Tarea 5
Algoritmos

Emmanuel Peto Gutiérrez
José Luis Vázquez Lázaro

6 de noviembre de 2022

Problema 2

a)

Se va a calcular el máximo número de empalmes que existen entre los intervalos en cualquier punto en el tiempo. Los pasos del algoritmo se describen de la siguiente manera:

1. Dada la entrada de intervalos A, construir una lista de pares. Para cada intervalo $(t_i, t_f) \in A$, se crean los pares $(t_i$, 'i'), $(t_f$, 'f'), donde la primera entrada del par es un número y la segunda es un caracter. Evidentemente, las letras 'i' y 'f' son de "inicial" y "final".

2. Ordenar la lista de pares respecto a la primera entrada.

3. Se crea una variable *empalme* que cuenta el número de empalmes que hay en un punto en el tiempo y tener otra variable *maxval* que guardará el número máximo de empalmes, ambas variables inicialmente en 0.

4. Recorrer la lista de pares y hacer comprobaciones sobre la segunda entrada de cada par: si es 'i', incrementar *empalme* en 1; si es 'f', decrementar en 1. Si en algún punto $empalme > maxval$, actualizar *maxval* al valor de *empalme*.

5. Retornar el valor de *maxval*.

El siguiente código en Python resuelve el problema.

```
1: def skeduling(intervalos):
2:     pares = []
3:     for elem in intervalos:
4:         pares.append((elem[0], 'i'))
5:         pares.append((elem[1], 'f'))
6:     pares.sort(key=lambda p : p[0])
7:     empalme = 0
```

```
 8:      maxval = 0
 9:      for par in pares:
10:          if par[1] == 'i':
11:              empalme += 1
12:          else:
13:              empalme -=1
14:          if empalme > maxval:
15:              maxval = empalme
16:      return maxval
```

b)

Se tiene la instancia:

[(1, 3), (2, 5), (6, 8), (9, 11), (10, 12)]

A partir de esta se crea la lista de pares:

[(1,'i'), (3, 'f'), (2, 'i'), (5, 'f'), (6, 'i'), (8, 'f'), (9, 'i'), (11, 'f'), (10, 'i'), (12, 'f')]

Luego se ordena:

[(1, 'i'), (2, 'i'), (3, 'f'), (5, 'f'), (6, 'i'), (8, 'f'), (9, 'i'), (10, 'i'), (11, 'f'), (12, 'f')]

Se hace un recorrido sobre esta lista y se cuenta el número de empalmes. Se subrayará el elemento que se está revisando en la iteración actual.

1. [(1, 'i'), (2, 'i'), (3, 'f'), (5, 'f'), (6, 'i'), (8, 'f'), (9, 'i'), (10, 'i'), (11, 'f'), (12, 'f')]
   empalme = 1
   maxval = 1

2. [(1, 'i'), (2, 'i'), (3, 'f'), (5, 'f'), (6, 'i'), (8, 'f'), (9, 'i'), (10, 'i'), (11, 'f'), (12, 'f')]
   empalme = 2
   maxval = 2

3. [(1, 'i'), (2, 'i'), (3, 'f'), (5, 'f'), (6, 'i'), (8, 'f'), (9, 'i'), (10, 'i'), (11, 'f'), (12, 'f')]
   empalme = 1
   maxval = 2

4. [(1, 'i'), (2, 'i'), (3, 'f'), (5, 'f'), (6, 'i'), (8, 'f'), (9, 'i'), (10, 'i'), (11, 'f'), (12, 'f')]
   empalme = 0
   maxval = 2

5. [(1, 'i'), (2, 'i'), (3, 'f'), (5, 'f'), (6, 'i'), (8, 'f'), (9, 'i'), (10, 'i'), (11, 'f'), (12, 'f')]
   empalme = 1
   maxval = 2

6. [(1, ‘i’), (2, ‘i’), (3, ‘f’), (5, ‘f’), (6, ‘i’), (8, ‘f’), (9, ‘i’), (10, ‘i’), (11, ‘f’), (12, ‘f’)]
   empalme = 0
   maxval = 2

7. [(1, ‘i’), (2, ‘i’), (3, ‘f’), (5, ‘f’), (6, ‘i’), (8, ‘f’), (9, ‘i’), (10, ‘i’), (11, ‘f’), (12, ‘f’)]
   empalme = 1
   maxval = 2

8. [(1, ‘i’), (2, ‘i’), (3, ‘f’), (5, ‘f’), (6, ‘i’), (8, ‘f’), (9, ‘i’), (10, ‘i’), (11, ‘f’), (12, ‘f’)]
   empalme = 2
   maxval = 2

9. [(1, ‘i’), (2, ‘i’), (3, ‘f’), (5, ‘f’), (6, ‘i’), (8, ‘f’), (9, ‘i’), (10, ‘i’), (11, ‘f’), (12, ‘f’)]
   empalme = 1
   maxval = 2

10. [(1, ‘i’), (2, ‘i’), (3, ‘f’), (5, ‘f’), (6, ‘i’), (8, ‘f’), (9, ‘i’), (10, ‘i’), (11, ‘f’), (12, ‘f’)]
    empalme = 0
    maxval = 2

Finalmente se devuelve el valor de *maxval*, que en este caso es 2.

c)

Definición. Se define la *profundidad* de un conjunto de intervalos como el máximo número de intervalos que pasa por cualquier punto en la línea de tiempo.

Lo que hace el algoritmo descrito es calcular la profundidad del conjunto de intervalos, pues lo que hace es recorrer la línea del tiempo en orden, incrementa un contador cada vez que encuentra un intervalo que inicia y decrementa el contador cada vez que encuentra un intervalo que cierra. El máximo valor de ese contador (*empalme*) será igual a la profundidad.

La profundidad es igual al mínimo número de salones necesarios para calendarizar los horarios sin que se empalmen. Esa afirmación se sigue del siguiente teorema:

Teorema. El algoritmo greedy de arriba calendariza cada intervalo en un recurso, usando el número de recursos igual a la profundidad del conjunto de intervalos. Este es el número óptimo de recursos necesitados.[1]

No se mostrará el algoritmo al que hace referencia el teorema, la parte importante del teorema es que la profundidad del conjunto de intervalos es el número óptimo de recursos (salones) necesitados. Esto demostraría la correctitud del algoritmo `skeduling` descrito arriba.

d)

[1] Este teorema está en el libro de Algorithm Design de Jon Kleinberg y Eva Tardos.

- Construir la lista de pares (las que tienen 'i' y 'f') toma tiempo $O(n)$.
- Ordenar la lista de pares toma tiempo $O(n \log n)$.
- Recorrer la lista de pares para calcular el máximo número de empalmes toma tiempo $O(n)$.

Por lo tanto, la complejidad total del algoritmo es $O(n \log n)$.

Problema 3

a)

Se propondrá un algoritmo para emparejar al i-ésimo elemento del vector bailarín con el i-ésimo elemento del vector fijo, suponiendo que se ordenan ambos. Es decir, se buscará que en el producto punto se multiplique al elemento más grande de B con el más grande de F, el segundo más grande de B con el segundo más grande de F y así sucesivamente.

Se describe el algoritmo en los siguientes pasos:

1. A partir del vector fijo, construir un vector de pares $(F[p], p)$, donde $F[p]$ es el elemento en el vector y p es su posición.
2. Ordenar el vector de pares respecto a la primera entrada.
3. Ordenar el vector bailarín.
4. Construir un vector de salida del mismo tamaño que el vector bailarín.
5. Para cada índice i de los vectores ordenados (el de pares y el bailarín): colocar en el vector de salida al elemento $B[i]$ en la posición $P[i][1]$, sabiendo que el elemento $P[i]$ es un par y $P[i][1]$ es la segunda entrada de ese par.

Se tiene el siguiente código en Python para el algoritmo:

```
 1: def bailarin(F, B):
 2:     pares = []
 3:     for i in range(len(F)):
 4:         pares.append((F[i], i))
 5:     pares.sort(key=lambda p : p[0])
 6:     B.sort()
 7:     retVal = []
 8:     for i in range(len(B)):
 9:         retVal.append(0)
10:     for i in range(len(B)):
11:         retVal[pares[i][1]] = B[i]
12:     return retVal
```

b)

Instancia: $F = [1.3, 1, 4, 2.5]$, $B = [9, 5, 3, 8]$.

Se construye la lista de pares:

[(1.3, 0), (1, 1), (4, 2), (2.5, 3)]

Luego se ordena respecto a la primera entrada:

[(1, 1), (1.3, 0), (2.5, 3), (4, 2)]

Se ordena el vector B:

[3, 5, 8, 9]

Se recorren en orden los vectores B y *pares*. Dado el vector de salida (inicialmente con todos sus valores en 0), se colocan en éste los elementos de B.

1. pares = [$\underline{(1, 1)}$, (1.3, 0), (2.5, 3), (4, 2)]
   B = [$\underline{3}$, 5, 8, 9]
   retVal = [0, $\underline{3}$, 0, 0]
2. pares = [(1, 1), $\underline{(1.3, 0)}$, (2.5, 3), (4, 2)]
   B = [3, $\underline{5}$, 8, 9]
   retVal = [$\underline{5}$, 3, 0, 0]
3. pares = [(1, 1), (1.3, 0), $\underline{(2.5, 3)}$, (4, 2)]
   B = [3, 5, $\underline{8}$, 9]
   retVal = [5, 3, 0, $\underline{8}$]
4. pares = [(1, 1), (1.3, 0), (2.5, 3), $\underline{(4, 2)}$]
   B = [3, 5, 8, $\underline{9}$]
   retVal = [5, 3, $\underline{9}$, 8]

Finalmente se devuelve el vector retVal = [5, 3, 9, 8].

c)

Lo que hace el algoritmo es acomodar los elementos de B de forma que el producto punto entre B y F sea de la siguiente forma:

$$sum_{i=0}^{n} b_i * f_i$$

donde se cumple que $b_i \leq b_{i+1}$ y $f_i \leq f_{i+1}$.

Se va a demostrar que ésta es producto punto más grande que se puede obtener con los valores de B y F.

Sean B_{ord} y F_{ord} los vectores B y F pero ordenados. El producto punto que se obtiene con el reordenamiento del algoritmo es la suma de todos los productos $B_{ord}[i] * F_{ord}[i]$ para $0 \leq i < n$. Sea B' el resultado de intercambiar exactamente dos elementos de B_{ord}, digamos, los índices j y k con $j < k$.

En el producto punto de B_{ord} con F_{ord} se multiplica b_j con f_j y b_k con f_k, mientras que en el producto punto de B' con F_{ord} se multiplica b_k con f_j y b_j con f_k. Sea $pp(A, B)$ la función que calcula el producto punto de A con B. Entonces $pp(B', F_{ord}) = pp(B_{ord}, F_{ord}) - (b_j * f_j + b_k * f_k) + (b_k * f_j + b_j * f_k))$.

Pero se tiene que $b_k * f_j + b_j * f_k \leq b_j * f_j + b_k * f_k$ y entonces $(b_k * f_j + b_j * f_k) - (b_j * f_j + b_k * f_k) \leq 0$ y así $pp(B', F_{ord}) \leq pp(B_{ord}, F_{ord})$. Con esto se tiene que, cualquier permutación de B que sea diferente a la que el algoritmo da como resultado tendrá un producto punto con F menor o igual a $pp(B_{ord}, F_{ord})$.

d)

- Construir la lista de pares a partir de F toma tiempo $O(n)$.
- Ordenar la lista de pares toma tiempo $O(n \log n)$.
- Ordenar el vector bailarín toma tiempo $O(n \log n)$.
- Construir el vector de salida y colocar los elementos de B en éste toma tiempo $O(n)$.

Por lo tanto, la complejidad total del algoritmo es $O(n \log n)$.