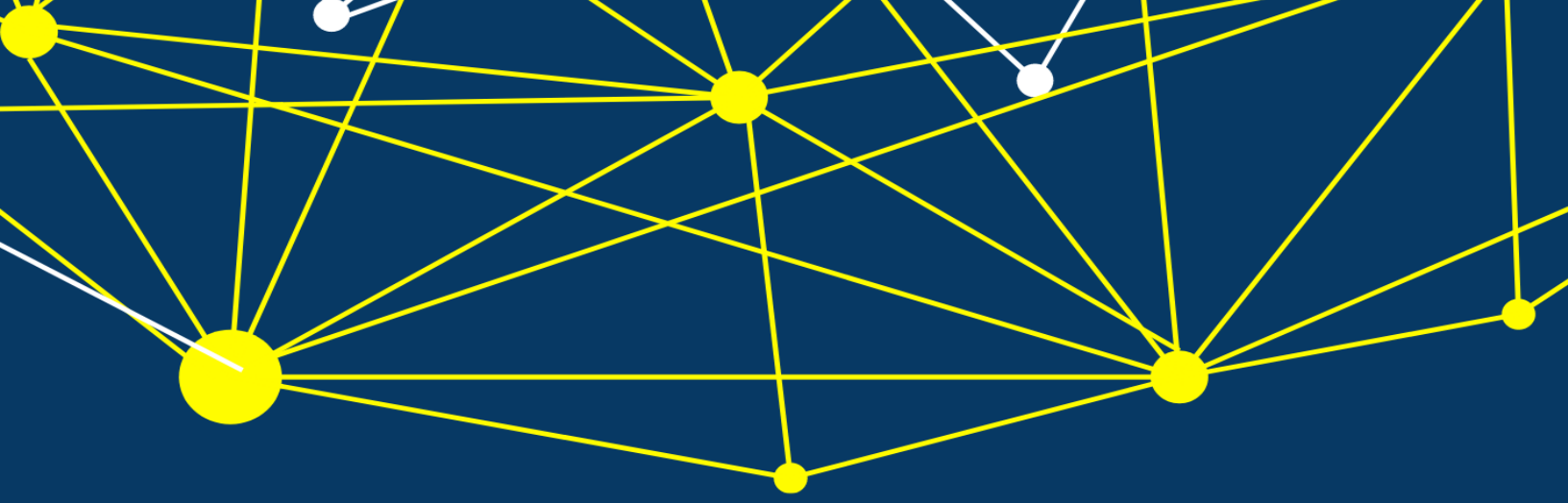


Adrián Borges Cano

Marco González Martínez

Laboratorio: Miércoles 15:00 - 17:00

Algoritmia

Tema 2: Algoritmos Voraces

Tabla de contenido

PROBLEMA 3:

Se dispone de un vector V formado por n datos, del que se quiere encontrar el elemento mínimo del vector y el elemento máximo del vector. El tipo de los datos que hay en el vector no es relevante para el problema, pero la comparación entre dos datos para ver cuál es menor es muy costosa, por lo que el algoritmo para la búsqueda del mínimo y del máximo debe hacer la menor cantidad de comparaciones entre elementos posible.

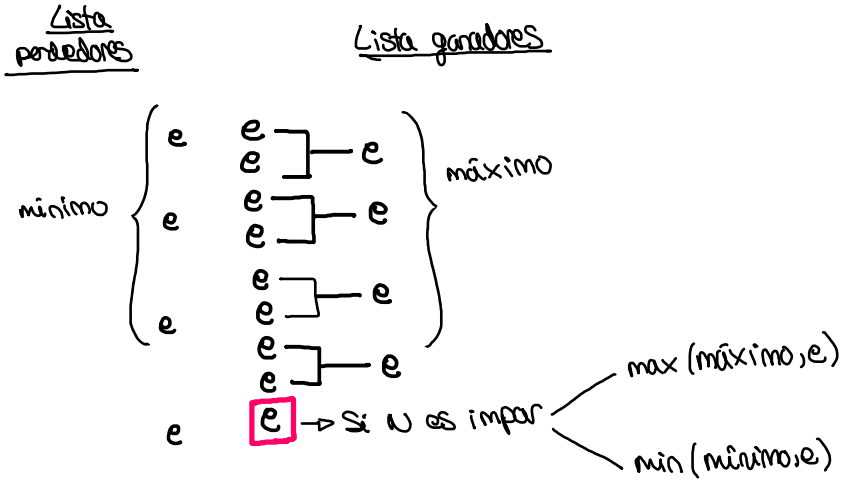
Un método trivial consiste en un recorrido lineal del vector para buscar el máximo y después otro recorrido para buscar el mínimo, lo que requiere un total de aproximadamente ente 2n comparaciones entre datos. Este método no es lo suficientemente rápido, por lo que se pide implementar un método con metodología Voraz que realice un máximo de $3n/2$ comparaciones.

Solución adoptada:

Para poder resolver este problema lo más eficiente posible, hemos optado por implementar un algoritmo que aplica el simple concepto de "rey de la pista" (el elemento "ganador" mantiene su puesto, mientras que el perdedor queda eliminado). Dividimos los elementos del vector original en dos nuevos vectores, de la mitad de tamaño cada uno de ellos. Una vez tengamos divididos ambos vectores (donde uno almacenará los elementos mayores, y otro, los elementos menores).

Al realizar este procedimiento de "rey de la pista", el primer elemento de cada vector será comparado con el resto de ellos. Si cumple la condición (que el primer elemento sea el mayor en el primer vector, y que sea el menor en el segundo) se mantendrá, de lo contrario, se eliminará y el elemento que le ha eliminado pasará a ser el rey de la pista. Cuando todas las iteraciones terminen, obtendremos nuestros resultados buscados. Gráficamente, para facilitar la explicación, quedaría algo tal que así:

Siendo ‘e’ los elementos comparados, donde N es el número total de elementos:



Componentes voraces del algoritmo:

1. **Conjunto de Candidatos**: Todos los elementos de entrada del vector.

2**. Conjunto de Decisiones**: Calcular el elemento mayor

3. **Función que determina la solución del problema**: Mínimo y máximo de la lista de candidatos dados.

4. **Completable**: Sí, puesto que recorreremos toda la lista para hallar los dos valores buscados.

5. **Función Selección**: La que realiza comparaciones dos a dos para poder hallar tanto el máximo como el mínimo (de hecho, es la única función del código).

6. **Función Objetivo**: Una vez más, la función máximo y mínimo.

Ejemplos de ejecución:

```
#PROBADORES
lista = [3,4,54,5,7,2,8,6] #Datos de entrada 1
lista1 = [34,10,4,8,16,23,23,2,1,3,3,9,10,64,0,35,19,38] #Datos de entrada 2
(a,b) = minimo_y_maximo(lista)
(c,d) = minimo_y_maximo(lista1)
print(f"Resultado de la 1ª prueba => Mínimo: {a} Máximo:{b}")
print(f"Resultado de la 2ª prueba => Mínimo: {c} Máximo:{d}")
```

Resultado de la 1ª prueba => Mínimo: 2 Máximo:54

Resultado de la 2ª prueba => Mínimo: 0 Máximo:64

Código del ejercicio:

```
def minimo_y_maximo(candidatos):
    """
    list(COMPARABLE)--> COMPARABLE,COMPARABLE
    OBJ: obtener el minimo y el máximo de una lista
    """
    cand_max = []
    cand_min = []
    while (len(candidatos)>=2):
        if candidatos[0]>candidatos[1]:
            cand_max.append(candidatos[0])
            del(candidatos[0])
            cand_min.append(candidatos[1])
            del(candidatos[1])
        else:
            cand_max.append(candidatos[1])
            del(candidatos[1])
            cand_min.append(candidatos[0])
            del(candidatos[0])
    while(len(cand_max)>1):
        if (cand_max[1]>cand_max[0]):
            del(cand_max[0])
        else:
            del(cand_max[1])
    while(len(cand_min)>1):
        if (cand_min[1]<cand_min[0]):
            del(cand_min[0])
        else:
            del(cand_min[1])
    if (len(candidatos)>0):
```

```
        if candidatos[0]<cand_min[0]:
            return (candidatos[0],cand_max[0])
        elif candidatos[0]>cand_max:
            return (cand_min[0],candidatos[0])
    return (cand_min[0],cand_max[0])
```

PROBLEMA 4:

Se tiene un grafo no dirigido G = < N, A >, siendo N = {1, …, n} el conjunto de nodos y A ⊆ NxN el conjunto de aristas. Cada arista (i, j) ∈ A tiene un coste asociado cij (cij > 0 ∀i, j ∈ N; si (i, j) ∉ A puede considerarse cij = +∞). Sea M la matriz de costes del grafo G, es decir, M[i, j] = cij. (al ser el grafo no dirigido se tiene que (i, j) = (j, i) por lo que la matriz M es simétrica). Teniendo como datos la cantidad de nodos n y la matriz de costes M, se pide encontrar el árbol soporte mínimo del grafo G utilizando el algoritmo de Prim, utilizando las siguientes ideas:

- **A diferencia del algoritmo de Kruskal (que crea el árbol utilizando componentes conexas independientes que se van uniendo entre sí), el algoritmo de Prim se basa en la idea de ir construyendo un árbol cada vez más grande, empezando por un único nodo y acabando por recubrir todo el grafo.**
- **El algoritmo comienza con un árbol de un nodo, al que se le añade un segundo nodo, luego un tercero, etc, hasta tener los n nodos unidos. La forma de escoger un nodo es buscando el nodo más cercano a todo el árbol, sin que se creen ciclos.**
- **A medida que crece el tamaño del árbol la búsqueda del nodo más cercano se complica, por lo que para que el algoritmo sea eficiente (el método debe tener O(n2)) hay que crear una estructura de datos que almacene la mejor distancia de cada nodo al conjunto de nodos del árbol.**
- **Se necesitará almacenar de alguna manera la forma en que se ha creado el árbol, por ejemplo indicando a qué nodo del árbol se está uniendo el nuevo candidato seleccionado.**

Solución adoptada:

Conociendo las propiedades del algoritmo de Prim, realizamos nuestro algoritmo, primero de todo, convirtiendo el grafo dado en dos listas, una de nodos, y otra de aristas. Después, decidimos por escoger el primer nodo de nuestra lista creada para simplificar los cálculos (ya que,

según el algoritmo de Prim, cualquier nodo por el que empezar es válido). Una vez hecho esto, utilizamos una función auxiliar que nos determina la menor arista conexa dado un nodo. Con ello, la añadimos a nuestra solución, habiendo comprobado previamente que esta misma no genera ciclo en nuestro árbol solución.

Finalmente, convertimos nuestro árbol a una matriz de adyacencia, ya que es la salida que se pide en el ejercicio, para así completar la solución del problema.

Componentes voraces del algoritmo:

1. **Conjunto de Candidatos**: Las aristas que unen los nodos del grafo.

2. **Conjunto de Decisiones**: Introducir a la solución todos los nodos, usando sus aristas correspondientes, comprobando previamente que estas son conexas con las soluciones ya halladas, y que no forman ciclo con estas últimas.

3. **Función que determina la solución del problema**: función 'prim' que realiza todo lo comentado previamente.

4. **Completable**: Sí, ya que dado un grafo como parámetro de entrada, siempre hallaremos la misma solución.

5. **Función Selección**: función que halla la matriz de adyacencia que cumpla el algoritmo de Prim.

6. **Función Objetivo**: la función principal 'prim', que hace uso de la función auxiliar 'menor_arista_conexa', que esta, a su vez, se ayuda de la segunda función auxiliar 'arista_conexa'.

Ejemplos de ejecución:

```
#PROBADORES
grafo1=[[0,1,0,5],[1,0,4,5],[0,4,0,2],[5,5,2,0]]
grafo2=[[0,4,0,8],[2,0,3,6],[0,7,0,10],[23,3,9,0]]
print(prim(grafo1))
print(prim(grafo2))
```

Resultado del grafo 1 = [[0, 1, 0, 0], [1, 0, 4, 0], [0, 4, 0, 2], [0, 0, 2, 0]]

Resultado del grafo 2 = [[0, 4, 0, 0], [4, 0, 3, 0], [0, 3, 0, 6], [0, 0, 6, 0]]

Código del ejercicio:

```
from cmath import inf

def arista_conexa(arista,nodos):
    return arista[0] in nodos or arista[1] in nodos

def menor_arista_conexa(aristas,nodos):
    menor=[0,0,inf]
    for i in range(len(aristas)):
            if aristas[i][2]<menor[2] and arista_conexa(aristas[i], nodos):
                menor = aristas[i]
    return menor

def prim (grafo):
    """
    matriz_ady_grafo -> matriz_ady_grafo
    OBJ: Devolver el árbol de recubrimiento mínimo de un grafo dado mediante el
algoritmo de Prim.
    PRE: El grafo debe ser no dirigido.
    """
    #Interpretar candidatos. Convertimos el grafo a una lista de nodos y una
lista de aristas
    nodos_candidatos = []
    aristas_candidatos = []
    for i in range(len(grafo)):
        nodos_candidatos.append(i)
        for j in range(i):
            if grafo[i][j]>0:
                aristas_candidatos.append([i,j,grafo[i][j]])

    nodos_solucion = []
    aristas_solucion = []

    #Escoger el primer nodo (según Prim vale cualquiera, escogemos el primero en
la lista de nodos)
    nodos_solucion.append(nodos_candidatos[0])

    #En cuanto el árbol contenga todos los nodos, será un árbol generador del
grafo
    while nodos_solucion!=nodos_candidatos:

        #Obtener arista mínima, eliminarla de candidatos y añadirla a la solución
        arista = menor_arista_conexa(aristas_candidatos,nodos_solucion)
        aristas_candidatos.remove(arista)

        #Comprobar que la arista a añadir no genera ciclo
        if not arista[0] in nodos_solucion or not arista[1] in nodos_solucion:
```

```
            if not(arista[0] in nodos_solucion): nodos_solucion.append(arista[0])
            if not(arista[1] in nodos_solucion): nodos_solucion.append(arista[1])
            aristas_solucion.append(arista)

    #Convertir a matriz de adyacencia
    arbol = []
    for i in range(len(nodos_solucion)):
        lista_aux = []
        for i in range(len(nodos_solucion)):
            lista_aux.append(0)
        arbol.append(lista_aux)

    for arista in aristas_solucion:
        arbol[arista[0]][arista[1]]=arista[2]
        arbol[arista[1]][arista[0]]=arista[2]

    return arbol
```

PROBLEMA 6:

Shrek, Asno y Dragona llegan a los pies del altísimo castillo de Lord Farquaad para liberar a Fiona de su encierro. Como sospechaban que el puente levadizo estaría vigilado por numerosos soldados se han traído muchas escaleras, de distintas alturas, con la esperanza de que alguna de ellas les permita superar la muralla; pero ninguna escalera les sirve porque la muralla es muy alta. Shrek se da cuenta de que, si pudiese combinar todas las escaleras en una sola, conseguiría llegar exactamente a la parte de arriba y poder entrar al castillo. Afortunadamente las escaleras son de hierro, así que con la ayuda de Dragona van a "soldarlas". Dragona puede soldar dos escaleras cualesquiera con su aliento de fuego, pero tarda en calentar los extremos tantos minutos como metros suman las escaleras a soldar. Por ejemplo, en soldar dos escaleras de 6 y 8 metros tardaría 6 + 8 = 14 minutos. Si a esta escalera se le soldase después una de 7 metros, el nuevo tiempo sería 14 + 7 = 21 minutos, por lo que habrían tardado en hacer la escalera completa un total de 14 + 21 = 35 minutos. Diseñar un algoritmo eficiente que encuentre el mejor coste y manera de soldar las escaleras para que Shrek tarde lo menos posible es escalar la muralla, indicando las estructuras de datos elegidas y su forma de uso. Se puede suponer que se dispone exactamente de las escaleras necesarias para subir a la muralla (ni sobran ni faltan), es decir, que el dato del problema es la colección de medidas de las "miniescaleras" (en la estructura de datos que se elija), y que solo se busca la forma óptima de fundir las escaleras.

Solución adoptada:

La forma que hemos encontrado de hallar semejante solución es sumamente sencilla. Realizamos un par de búsquedas secuenciales cada vez que queramos sumar los mínimos valores posibles.

Componentes voraces del algoritmo:

1. **Conjunto de Candidatos**: Las longitudes de las escaleras disponibles en la lista de entrada.

2. **Conjunto de Decisiones**: Fundir juntas las dos escaleras más cortas que haya disponibles en cada momento.

3. **Función que determina la solución del problema**: Función que obtiene el tiempo de fundición de las escaleras.

4. **Completable**: Sí, ya que siempre encontraremos la forma más óptima posible de unir todas las escaleras.

5. **Función Selección**: La función que obtiene las dos escaleras menores y las funde.

6. **Función Objetivo**: Devuelve el mínimo tiempo posible para fundir las escaleras.

Ejemplos de ejecución:

```
#PROBADORES
escaleras = [2,4,6,7,13,25,1]
escaleras1 = [4,24,8,6,9,1,12,35,5]
print("Tiempo mínimo en fundir todas las escaleras: ",tiempo_min_fundicion(esca-
leras))
print("Tiempo mínimo en fundir todas las escaleras: ",tiempo_min_fundicion(esca-
leras1))
```

Tiempo mínimo en fundir todas las escaleras: 134

Tiempo mínimo en fundir todas las escaleras: 282

Código del ejercicio:

```python
def tiempo_min_fundicion(escaleras):
    """
    list -> list
    OBJ: Hallar el tiempo mínimo de fundición entre todas las escaleras.
    """
    tiempo = 0
    while len(escaleras)>1:
        menor1 = escaleras[0]
        for escalera in escaleras:
            if escalera < menor1:
                menor1 = escalera
        escaleras.remove(menor1)
        menor2 = escaleras[0]
        for escalera in escaleras:
            if escalera < menor2:
                menor2 = escalera
        escaleras.remove(menor2)
        tiempo+=menor1+menor2
        escaleras.append(menor1+menor2)
    return tiempo
```