

Trabajo Práctico Nro 1: Algoritmos Greedy y División y conquista

Nestor Huallpa, *Padrón Nro. 88614*
huallpa.nestor@gmail.com

[illegible]

1º Entrega: xx/xx/xxx

1do. Cuatrimestre de 2020
75.29/95.06 Teoría de Algoritmos I
Facultad de Ingeniería, Universidad de Buenos Aires

Contents

1	Introducción	4
2	Un problema de ausentismo	4
2.1	Descripción del problema	4
2.2	Hipótesis	4
2.3	Descripción del algoritmo	5
2.4	Pseudocódigo del algoritmo	6
2.5	Ejemplo de ejecución	6
2.6	Análisis del algoritmo	7
3	Una nueva regulación industrial	9
3.1	Descripción del problema	9
3.2	Proceso A	10
3.2.1	Descripción	10
3.2.2	Pseudocódigo	10
3.2.3	Análisis del algoritmo	11
3.3	Proceso B	13
3.3.1	Descripción	13
3.3.2	Pseudocódigo	13
3.3.3	Análisis del algoritmo	13
3.4	Proceso C	13
3.4.1	Descripción	13

3.4.2	Pseudocódigo	13
3.4.3	Análisis del algoritmo	13
3.5	Conclusiones	13

1 Introducción

En el presente trabajo plantearemos dos soluciones mediante algoritmos para al problema de ausentismo de una empresa y sobre una nueva regulación industrial.

2 Un problema de ausentismo

2.1 Descripción del problema

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.

2.2 Hipótesis

- Los tiempos informados son enteros de 0 en adelante.
- DTI les cobra por cada encendido / apagado.
- Cada encendido / apagado es casi instantáneo y se lo programa para algún valor de t entero.
- Cada encendido / apagado (y su consecuente rastreo) es $O(1)$.
- El empleado una vez en su puesto no se retira hasta concluir su turno.

2.3 Descripción del algoritmo

Tenemos un conjunto de empleados $\{1, 2, \dots, n\}$; el empleado i^{th} corresponde a un intervalo de tiempo que comienza al instante $s(i)$ y finaliza de trabajar al instante $f(i)$. Diremos que un subconjunto de empleados es compatible si no hay dos de ellos que al mismo tiempo se superpongan y nuestro objetivo es encontrar un subconjunto compatible tan grande como sea posible. El tiempo $f(i)$ de los empleados del conjunto serán los tiempos que deben encender el sensor de DTI.

La idea básica para el algoritmo greedy es seleccionar el primer empleado i_1 , una vez seleccionado, descartamos del resto de los empleados a aquellos que tienen una intersección con el seleccionado i_1 . Luego seleccionamos el empleado i_2 y volvemos a descartar todos los empleados que intesequen con el empleado i_2 . Continuamos de esta manera hasta que nos quedamos sin empleados para evaluar.

De esta forma nos iremos quedando con los empleados que terminan primero, o sea el empleado que tenga menor $f(i)$ para poder tomar el valor del instante $f(i)$ como el tiempo donde se debe encender el sensor para abarcar la mayor cantidad de empleados por cada encendido del sensor. El resultado será **óptimo** si obtenemos abarcamos a todos los empleados con la menor cantidad de encendidos del sensor de DTI. Para esta solución nos basamos en el problema de planificación de intervalos.

El algoritmo funciona de la siguiente manera:

1. Ordena la lista de empleados de menor a mayor según el valor t_f .
2. Selecciona el t_f del primer empleado de la lista t_{fx} .
3. Elimina al empleado seleccionado y todos los demás empleados que cumplan que $t_i \leq t_{fx} \leq t_f$, lo cual puede simplificarse y pedir que simplemente $t_i \leq t_{fx}$, puesto que ya se encuentra ordenado por t_f de forma ascendente.
4. Repite los pasos 2 y 3 hasta que ya no queden empleados en la lista.
5. Los t_{fx} seleccionados, son los tiempos en los cuales se debe realizar la consulta y encender el módulo DTI.

2.4 Pseudocódigo del algoritmo

```
1
2
3 lista[int] obtenerTiemposDeEncendido(lista[empleado] listaEmpleados
  )
4
5     OrdenarDeMenorAMayorPorTiempoFinal(listaEmpleados);
6
7     int Tprendido, i;
8
9     Mientras listaEmpleados != vacio
10
11         Tprendido = listaEmpleados(0).tf;
12
13         listaTiemposEncendidos.agregar(Tprendido);
14
15         listaEmpleados.remove(0);
16
17         i = 0;
18
19         Mientras listaEmpleados != fin
20
21             Si (Tprendido >= listaEmpleados(i).ti);
22
23                 listaEmpleados.remove(i);
24
25             Sino
26                 i++;
27
28             FinSi
29
30         FinMientras
31
32     Finmientras
33
34 Devolver listaTiemposEncendidos;
```

Listing 1: Algoritmo de greedy para Interval Scheduling

2.5 Ejemplo de ejecución

Solo se detallan los t_i y t_f de cada empleado. En las ejecuciones del bucle, se marcan previamente en rojo los empleados que serán eliminados de la lista.

Preparación:

listaEmpleados = {(1,4) - (2,4) - (2,5) - (3,5) - (3,6) - (1,6) - (2,7) - (5,8) - (2,8) - (6,8) - (1,10) - (5,10) - (7,10) - (7,11) - (8,11) - (10,12) - (11,12) - (12,14)}

listaTiemposEncendidos = {}

Primera ejecución del bucle:

listaTiemposEncendidos = {4}

listaEmpleados = {(1,4) - (2,4) - (2,5) - (3,5) - (3,6) - (1,6) - (2,7) - (5,8) - (2,8) - (6,8) - (1,10) - (5,10) - (7,10) - (7,11) - (8,11) - (10,12) - (11,12) - (12,14)}

listaEmpleados = {(5,8) - (6,8) - (5,10) - (7,10) - (7,11) - (8,11) - (10,12) - (11,12) - (12,14)}

Segunda ejecución del bucle:

listaTiemposEncendidos = {4 - 8}

listaEmpleados = {(5,8) - (6,8) - (5,10) - (7,10) - (7,11) - (8,11) - (10,12) - (11,12) - (12,14)}

listaEmpleados = {(10,12) - (11,12) - (12,14)}

Tercera ejecución del bucle:

listaTiemposEncendidos = {4 - 8 - 12}

listaEmpleados = {(10,12) - (11,12) - (12,14)}

listaEmpleados = {}

Resultado:

listaTiemposEncendidos = 4 - 8 - 12

2.6 Análisis del algoritmo

Necesitamos demostrar que la solución es óptima. Para esto, vamos a necesitar unas definiciones:

- Definimos R el conjunto de empleados que no fueron ni seleccionados y ni

descartados.

- Definimos E como el conjunto de empleados cuyos intervalos son compatibles.
- Definimos O como el conjunto de empleados cuyos intervalos de trabajo es óptimo.

Luego vamos a mostrar que $|E| = |O|$, o sea que el conjunto E tiene la misma cantidad de intervalos que O y por lo tanto E es una solución óptima.

Para la prueba introduciremos la siguiente notación:

- Dado $\{i_1, \dots, i_k\}$ el conjunto de empleados en E en orden que fueron agregados a E . Notar que $|E| = k$.
- Dado $\{j_1, \dots, j_m\}$ el conjunto de tiempos en O ordenados de izquierda a derecha. Notar que $|O| = m$.

El objetivo es mostrar que $k = m$. La manera en que el algoritmo de greedy se mantiene adelante (stays ahead) es que por cada uno de los intervalos de los empleados, finalice tan pronto como lo haga el correspondiente intervalo en O .

(1.1) Para todos los índices $r < k$ tenemos que $f(i_r) \leq f(j_r)$

Demostración: Probaremos la sentencia anterior mediante el método inductivo. Para $r = 1$ la sentencia anterior es cierta, el algoritmo empieza seleccionando el empleado i_1 con el menor tiempo de finalización.

Para el caso inductivo, o sea $r > 1$ asumiremos como nuestra hipótesis inductiva que la sentencia es verdadera para $r - 1$, y queremos probar que es también lo es para r . La hipótesis inductiva nos dice que asumamos verdadero que $f(i_{r-1}) \leq f(j_{r-1})$. Queremos demostrar que $f(i_r) \leq f(j_r)$.

Dado que O consiste en *intervalos compatibles*, sabemos que $f(j_{r-1}) \leq s(j_r)$. Combinando esto último con la hipótesis inductiva $f(i_{r-1}) \leq f(j_{r-1})$, obtenemos $f(i_{r-1}) \leq s(j_r)$. Así el intervalo j_r está en conjunto R de los intervalos disponibles al mismo tiempo cuando el algoritmo de greedy selecciona i_r . El

algoritmo de greedy selecciona el empleado cuyo intervalo tiene el *tiempo final mas chico* (i_r); y dado que intervalo j_r es uno de estos intervalos, tenemos que $f(i_r) \leq f(j_r)$, completando así el paso inductivo. ■

De esta forma demostramos que nuestro algoritmo se mantiene adelante del conjunto optimo O . Ahora veremos porque esto implica optimalidad del conjunto E de algoritmo de greedy.

**(1.2) El algoritmo de greedy retorna un conjunto E óptimo.
El cual usaremos para extender los tiempos para encender el sensor**

Demostración: Para demostrarlo utilizaremos la contradicción. Si E no es optimo, entonces el conjunto O debe tener mas intervalos, o sea que tenemos $m > k$ y aplicando 1.1, cuando $r=k$, obtenemos que $f(i_k) \leq f(j_k)$. Dado que $m > k$, existe un empleado j_{k+1} en O . Este empleado empieza despues que el empleado j_k termina y por consiguiente despues de que el empleado i_k termine. Entonces, despues de eliminar todos los empleados que no son compatibles con los empleados i_1, \dots, i_k , el conjunto de posibles empleados R aún contiene el empleado j_{k+1} . Pero el algoritmo de greedy se detiene con el empleado i_k y este supuestamente se detiene porque R esta vacio, lo cual es una contradicción. ■

Complejidad de algoritmo e implementación: Si la lista no esta ordenada podemos utilizar un metodo de ordeamiento del orden $O(n \log(n))$ como el heapsort. La complejidad total es $O(n)$ si la lista ya viene ordenada.

3 Una nueva regulación industrial

3.1 Descripción del problema

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.

3.2 Proceso A

3.2.1 Descripción

Actualmente cuentan con el proceso “A” que consiste en para cada pieza del lote contar cuantas de las restantes tienen el mismo volumen. Si alguna de las piezas corresponde al “elemento mayoritario”, lo rotula. De lo contrario lo rechaza.

3.2.2 Pseudocódigo

```
1  int elementoMayoritario(list[pieza] listaPiezas)
2  int totalPiezas = listaPiezas.contar();
3  int volumenPieza, i, piezaActual, cantidadVolumen;
4  piezaActual = 0;
5
6  Mientras piezaActual < totalPiezas
7      volumenPieza= listaPiezas(piezaActual).volumen;
8      cantidadVolumen = 1;
9      i = 0;
10
11     Mientras listaPiezas != fin
12
13         Si(piezaActual != i && listaPiezas(i).volumen == volumenPieza
14         )
15             cantidadVolumen ++;
16             i++;
17
18     Fin Mientras
```

```

19     Si(cantidadVolumen >techo(totalPiezas/2))
20     Retornar volumenPieza;
21     Sino
22         piezaActual++;
23     FinSi
24
25     Fin Mientras
26
27 Retornar 0;

```

Listing 2: Algoritmo del proceso A

3.2.3 Analisis del algoritmo

Consideramos la unidad (1) de complejidad temporal como el tiempo de ejecución de una operación elemental (suma, asignación, etc). Y por otro lado, consideramos la unidad (1) de complejidad espacial como un byte. Y por último n el tamaño de la lista de piezas que guarda todas las piezas.

Realizamos el siguiente analisis detallado:

- Previo a primer mientras
 - +3 Declaración, inicialización de totalPiezas y una obtención.
 - +4 Declaración de volumenPieza, i, piezaActual, cantidadVolumen.
 - +1 asignación a piezaActual.
- dentro de primer mientras (éste recorre a lo sumo todas las piezas, el total de lo que se obtenga en costo interno será multiplicado por n).
 - +1 chequeo de condición de iteración
 - +2 obtención de volumen y asignación a volumenPieza.
 - +1 asignación a cantidadVolumen.
 - +1 asignación a i
- Dentro de segundo mientras (también dentro de primer mientras, como también puede que recorra la listaPiezas por completo, añade un $*n$ a la complejidad final).
 - +1 chequeo de condición de iteración
 - +4 dos comparaciones, una obtención y un and (condición si).

- +2 caso entra a si, suma y asignación a cantidadVolumen.
- +2 suma y asignación a i
- Termina segundo mientras, sigue primer mientras
 - +2 costo de condición si, comparación y techo.
 - +1 caso entra al si, retorno.
 - +2 caso no entra al si, suma y asignación.
- Termina primer mientras
 - +1 retorna, si el retorno no se realizó dentro del primer mientras.

Definimos $T(n)$, función de complejidad temporal:

$$T(n) = 8 + n * (5 + ((n - 1) * 9 + n * 7) + 2 + 2 + 1) \quad (1)$$

$$T(n) = 8 + n * (10 + (9n - 9 + 7n)) = 8 + n * (16n + 1) \quad (2)$$

$$T(n) = 16n * n + n + 8 \quad (3)$$

La asignación de costo en la definición previa de $T(n)$ se define según el peor caso posible. Este es que se entre a la condición si del segundo mientras, la mayor cantidad de veces posible. Este peor caso es que n sea impar y halla dos subconjuntos de tamaño $n/2$ (con división entera) cada uno con igual volumen interno y un elemento m , no incluido en alguno de estos subconjuntos. Esto provoca que se ingrese $n - 1$ veces al sí del segundo mientras (una vez por cada elemento de los subconjutos), además de n no ingresos a dicho si (uno por cada elemento que encuentra al menos a uno sin su mismo volumen, todos lo hacen).

Como resultado, $T(n)$ es n^2 .

3.3 Proceso B

3.3.1 Descripción

En este proceso se tiene en cuenta que se debe ordenar las piezas por volumen y con ello luego reducir el tiempo de búsqueda del elemento mayoritario.

3.3.2 Pseudocódigo

```
1
2 int elementoMayoritario(list[pieza] listaPiezas)
3
4 Ordenar(listaPiezas)
5 int totalPiezas = listaPiezas.contar()
6 int volumenPieza, i, cantidadVolumen
7 bool mismoVolumen
8 i = 0
9
10 Mientras i < totalPiezas
11     volumenPieza= listaPiezas(i).volumen
12     cantidadVolumen = 1
13     mismoVolumen = TRUE
14     i++
15
16     Mientras listaEmpleados != fin && mismoVolumen
17         Si(listaEmpleados(i).volumen == volumenPieza)
18             cantidadVolumen ++
19             i++
20         Sino
21             Si(cantidadVolumen > techo(totalPiezas/2))
22                 Retornar volumenPieza
23             Sino
24                 mismoVolumen = FALSE
25             FinSi
26         FinSi
27     FinMientras
28 FinMientras
29
30 Retornar 0
```

Listing 3: Algoritmo del proceso A

3.3.3 Analisis del algoritmo

3.4 Proceso C

3.4.1 Descripción

3.4.2 Pseudocódigo

3.4.3 Analisis del algoritmo

3.5 Conclusiones