Trabajo Práctico

75.29 Teoría de Algoritmos I

FACULTAD
DE INGENIERIA

Universidad de Buenos Aires

Profesores:

Víctor Podberezski

1° Cuatrimestre 2020

Integrantes:

NOMBRE Y APELLIDO	PADRON
Hernán Arroyo García	91257
Ruben Jimenez	92402
Federico Rodrigez Longhi	93336
Emanuel Condo	94773

Parte 1: Problema de ausentismo.

1. Pseudocódigo

```
n = tamanio EMPLEADOS
INTERVALOS = []
EMPLEADOS = ordenar_por_SI(EMPLEADOS,'creciente')
INTERVALO = obtener_intervalo(EMPLEADOS[0])
Para k desde 1 hasta n
    SI (INTERVALO se superpone con EMPLEADOS[k])
        INTERVALO = obtener_interseccion(INTERVALO,
EMPLEADOS[k])
        SI (k == n)
            agregar INTERVALO a INTERVALOS
    SINO
        agregar INTERVALO a INTERVALOS
        INTERVALO = obtener_intervalo(EMPLEADOS[k])
Para cada INTERVALO en INTERVALOS
        tiempo = obtener_cualquier_punto(INTERVALO)
        UBICACION_EMPLEADOS = FUNCION_DTI(tiempo)
Para cada EMPLEADO en UBICACION_EMPLEADOS
        imprimir_resultado(EMPLEADO)
```

2. Tipo de algoritmo y justificación

Objetivo

Encontrar la menor cantidad de intervalos de tiempo de tal forma que todos los intervalos se superpongan con el horario de trabajo de todos los empleados.

Estrategia

Ordenar los empelados por tiempo de inicio de actividades, de tal forma que se encuentre aquellos empleados que comiencen a realizar su trabajo primero, y encontrar los intervalos de tiempo más chicos que abarquen la mayor cantidad de empleados posible.

Justificación

Se puede observar que en cada iteración el intervalo de tiempo de consulta se hace más chico o se mantiene y va abarcando a más empleados hasta donde sea posible.

Cuando el intervalo deja de intersecarse con más empleado es porque se necesita otro intervalo de consulta. Esto quiere decir que es imposible hacer la consulta en un único tiempo "t" para poder localizar a todos los empleados en su horario laboral.

Entonces, en cada iteración por cada empleado con intersección de franja horarias se disminuye en una unidad la cantidad las consultas al sistema DTI. Por lo tanto es una elección Greedy ya que vamos mejorando y llegamos a un optimo local.

3. Complejidad

```
n = tamanio EMPLEADOS
INTERVALOS = []
EMPLEADOS = ordenar_por_SI(EMPLEADOS,'creciente') |
O(nlog(n))
INTERVALO = obtener_intervalo(EMPLEADOS[0])
Para k desde 1 hasta n | O(n)
    SI (INTERVALO se superpone con EMPLEADOS[k])
        INTERVALO = obtener_interseccion(INTERVALO,
EMPLEADOS[k])
        SI (k == n)
            agregar INTERVALO a INTERVALOS
    SINO
        agregar INTERVALO a INTERVALOS
        INTERVALO = obtener_intervalo(EMPLEADOS[k])
Para cada INTERVALO en INTERVALOS | O(m)
        tiempo = obtener_cualquier_punto(INTERVALO)
        UBICACION_EMPLEADOS = FUNCION_DTI(tiempo)
Para cada EMPLEADO en UBICACION_EMPLEADOS | O(w)
        imprimir_resultado(EMPLEADO)
```

Esto da como resultado : O(nlog(n)) + O(n) + O(m) + O(w) = O(nlog(n))

4. Justificacion solución óptima

Supongamos que tenemos una solución óptima (Op). Si la solución propuesta no es óptima, entonces INTERVALOS debe ser mayor que Op.

Entonces #(INTERVALOS) > #(Op)

Esto quiere decir que INTERVALOS tiene al menos un intervalo de consulta más que la solución óptima. En otras palabras el algoritmo propuesto pudo haber realizado uno o más intersecciones (el cual genera almenos un intervalo más de tiempo). Pero eso no se puede dar porque los empleados están ordenados por inicio de actividades, por lo cual si el intervalo de tiempo deja de intersecarse con un empleado, es porque al menos uno de los empleados previamente intersecados no se interseca con el nuevo empleado, lo que genera obligatoriamente un nuevo intervalo de consulta.

Por lo tanto la cantidad de INTERVALOS de consulta es óptimo.

Parte 2: Una nueva regulación industrial.

1. Pseudocódigo del proceso "A"

```
ProcesoA()
Constructor lote (O(n) lo desestimamos porque es la lectura del
archivo)
Inicializo esValido en falso
Para todo item ∈ lote y no esValido
   Para todo item2 ∈ lote y no esValido
      Si son iguales item y item2, incrementamos contador
      Si contador es mayor a mitad de lote, activo esValido
y fin los loops
   Fin Para
Fin Para
Retornar esValido
```

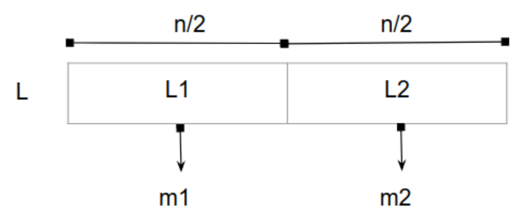
2. Pseudocódigo del proceso "B"

```
ProcesoB()
Constructor lote (O(n) lo desestimamos porque es la lectura del
archivo)
Ordeno lote (O(nlog2n))
Inicializo esValido en falso
Para todo item ∈ lote y no esValido
   Si son iguales item y item siguiente, incrementamos
contador
   Si son distintos inicializo contador en cero
   Si contador es mayor a mitad de lote, activo esValido y
fin los loops
Fin Para
Retornar esValido
```

3. Proceso "C"

Aplicando divide y venceras, buscamos particionar el problema en en sub problemas, resolverlos y finalmente resolver el problema inicial.

Si el lote "L" tiene un item mayoritario "m", entonces m tiene que ser mayoritario del sub problema L1 ó del sub problema L2. Esto es:
$\frac{|L|}{2} < |L|_m$, entonces $\frac{|L_1|}{2} < |L_1|_m$ ó $\frac{|L_2|}{2} < |L_2|_m$



Demostración: Por el absurdo. Supongamos que "m" no es mayoritario de L1, ni de L2. Esto es: $|L_1|_m \leq \frac{|L_1|}{2}$ y $|L_2|_m \leq \frac{|L_2|}{2}$

Entonces,

$\frac{|L|}{2} < |L|_m$ // Si existiera mayoritario se tendria que cumplir
$|L|_m = |L_1|_m + |L_2|_m$
$|L_1|_m + |L_2|_m \leq \frac{|L_1|}{2} + \frac{|L_2|}{2}$ // reemplazando hipotesis
$\frac{|L_1|}{2} + \frac{|L_2|}{2} = \frac{|L_1|+|L_2|}{2} = \frac{|L|}{2}$ // Absurdo, son iguales

Sabiendo esto, podemos calcular el mayoritario en L1 y L2. Si existe m1 ó m2, estos serán candidatos para ser mayoritario del lote L, se tendrá que contar las apariciones en el lote para asignarlo como mayoritario. Sino no podemos afirmar que no existe mayoritario de L.

Pseudicódigo:

```
ProcesoC()
   Constructor lote L[1...n](O(n) lo desestimamos porque es la lectura del archivo)
   MayoritarioRecursivo(L[1...n])
MayoritarioRecursivo(L[1 . . . n]):
   Si n = 1: Devolver L[1] (O(1))
   Sea m1 = MayoritarioRecursivo(L[1 . . . n/2])
   Sea m2 = MayoritarioRecursivo(L[n/2 + 1 . . . n])
   Si Apariciones(L, m1, 1, n) > n/2: Devolver m1 (O(n))
   Si Apariciones(L, m2, 1, n) > n/2: Devolver m2 (O(n))
   Devolver vacio (O(1))
Apariciones(L[1 . . . n], x, i, j):
   Inicializar contador
   Para cada k ∈ {i, . . . , j}:
      Si L[k] = x:
         Incrementar en uno contador
   Devolver contador
```

4. Complejidades temporales y espaciales

ProcesoA

Complejidad temporal $O(n^2)$, es cuadratica porque por cada item del lote se compara con todos los otros elementos
Complejidad espacial$O(1)$, es constante porque no se tiene memoria adicional a la entrada (lote).

ProcesoB

Complejidad temporal $O(nlog_2n)$, porque se ordena el lote, esto es de complejidad $O(nlog_2n)$, después se recorre todos los items ordenados contando los items de igual volumen $O(n)$
Complejidad espacial$O(n)$, es lineal porque el ordenamiento aloca un nuevo lote ordenado en memoria.

ProcesoC

Complejidad temporal $O(nlog_2n)$. Es un merge sort modificado. Se divide en mitades hasta llegar a lotes de tamaño uno. Complejidad de caso base $O(1)$, complejidad para el calculo de apariciones $O(cn)$. El tiempo de ejecucion del procesoC, es la sumatoria de todos los tiempos de los subproblemas $(l * cn)$, siendo l=altura de arbol, entonces $l = log_2n + 1$. Reemplazando, el tiempo de ejecucion quedaria que el tiempo de ejecucion es $cn.log_2n + cn$. Esto esta acotado por $O(nlog_2n)$.
Complejidad espacial$O(log_2n)$, ya que a pesar de que la division y conquista no usa memoria adicional más que la del lote, se tiene una porción de memoria no constante en el stack para la recursion y esta es de $O(log_2n)$.

4. Relacion de recurrencia

Para el procesoC, se tiene la relacion de recurrencia:

$$T(n) = 2T(\frac{n}{2}) + 2n$$

5. Teorema Maestro

$T(n) = 2T(\frac{n}{2}) + n)$, a = 2, b = 2, f(n) = 2n
Cumple caso 2 ?
$f(n) = 0(nlog_b a)$
$f(n) = 0(nlog_2 2)$
$f(n) = 0(n)$ // f(n) esta acotada superiormente y inferior n
Entonces:
$T(n) = 0(log_2 n.(n^{log_b a}))$
$T(n) = 0(log_2 n.(n^{log_2 2}))$
$T(n) = 0(nlog_2 n.)$