 con lo cual conseguiríamos cargar un total de 398 unidades de peso de 400 posibles.

-Dado que existe una solución mejor a la dada por el algoritmo podemos confirma que no se trata de un algoritmo optimal.

Problema 2: Granjero

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.

Solución:
Hemos declarado dos funciones, una para generar un vector de enteros comprendidos entre 1 y 31, que representará los días que abre la tienda, y más abajo una función que genera el parámetro r comprendido entre 1 y 13 de forma aleatoria.
Lo que hace la función realmente importante void DiasAacudir() es lo siguiente:
Recibe un set de enteros con los días de apertura de la tienda, y un entero con la duración máxima del fertilizante.
El procedimiento es el siguiente, mediante un bucle for pasamos por los 31 días que tiene el mes y a la vez comprobamos todos los días en los que la tienda estará abierta, si alguna de las dos cosas llega a su final, el bucle terminará.
Dentro del bucle nos encontramos tres condiciones, la primera es para que si se da el caso de que la duración del fertilizante es menor que el día siguiente de apertura, obviamente el problema no tendrá solución. La siguiente condición comprueba que la duración del fertilizante sea mayor o igual que el siguiente día de apertura pero menor al siguiente día del siguiente día de apertura para ir a comprar el fertilizante, y hacemos una actualización de que tendremos fertilizante disponible para la suma del día en el que compro, más lo que dura el fertilizante.Si se cumple esta condición se muestra por pantalla el día en el que el granjero compra fertilizante.
La última condición comprueba si el granjero ha visitado la tienda el último día para mostrar por pantalla dicho acontecimiento.
Así se obtiene la sucesión de días para visitar la tienda de forma óptima.

Optimalidad:

Para demostrar la optimalidad vamos a suponer dos conjuntos, el primero lo vamos a denotar con la letra O y va a almacenar los días de visita que compondrían la solución óptima, y por otra parte tendremos al conjunto D, formado por una serie de días, pero que no corresponderán a la solución óptima.Presentamos la situación, descrita anteriormente:

-D→ días seleccionados → (No es óptimo)=$d_1,d_2,d_3...d_n$
-O→ solución óptima=$o_1,o_2,o_3...o_n$

Por lo tanto, según la situación planteada, el conjunto D será más grande que el conjunto O, porque el conjunto D contiene más días que el conjunto O.
Dipondremos también de r, que será igual al máximo valor donde coinciden los elementos D y O, y lo que queremos representar con esto es lo siguiente:

$[d_1=o_1,d_2=o_2...d_r=o_r,d_{r+1}>o_{r+1}...]$
Es decir, todos los días coinciden hasta llegar a r, máximo valor donde coinciden, como ya hemos dicho. Y una vez pasado esto, el resto de elementos, siempre será mayor el elemento del conjunto D que el del O.

Por lo tanto suponemos otra solución óptima que va a ser igual a la composición de D y O:

$[d_1,d_2...d_r,d_{r+1},o_{r+2},o_{r+3}...O_n]$

Una vez planteado esto llegamos a que dicha solución tiene el mismo tamaño que la solución óptima y,además, llegamos a una contradicción: r no es el máximo valor donde se alcanza la igualdad entre D y cualquier solución óptima.

Problema 3: Minimizar tiempo medio de acceso

Sean n programas P_1,P_2, ..., P_n programas que se quieren almacenar en una cinta. El programa P_i requiere s_i Kb de memoria. La cinta tiene capacidad para almacenar todos los programas.

Se sabe con que frecuencia se utiliza cada programa: una fracción π_i de las solicitudes afecta al programa P_i (y por tanto $\sum_{i=1}^{n} \pi = 1$.)

Los datos se almacenan en la cinta con densidad constante y la velocidad de la cinta también es constante.

Una vez se carga el programa, la cinta se rebobina hasta el principio.

Si los programas se almacenan en orden i_1, i_2, ..., i_n el tiempo medio requerido para cargar un programa es, por tanto:

$$T = c \sum_{j=1}^{n} [\pi_{ij} \sum_{k=1}^{j} s_{ik}]$$

donde la constante c depende de la densidad de grabación y de la velocidad de la cinta.

Se desea minimizar *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:

1. **Programas en orden no decreciente de los s_i**
2. **Programas en orden no creciente de los π_i**
3. **Programas en orden no creciente de π_i/s_i**

Solución:

1.-

Puede resultar intuitivo posicionar los programar más cortos a comienzos de la cinta para así hacer que los programas que vienen detrás en la cinta tengan que esperar un tiempo menor. Esta afirmación podemos revocarla mediante el siguiente ejemplo.

Supongámos dos programas P_1 y P_2. P_1 con $\pi = 1$, $s = 10$ y P_2 con $\pi = 10$, $s = 11$.

Con el metodo de solución que se propone anteponemos el programa P_1 primero y luego el P_2. El costo total resultante es $1 \cdot 10 + 10 \cdot (10 + 11) = 220$. Pero si se hubiera ordenado al revés, ubicando el P_2 primero, el costo total será $10 \cdot 11 + 1 \cdot (11 + 10) = 131$. Podemos concluir por lo tanto en que el criterio empleado no es óptimo y además difiere de este bastante. En el ejemplo se han mostrado programas relativamente pequeños para ayudar a la comprensión, pero sin embargo cuando los tamaños de los programas aumenta, la diferencia entre el óptimo y el resultante de usar este método, se dispara.

Se observa claramente mirando el ejemplo que el problema central de esta estrategia es no considerar para nada la frecuencia de uso de un programa: En el ejemplo, si ambos programas tuvieran la misma frecuencia, la elección de poner el más corto primero sería óptima, pero como las frecuencias son extremadamente dispares, mientras que las longitudes son casi iguales,
conviene ubicar primero el programa más frecuente, aunque sea un poco
más largo, para evitar en la mayoría de los casos tener que leer dos programas de la cinta en lugar de solo el deseado.

Partiendo de la cinta vacía…
for i:=1 to n do
almacenar el programa más corto
almacenar(1)

2.-

Esta estrategia presenta el mismo problema que la anterior en el sentido de que solo considera una "variable", en este caso es solo la frecuencia. El error cometido en este caso es de misma magnitud que al usar la tecnica anterior.

Partiendo de la cinta vacía…
for i:=1 to n do
almacenar el programa con menos frecuencia
almacenar(2)

3.-

Como hemos visto en los enunciados anteriores, una solución óptima deberá incluir todos los datos presentes en el programa(en nuestro caso frecuencia y longitud).

Se nos pide demostrar que esta solución es mejor que las anteriores; para llevar a cabo esta demostración calcularemos el coste de intercambiar dos programas adyacentes en la cinta antes y después.

$$Coste\ original = \sum_{j=1}^{n} \left(\pi_j \sum_{k \le j} s_k \right) = ¿$$

Expandimos la expresión para mostrar por separado los términos correspondientes a los programas consecutivos que vamos a intercambiar:

$$= \sum_{j=1}^{i-1} \left(\pi_j \sum_{k \le j} s_k \right) + \pi_i \left(s_i + \sum_{k<i} s_k \right) + \pi_{i+1} \left(s_i + s_{i+1} + \sum_{k<i} s_k \right) + \sum_{j=i+2}^{n} \left(\pi_j \sum_{k \le j} s_k \right)$$

Ahora para calcular el coste del intercambio sólo hay que intercambiar *i* e *i+1* y nos queda la siguiente expresión:

$$\text{Swap} = \sum_{j=1}^{i-1} \left(\pi_j \sum_{k \le j} s_k \right) + \pi_{i+1} \left(s_{i+1} + \sum_{k<i} s_k \right) + \pi_i \left(s_{i+1} + s_i + \sum_{k<i} s_k \right) + \sum_{j=i+2}^{n} \left(\pi_j \sum_{k \le j} s_k \right)$$

Finalmente restamos para conocer el costo de intercambio:

CI=Coste original – Swap = $\pi_i\, s_i + 1 - \pi_{i+1}\, s_i$

Podemos deducir que el signo de CI dependerá de:

$$signo(CI) = signo\left(\frac{\pi_i}{s_i} - \frac{\pi_{(i+1)}}{s_{(i+1)}} \right)$$ *Esta expresion es la misma que la anterior, pero modificada a nuestro interés.*

Finalmente para demostrar que la solución es óptima se puede usar el siguiente ejemplo: Sea A una permutación de programas que está ordenada de acuerdo a la estrategia 3, es decir:

para $1 \leq i < n$

Queremos demostrar que A tiene costo óptimo. Tomamos alguna permutación optima B . Lo que queremos probar es que A y B tienen el mismo costo.
Como B es óptima, (ya que si no lo fuese tendría un coste menor y estamos suponiendo que los términos están ordenados de forma óptima) A y B están ordenados . Por lo tanto, la única diferencia que se puede plantear es que A y B sean iguales. Esto nos generaría la siguiente situación:

$$\frac{\pi_i}{s_i} = \frac{\pi_{(i+1)}}{s_{(i+1)}}$$

Por lo tanto, es posible pasar de A a B realizando únicamente intercambios de elementos consecutivos que empatan.

Cuando dos elementos empatan podemos comprobar que ambos elementos son iguales y por lo tanto el coste de intercambio no se modifica(En la fórmula del signo se puede ver cláramente). Entonces, es posible pasar de A a B sin aumentar coste, lo que queríamos demostrar. Si podemos intercambiar dos elementos de la cinta sin aumentar el coste estamos por lo tanto ante el algoritmo óptimo(ya que reducir el coste como hemos visto no tiene sentido).

Problema 5: Reparaciones(Electricista)

• 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.

Solución:
Para el primer algoritmo simplemente tenemos un vector con el tiempo que se tarda en hacer cada reparación.
Con ese vector lo que hacemos es ordenarlo del menor tiempo al mayor, de esta manera vamos reparando de uno en uno en el orden en el que menos se tarda en reparar.

Tabla con los tiempos de cada cliente;

1	4	2	3	4	9	1	7	10	9

Tabla ordenada:

1	1	2	3	4	4	7	9	9	10

Suma de tiempo total: 50 minutos.

Y vamos haciendo tareas:

1	2	3	4	4	7	9	9	10

Tiempo consumido: 1
le restan 49 minutos minutos por consumir

2	3	4	4	7	9	9	10

Tiempo consumido: 2
le restan 48 minutos por consumir

Así hasta llegar a 0 minutos por consumir, como se ve en la salida del programa a continuación.

Este es el código usado en c++ en el que hacemos el algoritmo Greedy:

```
clientes.push_back(1);
clientes.push_back(4);
clientes.push_back(2);
clientes.push_back(3);
clientes.push_back(4);
clientes.push_back(9);
clientes.push_back(1);
clientes.push_back(7);
clientes.push_back(10);
clientes.push_back(9);
//Ordenar el vector
sort(clientes.begin(),clientes.end());
//Mostrar el vector:
cout << "Lista de tiempos asignados a los clientes: " << endl;
for (int i = 0; i < clientes.size(); i++){
        suma += clientes[i];
        cout << clientes[i] << endl;
}
cout << "\nTiempo total para atender clientes: " << suma << " min" << endl;

//Atender a clientes
for(int i = 0; i < clientes.size(); i++){
int restante = 0;
        consumido+=clientes[i];
        restante = suma - consumido;
        cout << "Cliente n" << i << ": electricista ha consumido " << consumido
<< " min, para terminar le quedan" << restante << " min" << endl;
}
```

y esta la salida que nos da:

```
Tiempo total para atender clientes: 50 min
Cliente n0: electricista ha consumido 1 min, para terminar le quedan49 min
Cliente n1: electricista ha consumido 2 min, para terminar le quedan48 min
Cliente n2: electricista ha consumido 4 min, para terminar le quedan46 min
Cliente n3: electricista ha consumido 7 min, para terminar le quedan43 min
Cliente n4: electricista ha consumido 11 min, para terminar le quedan39 min
Cliente n5: electricista ha consumido 15 min, para terminar le quedan35 min
Cliente n6: electricista ha consumido 22 min, para terminar le quedan28 min
Cliente n7: electricista ha consumido 31 min, para terminar le quedan19 min
Cliente n8: electricista ha consumido 40 min, para terminar le quedan10 min
Cliente n9: electricista ha consumido 50 min, para terminar le quedan0 min
```

Para la segunda parte del problema en el que se nos pide que adaptemos el problema a varios electricistas tenemos el mismo vector:

1	4	2	3	4	9	1	7	10	9

Que también tendremos que ordenar:

1	1	2	3	4	4	7	9	9	10

Para este problema hemos creado una funcion a la cual metemos como parámetros un vector de vectores de enteros que representa el trabajo, un vector con los tiempos de las reparaciones de los clientes y el numero de electricistas. Lo que hacemos es ir repartiendo el trabajo que menor tiempo de reparación tiene a los electricistas, por ejemplo, para este caso en la primera iteración repartiriamos las tareas de la siguiente forma:
electricista 0 → 1
electricista 1 → 1
electricista 2 → 2

Veamos un ejemplo gráfico:
Supongamos que tenemos 3 electricistas:

Trabajo

Tarea para electricista 0	Tarea para electricista 1	Tarea para electricista 2

Y empezamos a repartir las tareas de la siguiente forma:

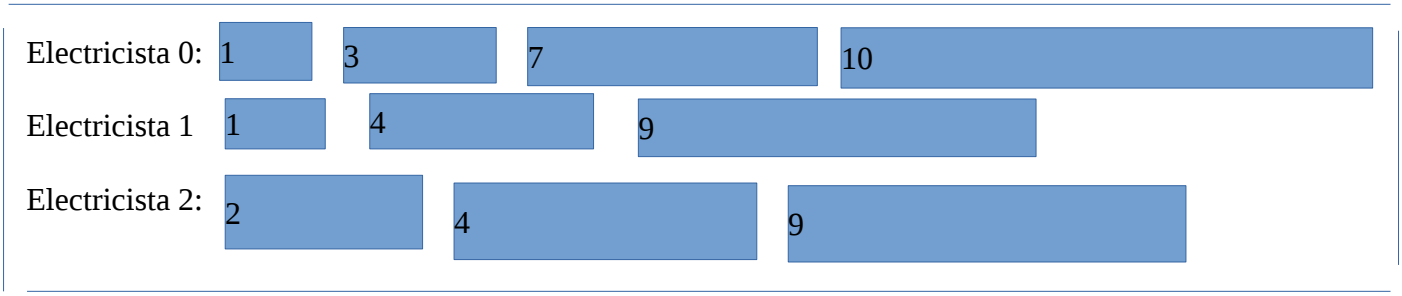
1	1	2

1 3	1 4	2 4

1 3 7	1 4 9	2 4 9

1 3 7 10	1 4 9	2 4 9

Lo que nos quedaría de una manera más gráfica:



vector <vector<int> >trabajo

para realizar esta tarea hemos realizado el siguiente código en C++:

```
//Funcion para n electricistas
void varioselectricistas(vector<vector<int> > trabajo,vector<int> clientes, int
nelectricistas){
int contador = 0;

trabajo.reserve(nelectricistas);
sort(clientes.begin(), clientes.end());

//A cada vector del vector de vectores se le va añadiendo tareas segun los
electricistas que se metan
for(int i = 0; i < clientes.size(); i++){
        if(contador < nelectricistas){
                trabajo[contador].push_back(clientes[i]);
                contador++;
        }
        else{
                contador = 0;
                i--;
        }
}

}
```

Optimalidad:
El electricista necesita hacer n reparaciones, sabe de antemano el tiempo que le va a llevar cada una de ellas: en la tarea i-ésima tardará t_i minutos.
Necesita decidir el orden en el que atiende a los clientes ya que en su empresa le pagan por la satisfacción del cliente, por lo que necesitara decidir el orden en que atiene para minimizar el tiempo medio de espera de los clientes.
Por tanto si llamamos E_i a lo que espera el cliente i-ésimo hasta tener reparada su avería por completo, necesitamos minimizar la expresión:
$E(n)=\Sigma^{n}_{i=1} \quad E_i$
En primer lugar hemos de observar que el electricista siempre tardará el mismo tiempo global T = t 1 + t 2 + ... + t n en realizar todas las reparaciones, independientemente de la forma en que las ordene. Sin embargo, los tiempos de espera de los clientes sí dependen de esta ordenación.
En efecto, si mantiene la ordenación original de las tareas (1, 2, ..., n), la expresión de los tiempos de espera de los clientes viene dada por:
$E_1 = t\ 1$
$E_2 = t\ 1 + t\ 2$
.....
$E_n = t\ 1 + t\ 2 + ... + t\ n$.

Lo que queremos encontrar es una permutación de las tareas en donde se minimice la expresión de E(n) que, basándonos en las ecuaciones anteriores, viene dada por:

$$E(n) = \sum_{i=1}^{n} E_i = \sum_{i=1}^{n} (n-i+1)t_i \ .$$

Vamos a demostrar que la permutación óptima es aquella en la que los avisos se atienden en orden creciente de sus tiempos de reparación.
Para ello, denominemos X = (x 1 ,x 2 ,...,x n) a una permutación de los elementos:
(1,2,...,n), y sean (s 1 ,s 2 ,...,s n) sus respectivos tiempos de ejecución, es decir, (s 1 ,s 2 ,...,s n) va a ser una permutación de los tiempos orginales (t 1 ,t 2 ,...,t n).
Supongamos que no está ordenada en orden creciente de tiempo de reparación, es decir, que existen dos números $x_i < x_j$ tales que $s_i > s_j$.
Sea Y = (y 1 ,y 2 ,...,y n) la permutación obtenida a partir de X intercambiando x_i con x_j , es decir, $y_k = x_k$ si $k \neq i$ y $k \neq j$, $y_i = x_j$, $y_j = x_i$.
Si probamos que E(Y) < E(X) habremos demostrado lo que buscamos, pues mientras más ordenada (según el criterio dado) esté la permutación, menor tiempo de espera supone. Pero para ello, basta darse cuenta que:

$$E(Y) = (n - x_i + 1)s_j + (n - x_j + 1)s_i + \sum_{k=1, k \neq i, k \neq j}^{n} (n - k + 1)s_k$$

y que, por tanto:

$$E(X) - E(Y) = (n - x_i + 1)(s_i - s_j) + (n - x_j + 1)(s_j - s_i) = (x_j - x_i)(s_i - s_j) > 0.$$

En consecuencia, el algoritmo pedido consiste en atender a las llamadas en orden inverso a su tiempo de reparación. Con esto conseguirá minimizar el tiempo medio de espera de los clientes, tal y como hemos probado.

Optimalidad para más de un electricista:

Supongamos que en la empresa del electricista aumenta el número E electricistas para realizar las n tareas.
En primer lugar, se ordenan los avisos por orden creciente de tiempo de reparación.
Un vez hecho esto, se van asignando los avisos por este orden, siempre al fontanero menos ocupado. En caso de haber varios con el mismo grado de ocupación, se escoge el de número menor.
En otras palabras, si los avisos están ordenados de forma que $t_i \leq t_j$ si $i < j$, asignaremos al fontanero k los avisos k, k+E, k+2E, …
Como para 1 electricista ya se ha demostrado la optimalidad, para n electricistas también sera óptimo de la misma manera.

Códigos en c++:

Ejercicio 1:

```
/*
Se tiene un buque mercante cuya capacidad de carga es de K
toneladas y un conjunto de
contenedores c1 , . . . , cn cuyos pesos respectivos son p1 , . . . , pn
(expresados también en
toneladas).
Teniendo en cuenta que la capacidad del buque es menor que la suma
total de los pesos
de los contenedores:
Diseñamos un algoritmo que maximice el número de contenedores
cargados. Demostrar su optimalidad.
*/
//COMPILACION: g++ numerodetoneladascargados.cpp -o num
-std=gnu++0x

#include <iostream>
#include <vector>
#include <set>
#include <cstdlib>
#include <chrono>
using namespace std;

//Entrada: vector de contenedores y la capacidad maxima del barco.
//Devuelve un vector con los contenedores del barco.
vector<int> llenaBarcoContenedores(multiset<int, less<int> > c ,int
peso_max, int &peso_actual) {
    vector<int> s;
    int peso_contenedor;
    int num_contenedor;
    peso_actual = 0;
    bool lleno = false;

    while( !c.empty() && !lleno ) {
        peso_contenedor = *(c.begin());//Seleccionamos el contenedor
más ligero.

        if((peso_actual + peso_contenedor) <= peso_max) { //Funcion de
factibilidad
            s.push_back(peso_contenedor);       //Introducimos el
contenedor
```

```
            peso_actual += peso_contenedor;     //en el barco.

            c.erase(c.begin());  //Eliminamos el contenedor de la lista.
        }
        else
            lleno = true;
    }
    return s;
}

//Genera un vector de tamaños aleatorios entre 1 y 100;
multiset<int, less<int> > genera(int n) {
    multiset<int, less<int> > c;
    srand(time(0));

    for(int i=0; i < n; i++)
        c.insert( (rand()%100)+1 );

    return c;
}
int main(int argc, char *argv[]) {
    if(argc != 2) {
        cerr << "Formato " << argv[0] << " <numero contenedores>" <<
"\n";
        return -1;
    }
    std::chrono::high_resolution_clock::time_point tantes, tdespues;
    std::chrono::duration<double> transcurrido;

    int n = atoi(argv[1]);
    int peso_actual = 0;
    int peso_max = 30*n;   //Tamaño de los contenedores en funcion del
nuemero de contenedores

    multiset<int, less<int> > contenedores;
    contenedores = genera(n);
    vector <int> barco;

    tantes = std::chrono::high_resolution_clock::now();     //Tiempo antes
de la operación
    barco = llenaBarcoContenedores(contenedores,peso_max,
peso_actual);
```

```
    tdespues = std::chrono::high_resolution_clock::now();   //Tiempo
después de la operación
    transcurrido =
std::chrono::duration_cast<std::chrono::duration<double>>(tdespues -
tantes);

    cout << "Tiempo: " << transcurrido.count() << endl;
    cout << "\nPara " << n << " contenedores y una carga maxima de " <<
peso_max << endl;
    cout << "en el barco sobran " << peso_max - peso_actual << "
unidades de peso." << endl;
    cout << "Contenedores cargados: " << barco.size() << endl;

    cout << "\nCarga del barco: " << endl;
    for(int i=0; i < barco.size(); i++)
        cout << "[" << i+1 << "]" << barco[i] << "\t";
    cout << endl;

    cout << "\nContenedores a cargar: " << endl;
    for(auto it = contenedores.begin(); it != contenedores.end(); it++)
        cout << *it << " - ";
    cout << endl;
}


/*
Se tiene un buque mercante cuya capacidad de carga es de K
toneladas y un conjunto de
contenedores c1 , . . . , cn cuyos pesos respectivos son p1 , . . . , pn
(expresados también en
toneladas).
Teniendo en cuenta que la capacidad del buque es menor que la suma
total de los pesos
de los contenedores:
Diseñamos un algoritmo que maximice las toneladas cargadas.
*/

//COMPILACION: g++ numerodetoneladascargados.cpp -o num
-std=gnu++0x

#include <iostream>
#include <vector>
```

```
#include <set>
#include <cstdlib>
#include <chrono>
using namespace std;

//Entrada: vector de contenedores y la capacidad maxima del barco.
//Devuelve un vector con los contenedores del barco.
vector<int> llenaBarcoPeso(multiset<int, greater<int> > c ,int peso_max,
int &peso_actual) {
    vector<int> s;
    int peso_contenedor;
    int num_contenedor;
    peso_actual = 0;
    bool lleno = false;

    while( !c.empty() && !lleno ) {
        peso_contenedor = *(c.begin());//Seleccionamos el contenedor
más ligero.

        if((peso_actual + peso_contenedor) <= peso_max) { //Funcion de
factibilidad
            s.push_back(peso_contenedor);       //Introducimos el
contenedor
            peso_actual += peso_contenedor;     //en el barco.

            c.erase(c.begin());  //Eliminamos el contenedor de la lista.
        }
        else
            lleno = true;
    }
    return s;
}

//Genera un vector de tamaños aleatorios entre 1 y 100;
multiset<int, greater<int> > genera(int n) {
    multiset<int, greater<int> > c;
    srand(time(0));

    for(int i=0; i < n; i++)
        c.insert( (rand()%100)+1 );

    return c;
```

```
}

int main(int argc, char *argv[]) {
    if(argc != 2) {
        cerr << "Formato " << argv[0] << " <numero contenedores>" <<
"\n";
        return -1;
    }

    std::chrono::high_resolution_clock::time_point tantes, tdespues;
    std::chrono::duration<double> transcurrido;

    int n = atoi(argv[1]);
    int peso_actual = 0;
    int peso_max = 30*n;   //Tamaño de los contenedores en funcion del
nuemero de contenedores

    multiset<int, greater<int> > contenedores;
    contenedores = genera(n);
    cout << contenedores.size() << endl;
    vector <int> barco;

    tantes = std::chrono::high_resolution_clock::now();     //Tiempo antes
de la operación
    barco = llenaBarcoPeso(contenedores,peso_max, peso_actual);
    tdespues = std::chrono::high_resolution_clock::now();   //Tiempo
después de la operación
    transcurrido =
std::chrono::duration_cast<std::chrono::duration<double>>(tdespues -
tantes);

    cout << "Tiempo: " << transcurrido.count() << endl;
    cout << "\nPara " << n << " contenedores y una carga maxima de " <<
peso_max << endl;
    cout << "en el barco sobran " << peso_max - peso_actual << "
unidades de peso." << endl;
    cout << "Peso cargado: " << peso_actual << "/" << peso_max <<
endl;
    cout << "\nCarga del barco: " << endl;
    for(int i=0; i < barco.size(); i++)
        cout << "[" << i+1 << "]" << barco[i] << "\t";
    cout << endl;
```

```
    cout << "\nContenedores a cargar: " << endl;
    for(auto it = contenedores.begin(); it != contenedores.end(); it++)
        cout << *it << " - ";
    cout << endl;
}
```

Ejercicio 2:

```
#include <iostream>
#include <set>
#include <stdlib.h>

using namespace std;

set<int> NumeroDiasAperturaAleatorio(int dias_a_abrir)
{
	set<int> dias_apertura;
	int numero_aleatorio;
	set<int>::iterator it;

	while(dias_apertura.size() < dias_a_abrir)
	{
		numero_aleatorio=1+rand()%(32-1); //aleatorio entre 1 y 31
		dias_apertura.insert(numero_aleatorio);
	}

	return dias_apertura;

}

int ConsumoMaximo()
{
	return 1+rand()%(14-1);

}

void DiasAacudir(set<int> dias_apertura,int duracion)
{
	set<int> resultado;
	int d=duracion;

	set<int>::iterator it=dias_apertura.begin();
```

```
	set<int>::iterator a=dias_apertura.begin();
	a++;

	for(int i=1;i<=31 && it!=dias_apertura.end();i++)
	{
		if(duracion < *it)
		{
			cout << "NO SOLUCION" << endl;
			i=32;
		}
		else if( (duracion>=*it) && ( duracion < *a ) )
		{
			cout << "El granjero va a la tienda el día " << *it << endl;
			duracion=*it+d;
		}

		else if(i==31)
		{
			if(duracion==31)
				cout << "El granjero va a la tienda el último día del
mes" << endl;
		}
		a++;
		it++;
	}
	return resultado;
}
int main(int argc, char *argv[])
{

	if(argc < 1){
		cout << "Introduce el numero de días que quiere que abra la
tienda" << endl;
		return 0;
	}

	int tamanio = atoi(argv[1]);
	int r=ConsumoMaximo();
	set<int> prueba=NumeroDiasAperturaAleatorio(tamanio);

	set<int> resultado;
	set<int>::iterator it;
```

```
	DiasAacudir(prueba,r);
}
```

Ejercicio 5:

```
#include<iostream>
#include<stdlib.h>
#include<vector>
#include <algorithm>
using namespace std;

//Funcion para n electricistas
void varioselectricistas(vector<vector<int> > trabajo,vector<int> clientes,
int nelectricistas){
int contador = 0;

trabajo.reserve(nelectricistas);
sort(clientes.begin(), clientes.end());

//A cada vector del vector de vectores se le va añadiendo tareas segun
los electricistas que se metan
for(int i = 0; i < clientes.size(); i++){
	if(contador < nelectricistas){
		trabajo[contador].push_back(clientes[i]);
		contador++;
	}
	else{
		contador = 0;
		i--;
	}
}

}

int main(){
int nclientes = 10;
vector<int> clientes;
int suma = 0;
int consumido = 0;
vector<vector<int> > trabajo;
int nelectricistas = 3;
```

```
clientes.push_back(1);
clientes.push_back(4);
clientes.push_back(2);
clientes.push_back(3);
clientes.push_back(4);
clientes.push_back(9);
clientes.push_back(1);
clientes.push_back(7);
clientes.push_back(10);
clientes.push_back(9);

sort(clientes.begin(),clientes.end());
//Mostrar el vector:
cout << "Lista de tiempos asignados a los clientes: " << endl;
for (int i = 0; i < clientes.size(); i++){
        suma += clientes[i];
        cout << clientes[i] << endl;
}
cout << "\nTiempo total para atender clientes: " << suma << " min" <<
endl;

//Atender a clientes
for(int i = 0; i < clientes.size(); i++){
int restante = 0;
        consumido+=clientes[i];
        restante = suma - consumido;
        cout << "Cliente n" << i << ": electricista ha consumido " <<
consumido << " min, para terminar le quedan" << restante << " min" <<
endl;
}

}
```

Bibliografía

Fundamentos de algoritmica – G. Brassard, P. Bratley

Fundamentals of Computer Algorithms - Ellis Horowitz, Sartaj Sahni