Algoritmia y Complejidad

Ejercicios tema 2

Laboratorio Miércoles 15:00-17:00

Integrantes:

Carlos Javier Hellín Asensio
Diego Gutiérrez Marcos

Curso: 2020-2021

Problema 3

Descripción del algoritmo:
El problema pide buscar el mínimo y el máximo con 3n/2 comparaciones de un vector V formado por n datos. Para ello, el algoritmo primero recorre la mitad del vector, es decir, hasta n/2 y comprueba si los valores de esa mitad es mayor que los valores de la mitad restante (el opuesto) para intercambiarlo y así dejar los valores menores a la izquierda y los mayores a la derecha del vector.
Una vez hecho esto, el algoritmo pasa a buscar el mínimo recorriendo el vector desde 0 hasta n/2 del vector, y luego se busca el máximo desde n/2 hasta n (la otra mitad restante).
En total se hacen un máximo de 3n/2 comparaciones como así pide el problema.

Casos de prueba:
El primer caso de prueba consiste en tener un vector de una longitud n creada aleatoriamente por la función randint (esta función recibe dos números como parámetros entre los cuales queremos que se genere un número aleatorio entero, él cuál será la longitud que tendrá el vector). Luego creamos los valores correspondientes al vector siendo su longitud la antes obtenida. Tras ello imprimimos por pantalla el resultado del vector de longitud n obtenido. Posteriormente realizamos el algoritmo de búsqueda indicado en el ejercicio y el vector resultante consiste en que la 1ª mitad tenga los valores más pequeños en su izquierda con respecto a sus valores espejo en la 2ª mitad. Es decir, el resultado del vector antes de finalizar completamente el algoritmo es el segundo vector indicado en la captura posterior. Finalmente, tras las búsquedas, en la 1ª mitad del menor y del mayor en la 2ª se llega a los valores máximos y mínimos correspondientes. Este caso de prueba lo hemos realizado para comprobar que el algoritmo funciona independientemente de si la longitud del vector es muy grande y que cualquier valor del vector generado no da ningún problema ya sean estos últimos tanto valores negativos como positivos.

```
El vector generado es:
[-4, -58, 49, -45, 31, 73, 45, -73, -77, -67]
Se recorre la mitad y si los valores de la izquierda es mayor que los de la derecha, se intercambian
[-67, -77, -73, -45, 31, 73, 45, 49, -58, -4]
El valor mínimo es: -77 y el valor máximo es 73
```

Funcionamiento del algoritmo en el ejemplo: En esta primera imagen se puede observar la creación de un vector de forma aleatoria ya que lo realizamos mediante la utilización de varios randint que escogen un valor

entero entre los números que les pasa como argumento. Tras la obtención de este vector se recorre la mitad del vector generado y realiza una comparación entre las dos mitades en las que hemos dividido el vector. Si un valor de la mitad izquierda es mayor que el valor opuesto en la parte derecha, estos valores son intercambiados. Como se puede ver en la imagen anterior, esto ocurre con el primer valor del vector(-4) y el último valor(-67) ya que este es menor al primero se intercambian. Como se puede observar estos intercambios se realizan entre -58 y -77, 49 y -73. Tras haber ya intercambiado los valores mostramos como quedaría el vector tras lo intercambios y entonces para encontrar el mínimo solo realizamos una búsqueda en la 1ª mitad del vector resultante tras los intercambios y para encontrar el máximo hacemos una búsqueda del mayor en la 2ª mitad de ese vector así obteniendo el máximo y el mínimo y mostrándolo por pantalla

El segundo caso de prueba lo utilizamos para comprobar que los vectores de longitud par, no dan ningún error. El valor intermedio al realizar este algoritmo es el segundo vector mostrado en la captura tras haber realizado los cambios por los valores espejo dependiendo si son mayores o menores.

```
[30, 20, 10, 40]
Se recorre la mitad y si los valores de la izquierda es mayor que los de la derecha, se intercambian
[30, 10, 20, 40]
El valor mínimo es: 10 y el valor máximo es 40
```

Y el último caso de prueba realizado es para comprobar el funcionamiento del algoritmo en vectores de longitud impar. En este caso el valor intermedio del vector no varía respecto al primero ya que los valores espejos del 1ª mitad y 2ª mitad ya están en sus respectivas posiciones tras comparar sus valores.

```
[56, -39, 73]
Se recorre la mitad y si los valores de la izquierda es mayor que los de la derecha, se intercambian
[56, -39, 73]
El valor mínimo es: -39 y el valor máximo es 73
```

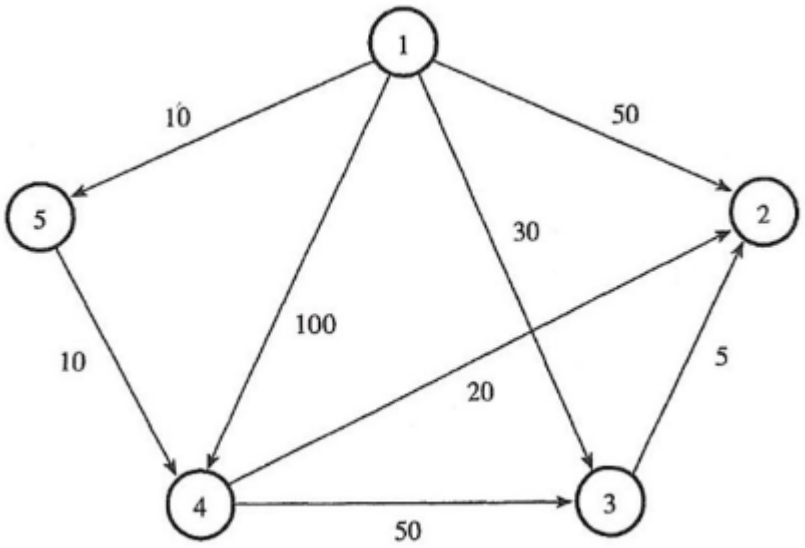
Problema 5

Descripción del algoritmo:
Se ha implementado el algoritmo de Dijkstra visto en teoría en pseudocódigo y teniendo en cuenta las ideas dadas del problema.
El algoritmo tiene una serie de nodos como candidatos temporales (excepto el nodo 1) y unas distancias obtenidas de la matriz con los costes del grafo. Se realiza un bucle voraz desde 0 hasta n - 1 nodos para seleccionar como candidato el que tenga menor distancia temporal conocida y se elimina del

conjunto de vértices no recorridos. Una vez seleccionado dicho nodo se actualiza el resto de distancias temporales usando el nodo actual, comprobando si hay una alternativa mejor o se queda como estaba.

Una de las ideas que pide el problema es almacenar la forma de recorrer el grafo desde el nodo 1 al nodo n. Para ello, se ha creado un array de los predecesores, es decir, se almacena el nodo de “dónde vengo” cuando se actualizan el resto de distancias temporales. De esta forma, el algoritmo puede crear un camino mirando el predecesor del nodo n, luego se mira el predecesor del anterior predecesor, etc.. así hasta completar el camino hasta el nodo 1.

Casos de prueba:
En este primer caso de prueba hemos elegido el enunciado puesto en los apuntes de teoría para así comprobar que hemos llegado de forma correcta al resultado siguiendo el algoritmo de Dijkstra. Como en python la asignación de los array empieza por 0 todos los vértices son un número menor al que se encuentra en la imagen del grafo. En el primer paso se selecciona el valor cuya longitud sea menor entre los cuatro nodos que tienen una arista que conecta esos nodos con el nodo 1. En este paso se indican las diferentes distancias de los cuatros nodos con el primero y se selecciona la distancia más pequeña en este caso el 4 (recuerdo que hemos empezado con el nodo 0 por python). Además también se quedan almacenados los distintos nodos que no han sido seleccionados. Tras este primer paso realizamos un bucle realizando las mismas operaciones para comparar nodos (siempre escogiendo el valor con distancia menor) hasta que hayamos recorrido todos los nodos existentes. Dando como valor final [1, 5], el camino mínimo entre los nodos 1 (0 en python) y el caso n, en este caso de prueba 5 (o 4 en python), y [10], la longitud de este camino

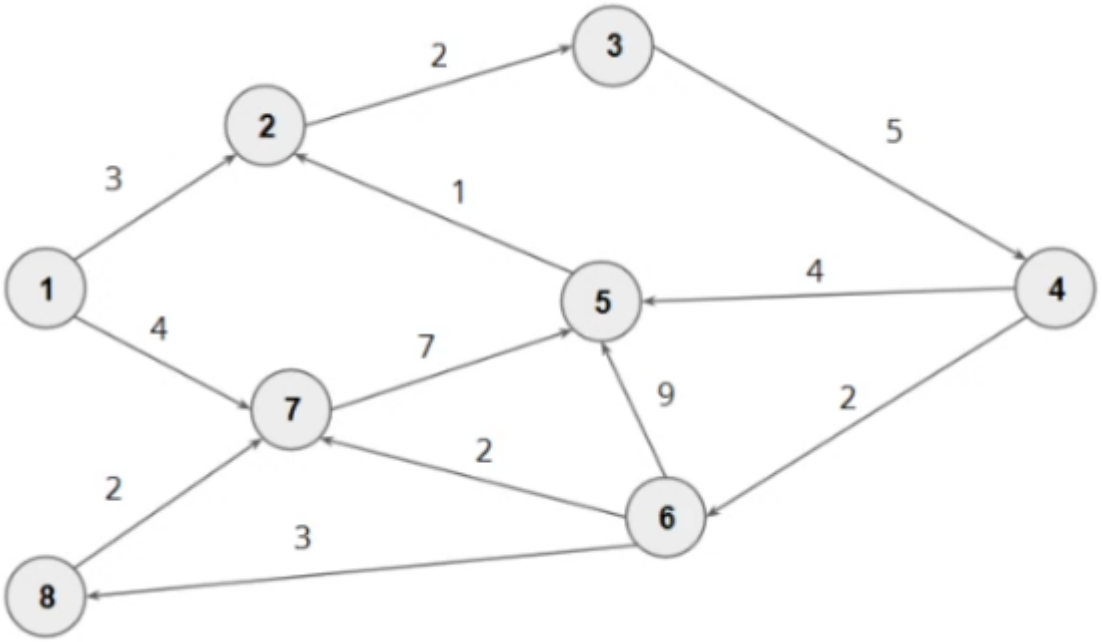



```
Paso: 0 Selección: 4 C: [1, 2, 3] D: [50, 30, 100, 10]
Paso: 1 Selección: 3 C: [1, 2] D: [50, 30, 20, 10]
Paso: 2 Selección: 2 C: [1] D: [40, 30, 20, 10]
Paso: 3 Selección: 1 C: [] D: [35, 30, 20, 10]
Camino mínimo entre los nodos 1 y 5 : [1, 5] con longitud 10
```

Funcionamiento del algoritmo en el ejemplo: En la primera imagen mostramos el grafo en el que vamos a realizar el algoritmo de Dijkstra para que se nos muestre el camino más corto del primer vértice al último además de la longitud de ese camino. Como en python la asignación de los array empieza por 0 reducimos todos los vértices del grafo en 1 es decir que los vértices serán 0(1 antes),1(2 antes),2(3 antes),3(4 antes) y 4(5 antes). Para crear el grafo creamos una matriz que está inicializada al coste de la arista del vértice 0 a cada vértice j y si no existe arista que relacione los vértices como +math.inf. Tras ello realizamos el algoritmo de Dijkstra, el cual en el primer paso (paso 0)se selecciona el vértice 4 ya que es el camino más pequeño de longitud como se puede ver en el array en el que mostramos todas las longitudes (D) entre el vértice 0 y el resto ordenados por vértice, es decir, primero el vértice 1, luego el 2, tras este el 3 y por último el 4, y entonces se quedan como candidatos (C) los vértices por los que aún no hemos pasado. En el segundo paso se han actualizado las distancias respecto al vértice actual escogiendo las que tienen una distancia más pequeña. Por tanto, volvemos a elegir el vértice con una distancia menor al vértice de los que quedaban entre los candidatos y en este paso 1 es el vértice 3 y tras ello actualizamos los candidatos solo quedando en el los vértices 1 y 2 como opciones. Otras vez realizamos la actualización de las distancias al vértice, escogemos el de distancia más pequeña de los disponibles en los candidatos y actualizamos los candidatos hasta que ya no haya un candidato disponible

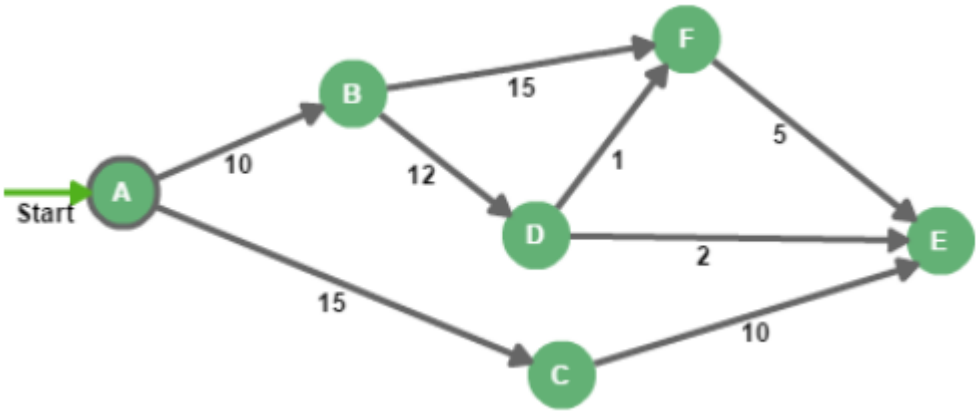
entonces en una variable en la que se guardaban los valores del camino se imprime por pantalla junto a la longitud del camino empleado. En este caso como se puede observar el camino más corto entre el vértice 0 y el 4 es la conexión existente entre ambos de una longitud 10

En este segundo caso de prueba hemos elegido el enunciado puesto en las diapositivas de clase para así comprobar que hemos llegado de forma correcta al resultado siguiendo el algoritmo de Dijkstra con un número de nodos superior al primer caso. Como en python la asignación de los array empieza por 0 todos los vértices son un número menor al que se encuentra en la imagen del grafo. En el primer paso se selecciona el valor cuya longitud sea menor entre los cuatro nodos que tienen una arista que conecta esos nodos con el nodo 1. En este paso se indican las diferentes distancias de los 8 nodos con el primero, siendo los no conectados infinito, y se selecciona la distancia más pequeña en este caso el 1 (recuerdo que hemos empezado con el nodo 0 por python). Además también se quedan almacenados los distintos nodos que no han sido seleccionados. Tras este primer paso realizamos un bucle realizando las mismas operaciones para comparar nodos (siempre escogiendo el valor con distancia menor) hasta que hayamos recorrido todos los nodos existentes. Dando como valor final [1, 2, 3, 4, 6, 8], el camino mínimo entre los nodos 1 (0 en python) y el caso n, en este caso de prueba 8 (o 7 en python), y [15], la longitud de este camino.



```
Paso: 0 Selección: 1 C: [2, 3, 4, 5, 6, 7] D: [3, inf, inf, inf, inf, 4, inf]
Paso: 1 Selección: 6 C: [2, 3, 4, 5, 7] D: [3, 5, inf, inf, inf, 4, inf]
Paso: 2 Selección: 2 C: [3, 4, 5, 7] D: [3, 5, inf, 11, inf, 4, inf]
Paso: 3 Selección: 3 C: [4, 5, 7] D: [3, 5, 10, 11, inf, 4, inf]
Paso: 4 Selección: 4 C: [5, 7] D: [3, 5, 10, 11, 12, 4, inf]
Paso: 5 Selección: 5 C: [7] D: [3, 5, 10, 11, 12, 4, inf]
Paso: 6 Selección: 7 C: [] D: [3, 5, 10, 11, 12, 4, 15]
Camino mínimo entre los nodos 1 y 8 : [1, 2, 3, 4, 6, 8] con longitud 15
```

En este tercer caso de prueba hemos elegido este grafo cogido de internet para hacer una última comprobación del correcto funcionamiento del algoritmo. Como en python la asignación de los array empieza por 0 todos los vértices son un número menor al que se encuentra en la imagen del grafo. En el primer paso se selecciona el valor cuya longitud sea menor entre los cuatro nodos que tienen una arista que conecta esos nodos con el nodo A. En este paso se indican las diferentes distancias de los 6 nodos con el primero, siendo los no conectados infinito, y se selecciona la distancia más pequeña en este caso el 1 (nodo B). Además también se quedan almacenados los distintos nodos que no han sido seleccionados. Tras este primer paso realizamos un bucle realizando las mismas operaciones para comparar nodos (siempre escogiendo el valor con distancia menor) hasta que hayamos recorrido todos los nodos existentes. Dando como valor final [1, 2, 4, 6], el camino mínimo entre los nodos A (0 en python) y el caso n, en este caso de prueba F (o 5 en python), y [23], la longitud de este camino.




```
Paso: 0 Selección: 1 C: [2, 3, 4, 5] D: [10, 15, inf, inf, inf]
Paso: 1 Selección: 2 C: [3, 4, 5] D: [10, 15, 22, inf, 25]
Paso: 2 Selección: 3 C: [4, 5] D: [10, 15, 22, 25, 25]
Paso: 3 Selección: 5 C: [4] D: [10, 15, 22, 24, 23]
Paso: 4 Selección: 4 C: [] D: [10, 15, 22, 24, 23]
Camino mínimo entre los nodos 1 y 6 : [1, 2, 4, 6] con longitud 23
```

Problema 6

Descripción del algoritmo:
Este algoritmo lo primero que hace es ordenar la lista de las escaleras de menor a mayor. Una vez hecho esto, se obtiene el primero y segundo valores menores de la lista. Hace una suma de ambos menores y esa suma lo añade a la lista de las escaleras para que en la siguiente iteración vuelva a ordenarse de menor a mayor y se repita el mismo proceso, buscando así la forma óptima de fundir las escaleras.
Esta suma se va acumulando para obtener el coste total y también como resultado de los minutos de soldar cada escaleras.

Se entiende por el problema que debería haber al menos dos escaleras para poder soldar, por lo tanto el algoritmo solo se mantiene en el bucle cuando haya dos o más escaleras.

Casos de prueba:
Realizamos el caso de prueba indicado en la fotografía de abajo porque queremos comprobar que al aplicar el algoritmo implementado los minutos en soldar serian menores a realizar la soldadura de forma aleatoria.

```
Escaleras: [6, 8, 7]
Iteración: 0
Se ordenan las escaleras de menor a mayor [6, 7, 8]
Primer menor: 6 Segundo menor: 7
Suma de ambos menores: 6 + 7 = 13
La suma se le añade a las escaleras: [8, 13]
Se añade al total: 13

Iteración: 1
Se ordenan las escaleras de menor a mayor [8, 13]
Primer menor: 8 Segundo menor: 13
Suma de ambos menores: 8 + 13 = 21
La suma se le añade a las escaleras: [21]
Se añade al total: 34

Los minutos de soldar son los siguientes: [13, 21] que en total son 34 minutos
```

Funcionamiento del algoritmo: El algoritmo recibe como parámetro las longitudes de las escaleras que se quieren soldar en este caso [6,8,7]. Tras recibir ese array el algoritmo ordena de menor a mayor su contenido quedando [6,7,8]. Posteriormente se cogen en dos variables los dos primeros números del array(ya que son los dos más pequeños, en este caso 6 y 7) en

el momento de su eliminación del array y finalmente los sumamos(6+7=13). El resultado de la suma se le añade al array en el que se contienen las escaleras (escaleras = [8,13]), el cual se volverá a ordenar de menor a mayor en la siguiente iteración y también se añade este resultado a la variable total, en la cual se guardan los minutos que se tarda en soldar dos escaleras(total =13). Estas iteraciones se seguirán realizando mientras haya dos o más escaleras dentro de esa variable. En este caso solo se necesita una iteración más porque tras la suma de las dos escaleras del array (8+13=21) únicamente la escalera tendría un valor(en este caso 21)y se le sumaría a la variable total el resultado de la suma (total=21+13=34). Por tanto, el tiempo empleado en soldar las escaleras sería de 34 minutos.

En este segundo caso de prueba comprobamos que el algoritmo siga funcionando aunque haya escaleras de una misma longitud a la hora de cuánto tiempo se tardaría en soldar.

```
Escaleras: [4, 3, 2, 2, 8]
Iteración: 0
Se ordenan las escaleras de menor a mayor [2, 2, 3, 4, 8]
Primer menor: 2 Segundo menor: 2
Suma de ambos menores: 2 + 2 = 4
La suma se le añade a las escaleras: [3, 4, 8, 4]
Se añade al total: 4

Iteración: 1
Se ordenan las escaleras de menor a mayor [3, 4, 4, 8]
Primer menor: 3 Segundo menor: 4
Suma de ambos menores: 3 + 4 = 7
La suma se le añade a las escaleras: [4, 8, 7]
Se añade al total: 11

Iteración: 2
Se ordenan las escaleras de menor a mayor [4, 7, 8]
Primer menor: 4 Segundo menor: 7
Suma de ambos menores: 4 + 7 = 11
La suma se le añade a las escaleras: [8, 11]
Se añade al total: 22

Iteración: 3
Se ordenan las escaleras de menor a mayor [8, 11]
Primer menor: 8 Segundo menor: 11
Suma de ambos menores: 8 + 11 = 19
La suma se le añade a las escaleras: [19]
Se añade al total: 41

Los minutos de soldar son los siguientes: [4, 7, 11, 19] que en total son 41 minutos
```

En este tercer caso de prueba comprobamos que si el algoritmo funciona correctamente si hay un número n de escaleras creado de forma aleatoria.

```
Escaleras: [1, 5, 9, 8, 10, 10, 5]
Iteración: 0
Se ordenan las escaleras de menor a mayor [1, 5, 5, 8, 9, 10, 10]
Primer menor: 1 Segundo menor: 5
Suma de ambos menores: 1 + 5 = 6
La suma se le añade a las escaleras: [5, 8, 9, 10, 10, 6]
Se añade al total: 6

Iteración: 1
Se ordenan las escaleras de menor a mayor [5, 6, 8, 9, 10, 10]
Primer menor: 5 Segundo menor: 6
Suma de ambos menores: 5 + 6 = 11
La suma se le añade a las escaleras: [8, 9, 10, 10, 11]
Se añade al total: 17

Iteración: 2
Se ordenan las escaleras de menor a mayor [8, 9, 10, 10, 11]
Primer menor: 8 Segundo menor: 9
Suma de ambos menores: 8 + 9 = 17
La suma se le añade a las escaleras: [10, 10, 11, 17]
Se añade al total: 34

Iteración: 3
Se ordenan las escaleras de menor a mayor [10, 10, 11, 17]
Primer menor: 10 Segundo menor: 10
Suma de ambos menores: 10 + 10 = 20
La suma se le añade a las escaleras: [11, 17, 20]
Se añade al total: 54

Iteración: 4
Se ordenan las escaleras de menor a mayor [11, 17, 20]
Primer menor: 11 Segundo menor: 17
Suma de ambos menores: 11 + 17 = 28
La suma se le añade a las escaleras: [20, 28]
Se añade al total: 82

Iteración: 5
Se ordenan las escaleras de menor a mayor [20, 28]
Primer menor: 20 Segundo menor: 28
Suma de ambos menores: 20 + 28 = 48
La suma se le añade a las escaleras: [48]
```

```
Los minutos de soldar son los siguientes: [6, 11, 17, 20, 28, 48] que en total son 130 minutos
```

En este último caso de prueba error lo hacemos para comprobar que sale error si únicamente haya una escalera por tanto el algoritmo no se podría realizar correctamente

```
Escaleras: [4]
0 : el coste total es cero si solo hay una escalera para soldar
```