




Teoría de Algoritmos I 75.29

Trabajo Práctico N° 1

Grupo Rosita Forever

Apellido y Nombre	Padrón	Correo electrónico
Jamilis, Netanel David	99093	njamilis@fi.uba.ar
Del Torto, Agustín	98867	adeltorto@fi.uba.ar
Daverede, Agustín	98540	agusdaverede@yahoo.com.ar
Betz, Joaquín	104348	

Github: 

<https://github.com/netaneldj/7529-TDA/TP1>

Índice

1	Introducción	2
1.1	Objetivos	2
1.2	Resumen	2
2	Un problema de ausentismo	3
2.1	Tipo de problema	3
2.2	Algoritmo	4
2.3	Tipo de algoritmo	5
2.4	Complejidad del algoritmo	6
2.5	Solución óptima	7
3	Una nueva regulación industrial	9
3.1	Compilación y ejecución	9
3.2	Proceso A	9
3.2.1	Pseudocódigo	9
3.2.2	Complejidad	9
3.3	Proceso B	10
3.3.1	Pseudocódigo	10
3.3.2	Complejidad	10
3.4	Proceso C: División y Conquista	10
3.4.1	Pseudocódigo	11
3.4.2	Relación de recurrencia	11
3.4.3	Complejidad	12
4	Conclusión	13
4.1	Un problema de ausentismo	13
4.2	Una nueva regulación industrial	13
5	Bibliografía y Referencias	14

1 Introducción

1.1 Objetivos

Los objetivos del presente trabajo son los siguientes:

- Aplicar los conceptos aprendidos en clase en un problema Greedy
- Aplicar los conceptos aprendidos en clase en un problema de División y Conquista
- Preparar un informe técnico

En las siguientes secciones se verá plasmado el cumplimiento de los objetivos.

1.2 Resumen

El presente trabajo se centrará en la presentación y consecuente resolución de los problemas planteados.

En primer lugar, se tratará el problema de ausentismo, perteneciente a la familia de problemas Greedy. Se analizará su solución y se planteará explicando el algoritmo empleado paso a paso. Luego, se repetirá el mismo análisis para el problema de una nueva regulación industrial, donde se aplicará la estrategia de división y conquista.

En último lugar, se analizarán los resultados obtenidos, donde se tendrán en cuenta los conocimientos adquiridos en el desarrollo del trabajo y la bibliografía consultada para la realización del presente informe.

2 Un problema de ausentismo

Una empresa de tercerización laboral nos convoca para que le ayudemos con un problema de ausentismo laboral. Tiene un conjunto de n empleados que realizan tareas en diferentes puntos de la ciudad. El turno de cada empleado i comienza en $T_i(i)$ y termina en $T_f(i)$ y durante todo ese lapso tiene que estar en la ubicación establecida. La dirección de la empresa sospecha que algunos de sus empleados suelen faltar sin aviso. Para verificarlo contrataron a la empresa “Dystopian Technologies Inc.” (DTI). Esta empresa implanta un microchip con un código único en cada empleado. Mediante rastreo satelital pueden conocer dónde se encuentra cada chip implantado en cualquier momento. Además posee el cronograma completo de las tareas.

DTI brinda un sistema que mediante una consulta (encendido / apagado) nos devolverá cuáles empleados aún no controlados y en horario de trabajo se encuentran en su sitio y cuáles no.

2.1 Tipo de problema

El problema presentado pertenece a la familia de problemas *Interval Scheduling*. El problema más famoso de esa familia es el del organizador de materias.

En particular nuestro problema es una variante del *Interval Partitioning Problem*, en donde en vez de ubicar la mayor cantidad de materias en la mínima cantidad de aulas debemos ubicar la mayor cantidad de turnos en la mínima cantidad de consultas.

2.2 Algoritmo

El siguiente pseudocódigo se podrá utilizar para controlar que todos los empleados estén en sus puestos de trabajo en algún momento de su turno.

```
Declaro la clase Consulta con dos atributos, inicio y fin.
Creo una pila de para apilar objetos Consulta.
Creo una lista vacía donde almacenare a los empleados presentes
Creo una lista vacía donde almacenare a los empleados ausentes
Ordeno todos los turnos por su tiempo de comienzo.
Llamare T(1), T(2), . . . , T(n) a los turnos en orden.

Creo la consulta(1) con inicio Ti(1) y fin Tf(1).
Apilo la consulta(1).

Para j = 1, 2, 3, . . . , n:
    Si el turno j se superpone la consulta del tope de la pila:
        Desapilo la consulta.
        Actualizo inicio y de fin con su intersección con el turno j.
        Apilo la consulta.
    Sino:
        Creo una nueva consulta con el inicio y fin del turno j.
        Apilo la consulta.
    Fin si.
Fin para.

Mientras la pila no este vacia:
    Desapilo una consulta.
    Llamo a DTI en un instante aleatorio entre el inicio y el fin de la consulta.
    Agrego presentes a la lista de presentes.
    Agrego ausentes a la lista de ausentes.
Fin Mientras.
```

2.3 Tipo de algoritmo

El algoritmo de la sección anterior es de tipo *Greedy*.

Esto se debe a que su comportamiento sigue una heurística consistente en elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima. Esta es elegir el mejor tiempo posible para un turno dado abarcando la mayor cantidad de turnos de otros empleados en una consulta.

Además una vez que nuestro algoritmo toma una decisión jamás vuelve a ella para revisarla o modificarla, simplemente asume que es óptimo lo que hizo previamente y sigue iterando hasta solucionar el problema.

2.4 Complejidad del algoritmo

Para ver la complejidad de nuestro algoritmo vamos a analizar en detalle el pseudocódigo:

```

Declaro la clase Consulta con dos atributos, inicio y fin.
Creo una pila de para apilar objetos Consulta.
Creo una lista vacía donde almacenare a los empleados presentes
Creo una lista vacía donde almacenare a los empleados ausentes
Ordeno todos los turnos por su tiempo de comienzo.
Llamare  $T(1), T(2), \dots, T(n)$  a los turnos en orden.

Creo la consulta(1) con inicio  $T_i(1)$  y fin  $T_f(1)$ .
Apilo la consulta(1).

Para  $j = 1, 2, 3, \dots, n$ :
    Si el turno  $j$  se superpone la consulta del tope de la pila:
        Desapilo la consulta.
        Actualizo inicio y de fin con su intersección con el turno  $j$ .
        Apilo la Consulta.
    Sino:
        Creo una nueva consulta con el inicio y fin del turno  $j$ .
        Apilo la consulta.
    Fin si.
Fin para.

Mientras la pila no este vacia:
    Desapilo una consulta.
    Llamo a DTI en un instante aleatorio entre el inicio y el fin de la consulta.
    Agrego los presentes a la lista de presentes.
    Agrego los ausentes a la lista de ausentes
Fin Mientras.

```

Diagrama de complejidad:

- Declaro la clase Consulta con dos atributos, inicio y fin. $O(1)$
- Creo una pila de para apilar objetos Consulta. $O(1)$
- Creo una lista vacía donde almacenare a los empleados presentes $O(1)$
- Creo una lista vacía donde almacenare a los empleados ausentes $O(1)$
- Ordeno todos los turnos por su tiempo de comienzo. $O(n \log n)$
- Llamare $T(1), T(2), \dots, T(n)$ a los turnos en orden. $O(1)$
- Creo la consulta(1) con inicio $T_i(1)$ y fin $T_f(1)$. $O(1)$
- Apilo la consulta(1). $O(1)$
- Para $j = 1, 2, 3, \dots, n$: $O(n)$
 - Si el turno j se superpone la consulta del tope de la pila: $O(1)$
 - Desapilo la consulta. $O(1)$
 - Actualizo inicio y de fin con su intersección con el turno j . $O(1)$
 - Apilo la Consulta. $O(1)$
 - Sino: $O(1)$
 - Creo una nueva consulta con el inicio y fin del turno j . $O(1)$
 - Apilo la consulta. $O(1)$
 - Fin si. $O(1)$
- Fin para. $O(n)$
- Mientras la pila no este vacia: $O(n)$
 - Desapilo una consulta. $O(1)$
 - Llamo a DTI en un instante aleatorio entre el inicio y el fin de la consulta. $O(1)$
 - Agrego los presentes a la lista de presentes. $O(1)$
 - Agrego los ausentes a la lista de ausentes $O(1)$
- Fin Mientras. $O(n)$

Por lo tanto el algoritmo sera $\mathcal{O}(n + n \log n) = \mathcal{O}(n \log n)$ siendo n la cantidad de trabajadores de la empresa.

2.5 Solución óptima

Vamos a definir que nuestro algoritmo es óptimo si nos devuelve la asistencia de los empleados efectuando la cantidad mínima de consultas posible a DTI. Esto es que no exista otro algoritmo que devuelva un resultado correcto efectuando una menor cantidad de consultas a DTI.

Nuestro algoritmo *greedy* elige en cada paso el mejor tiempo posible para un turno dado abarcando la mayor cantidad de turnos de otros empleados en una consulta, llegando así al conjunto de consultas óptimo global.

A continuación **vamos a demostrar que nuestra solución *greedy* es óptima siguiendo el funcionamiento del algoritmo:**

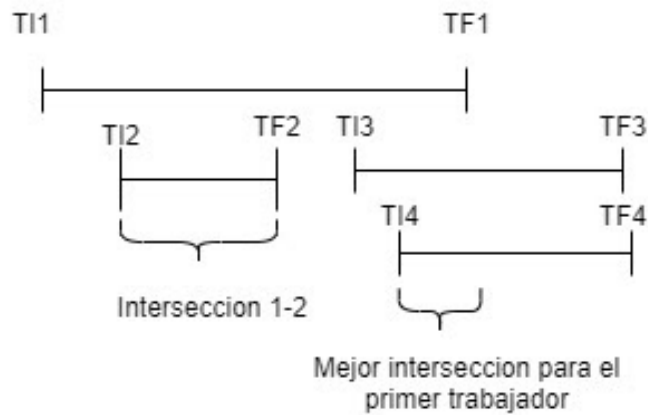
Comenzamos tomando el primer turno que consta de un tiempo inicial $t_i(1)$ y un tiempo final $t_f(1)$. Sabemos que este empleado debe figurar en el reporte de asistencia, por lo que vamos a buscar el mejor tiempo t entre $t_i(1)$ y $t_f(1)$.

Luego buscamos definir un t que abarque a tantos turnos como sea posible. En este caso el ordenamiento inicial nos garantiza que si algún turno coincide en un algún instante con nuestro primer turno este será por lo menos el segundo. Para afirmar esto nos basamos en el siguiente principio: *"si un empleado no se cruzó con ningún otro, entonces no se cruzó con el que llegó inmediatamente después de él"*.

- Si no coinciden hacemos la consulta en cualquier instante dentro del primer turno y creamos una nueva consulta para el segundo. El hecho de que abarque a un solo turno no es un inconveniente ya que dicho turno se encuentra aislado del resto. La única forma de registrarlo es con una consulta particular.
- Si coinciden reducimos nuestro rango de la consulta al de la intersección de los dos turnos.

Luego preguntamos por la intersección con el siguiente y repetimos el mismo razonamiento hasta terminar ubicar a todos los empleados en alguna consulta.

Ahora bien, este escenario presenta la problemática de que quizás tenga elementos coincidentes con mi primer turno pero no con el rango de intersección, y por lo tanto puede haber un t que abarque a más turnos que este. Sin embargo, este caso no nos quita la optimalidad del algoritmo; puesto que si bien no es el t óptimo para nuestro primer turno, sí lo será para alguno de los empleados con los que se interceptó. Por lo tanto nuestro t hallado es la mejor consulta posible para algún turno y, siguiendo el postulado de que igualmente dicho trabajador debía estar consultado al final del algoritmo, t será un instante correcto para consultar.



Vemos que el trabajador 2 solo tiene a un trabajador dentro de su franja horario por lo que si bien para el primer trabajador podria ser mejor ignorar dicha interseccion, para el trabajador 2 va a ser optimo un tiempo alli y por lo tanto sigue siendo un tiempo valido como optimo local

Por lo tanto al buscar siempre en cada paso un t óptimo con este algoritmo, la cantidad de consultas sera mínima.

3 Una nueva regulación industrial

A raíz de una nueva regulación industrial un fabricante debe rotular cada lote que produce según un valor numérico que lo caracteriza. Cada lote está conformado por “n” piezas. A cada una de ellas se le realiza una medición de volumen. La regulación considera que el lote es válido si más de la mitad de las piezas tienen el mismo volumen. En ese caso el rótulo deberá ser ese valor. De lo contrario el lote se descarta.

3.1 Compilación y ejecución

Para poder compilar el archivo part2.cpp ejecutar en consola de linux: `g++ -o tp part2.cpp`
Para poder ejecutarlo, en el directorio del ejecutable generado: `./tp PROCESO lotName`
Donde PROCESO puede ser 'A', 'B' o 'C', sin comillas y lotName es el nombre del archivo del lote, que deberá estar en el mismo directorio.

3.2 Proceso A

Consiste en para cada pieza del lote contar cuántas de las restantes tienen el mismo volumen. Si alguna de las piezas corresponde al “elemento mayoritario”, se rotula el lote. De lo contrario se lo rechaza.

3.2.1 Pseudocódigo

```
Llamar s a la cantidad de piezas del lote
Inicializar un contador en q = 1, como cantidad de
elementos con el mismo volumen
  Para cada pieza P_i del lote L:
    Para cada pieza P_j siguiente a P_i del lote L:
      Si el volumen v de las piezas P_i y P_j es el mismo:
        incrementar q en 1.
    fin para
  Si q > s/2, finalizar la ejecución y etiquetar el lote con v.
  Si no, reiniciar q a 1 y avanzar en la iteración
  con la siguiente pieza.
fin para
Rechazar el lote.
```

3.2.2 Complejidad

Esta solución tiene una complejidad temporal de $\mathcal{O}(n^2)$, siendo n la cantidad de piezas en el lote, pues hay dos ciclos anidados que se ejecutarían en función de esa variable. En el peor caso (no hay volumen mayoritario), para cada elemento i del lote se hacen $(n - i)$ consultas. $n * (n - i) = n^2 - i * n \Rightarrow \mathcal{O}(n^2)$. En el mejor caso, por otro lado, el ciclo

se ejecutará una sola vez para un solo elemento, resultando $\mathcal{O}(n)$. Complejidad espacial: $\mathcal{O}(1)$ pues no se utilizan estructuras auxiliares.

3.3 Proceso B

Consiste en ordenar las piezas por volumen y con ello luego reducir el tiempo de búsqueda del elemento mayoritario.

3.3.1 Pseudocódigo

```
# pre: recibe un lote ordenado (lista de numeros),
Iniciar un contador q = 0
Llamar k a la posición del elemento del medio del lote
Mientras el elemento del lote P_i sea igual al elemento de k:
    incrementar q
    incrementar i
fin mientras
Mientras el elemento del lote P_i sea igual al elemento de k - 1:
    incrementar q
    decrementar i
Si q es mayor que la mitad del tamaño del lote,
    aprobar lote e indicar v = v[elemento de posición k]
Sino, rechazar lote
```

3.3.2 Complejidad

La complejidad temporal de este proceso depende del algoritmo de ordenamiento utilizado. El algoritmo en su peor caso es $\mathcal{O}(n)$. Si se utiliza mergesort o quicksort, que es $\mathcal{O}(n \log n)$: $\mathcal{O}(n) + \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$. Complejidad espacial: $\mathcal{O}(1)$ pues no se utilizan estructuras auxiliares.

3.4 Proceso C: División y Conquista

Se utiliza una versión recursiva del algoritmo de votación del elemento mayoritario de Boyer-Moore para lograr una solución superadora a los procesos anteriormente propuestos, teniendo que consultar únicamente dos veces cada pieza.

En el algoritmo original, cada elemento "tiene un voto". Se postula el primer elemento como candidato a elemento mayoritario. Si el elemento "encuestado" es igual al candidato a elemento mayoritario, suma su voto. Si no, le resta un voto. Si la cantidad de votos llega a 0, el siguiente encuestado se postula como candidato y se continúa con la votación. Al finalizar, se verifica que el candidato final sea efectivamente el elemento mayoritario contando la cantidad de apariciones. Si no lo es, no hay elemento mayoritario.

Sea L una lista de elementos y M su elemento mayoritario (si existe). Si L está vacía, se devuelve el elemento de desempate D . Sino, se particiona L en pares y se incluye en una lista P sólo aquellos elementos que aparezcan dos veces en un mismo par (puesto que si la pareja no es idéntica, sus votos se cancelarían entre sí), entonces el elemento mayoritario en P es el mismo que en L . Como L puede no ser par, se utiliza en esos caso un elemento de desempate D igual al último elemento de L . Si L es par, D es nulo. Este D servirá para como un voto parcial para desempatar candidatos a M , dado el caso en alguno de los sub-problemas.

3.4.1 Pseudocódigo

```

ElementoMayoritario(lista, desempate=Ninguno):
    Si la lista está vacía:
        Retornar el elemento de desempate.
    Crear una lista vacía llamada pares.
    Si la lista tiene cantidad impar de elementos:
        Definir como elemento de desempate al último elemento de la lista.
    Para cada par de elementos en la lista:
        Si los dos elementos del par son idénticos:
            Añadir uno de los elementos a la lista pares
    Retornar ElementoMayoritario(pares, desempate)

ProcesoC():
    # Analiza la validez del lote e imprime la etiqueta del mismo si es
    # válido o "Rechazado" si no lo es
    R = ElementoMayoritario(lote)
    Si R es Ninguno, rechazar.
    Contar la cantidad de apariciones de R en el lote.
    Si la cantidad de apariciones es mayor a la mitad del tamaño del lote:
        imprimir R en pantalla
    Sino, rechazar.

```

3.4.2 Relación de recurrencia

En cada ejecución, se divide el problema en un subproblema $\Rightarrow q = a = 1$, y la cantidad de elementos en la lista es como máximo $n/2$ pues contiene uno o ningún número por cada par de elementos $\Rightarrow b = 2$.

Formar los pares iterando sobre la lista de elementos es $\mathcal{O}(n)$, al igual que el contar después de la recursión la cantidad de apariciones de los candidatos $\Rightarrow f(n) = 2n = \mathcal{O}(n)$.

$$T(n) = T(n/2) + 2n$$

Aplicando el caso 3 del teorema maestro:

$$\text{Si } f(n) = \Omega(n^{\log_b a + e}), e > 0 \Rightarrow T(n) = \Theta(f(n))$$

$$2n \stackrel{?}{=} \Omega(n^{0+e}), \text{ tomo } e = 0.1$$

$$2n = \Omega(n^{0.1})$$

y para la condición de regularidad:

$$\exists c < 1 \text{ para } n \text{ grande} / a * f(n/b) \leq c * f(n)$$

$$1 * 2n/2 \leq 2c * n$$

$$n/(2c) \leq n \quad \forall c \in (0, 1)$$

$$\therefore T = \mathcal{O}(n)$$

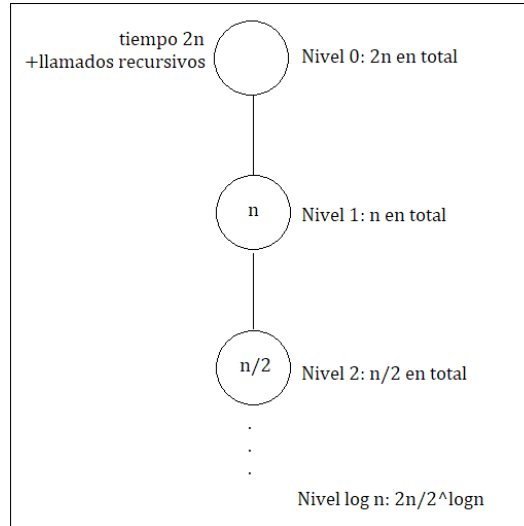


Figure 1: Desenrollado de la recurrencia para el proceso C.

Como se puede ver en la Figura 1, con cada instancia de recursividad el trabajo a realizar en cada nivel va disminuyendo. En un nivel i , habrá una sola instancia de tamaño $2n/2^i$. Sumando los $\log 2n$ niveles de la recursión, obtenemos la expresión:

$$T(n) \leq \sum_{i=0}^{\log_2 n - 1} \frac{2*n}{2^i} = 4n$$

$$\therefore T(n) = \mathcal{O}(n)$$

3.4.3 Complejidad

Complejidad temporal: $\mathcal{O}(n)$ por lo desarrollado en la sección anterior.

Complejidad espacial: $\mathcal{O}(\log n)$ por el uso de la pila de memoria. Se realizan como máximo $\log_2 n$ llamados recursivos.

4 Conclusión

Habiendo culminado el trabajo podemos efectuar las siguientes conclusiones:

4.1 Un problema de ausentismo

- El problema se simplifica al ordenar a los empleados por el horario de comienzo de su turno de forma ascendente.
- Si algún empleado se cruzó con algún otro tuvo que haber sido después de que llegó y antes se vaya.
- Como la complejidad de nuestro algoritmo es acotada por la complejidad de ordenar los elementos, si estos cumplieren requisitos para poder realizar un ordenamiento no comparativo, podría mejorar el tiempo de ejecución de nuestro programa.

4.2 Una nueva regulación industrial

- Es posible identificar el elemento mayoritario de un conjunto consultando cada elemento del mismo unas pocas veces.
- No es necesario ordenar el conjunto para solucionar el problema.
- El algoritmo encontrado escala linealmente con la cantidad de elementos del conjunto –i.e., es $\mathcal{O}(n)$ –.
- El algoritmo es simple y poderoso: si existe un elemento mayoritario, lo va a hallar.
- Si no hay un elemento mayoritario en el conjunto, el algoritmo puede obtener un candidato –a elemento mayoritario– incorrecto.
- Por esto, si con el algoritmo se obtiene un candidato, hay que recorrer el conjunto una última vez para corroborar que ese elemento sea efectivamente el que ocupa más de la mitad del conjunto. Esto no afecta el orden temporal del algoritmo.

5 Bibliografía y Referencias

- J. Kleinberg, E. Tardos, Algorithm Design
- T. Cormen, C. Leiserson, R. Rivest, C. Stein, Introduction to Algorithms
- Dr. Andrew Harrington, "Comp 363: Algorithms - Spring 2019" - Universidad Loyola Chicago.