Trabajo Práctico I

29 de agosto de 2015 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Cadaval, Matias	345/14	`matias.cadaval@gmail.com`
Campos Paso, María Candelaria	774/11	`cande.cp@gmail.com`
Lew, Axel Ariel	225/14	`axel.lew@hotmail.com`
Noli Villar, Juan Ignacio	174/14	`juaninv@outlook.com`

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

En el presente trabajo resolveremos 3 problemas algorítmicos que nos fueron dados, respetando sus requerimientos de complejidad temporal, analizaremos empíricamente los tiempos de ejecución de sus implementaciones, mostraremos un pseudocódigo de los mismos, y las experimentaciones realizadas con sus debidos gráficos.

2. Problema 1: Telégrafo

2.1. Idea general del problema

Se ha decidido conectar telegráficamente todas las estaciones de un sistema férreo que recorre el país en abanico con origen en la capital (el kilómetro 0). Se nos ofrece cierta cantidad de kilometros de cable para conectar la ciudades de cada ramal. Al ser escaso el presupuesto, se busca lograr conectar la mayor cantidad de ciudades con los metros asignados, sin hacer cortes en el cable.

Se nos propone resolver cuántas ciudades se pueden conectar para cada ramal, con una complejidad de O(n), siendo n la cantidad de estaciones en cada ramal.

Para ello se nos brinda un archivo de entrada, el cual tiene para cada ramal dos líneas: la primera contiene un entero con los kilómetros de cable dedicados al ramal y la segunda los kilometrajes de las estaciones en el ramal sin considerar el 0. Luego de ejecutar nuestro algoritmo, la salida del mismo debe contener, para cada ramal de la entrada, una línea con la cantidad de ciudades conectables encontradas.

Un ejemplo de archivo de entrada puede ser, (extracto del archivo Tp1Ej1.in):
6
6 8 12 15
35
8 14 20 40 45 54 60 67 74 89 99
100
35 87 141 163 183 252 288 314 356 387
90
6 8 16 19 28 32 37 45 52 60 69 78 82

El mismo indica, en su primer línea que para el ramal 1 tenemos 6km de cable, y en su segunda línea que dicho ramal contiene (además de la capital, implícita, en el kilómetro 0) una estación en el kilómetro 6, otra en el 8, otra en el 12 y la última en el kilómetro 15. Luego para el ramal 2, tenemos 35 kilómetros de cable, y estaciones en los kilómetros: 8 14 20 40 45 54 60 67 74 89 y 99. Así sucesivamente para el resto de los ramales.

El archivo de salida luego de ejecutarse nuestro algoritmo deberá ser de la siguiente pinta, (extracto del archivo Tp1Ej1.out):
3
6
4
14

Este último archivo indica la cantidad de ciudades que se pueden conectar para cada ramal. En el caso del ramal 1, para el cual se tienen 6km de cable disponibles, y contiene ciudades en los kilómetros: 0 6 8 12 15 vemos que la solución debería ser que se pueden conectar como máximo 3 ciudades, a continuación explicaremos cómo se deduce esto.

Si conectamos la capital con la ciudad del kilómetro 6, al tener sólo 6km de cable, nuestra solución sería que pudimos conectar sólo 2 ciudades. Pero como debemos maximizar esta cantidad, podemos ver que si en vez de conectar a la capital con la primer estación del ramal, conectamos la ciudad del kilómetro 6, con su siguiente y con la del kilómetro 12, entonces como entre el kilómetro 6 y el 8 hay una diferencia de 2kms y entre el 8 y el 12 una diferencia de 4kms, vemos que la máxima cantidad de estaciones conectadas con 6km de cable para el ramal 1, es 3. La misma lógica se la aplica para los ramales restantes.

2.2. Explicación y pseudocódigo

int conectar(vector<int> v , int longitud del cable) {
int resTemp ← 1 $O(1)$
int start ← 0 $O(1)$
int actual ← 0 $O(1)$
int aux ← 0 $O(1)$

while *(la longitud del cable sea > 0 y v[actual] no sea el ultimo)* **do**
{
en el peor caso el ciclo se ejecuta n veces.
aux ← longitud del cable $O(1)$
si es el primer elemento del vector (el km 0) *la complejidad de la sentencia del if es de* $O(1)$.
a la longitud del cable le restamos el valor del segundo elemento. $O(1)$
si no es el kilómetro 0
a la longitud del cable le restamos la diferencia entre dicho elemento y su próximo. $O(1)$

}

si la longitud del cable sigue > 0
incrementamos en 1 resTemp $O(1)$

incrementamos en 1 actual, siempre. $O(1)$

si resTemp sigue valiendo 1 *la complejidad de la sentencia del if es de* $O(1)$.
seteamos resTemp ← 0; porque no se conectó ninguna ciudad. $O(1)$

si nos pasamos y la longitud del cable < 0 *la complejidad de la sentencia del if es de* $O(1)$.
volvemos al valor anterior: la longitud del cable ← 'aux' $O(1)$

while *la longitud del cable sea > 0 y el elemento actual no sea el último)* **do**
{
en el peor caso el ciclo se ejecuta n veces.
int conectadas
si(resTemp ←0) *la complejidad de la sentencia del if es de* $O(1)$.
conectadas ← 0; $O(1)$
start++; $O(1)$
si no conectadas ← resTemp - 1; $O(1)$
longCable ← longCable + (v[start+1]-v[start]); $O(1)$
incrementamos start en 1; $O(1)$
while *(longitud del cable ≥ 0 y v[actual] no es el último)* **do**
{

en el peor caso el ciclo se ejecuta longituddelcable veces.
aux ← longCable; $O(1)$
a longitud del cable le restamos (v[actual+1]- v[actual]); $O(1)$
incrementamos en 1 actual $O(1)$
si (longitud del cable ≥ 0) *la complejidad de la sentencia del if es de* $O(1)$.
incrementamos en 1 conectadas $O(1)$
si no
longitud del cable ← 'aux' $O(1)$
}
si (conectadas > resTemp) *la complejidad de la sentencia del if es de* $O(1)$.
resTemp ← conectadas $O(1)$
} **si** (resTemp es 1) *la complejidad de la sentencia del if es de* $O(1)$.
resTemp ← 2; $O(1)$
}
}

2.3. Deducción de la cota de complejidad temporal

Dedujimos que el algoritmo indica cuantas ciudades se pueden conectar, para cada ramal en O(n), siendo n la cantidad de estaciones en cada ramal, ya que..

FALTA explicar porque dedujimos q es O(n)

Vamos a mostrar la implementación de un test generado sin ninguna intencionalidad, pero antes explicaremos detalladamente como fue creado.

Implementamos una función que genera números random para el archivo que le vamos a pasar por entrada a nuestro algoritmo con los siguientes criterios:

- Elegimos en este ejemplo que el primer ramal iba a contar con una estación, el segundo con 101, el tercero con 201 y asi sumando de a 100.
- Para el valor de la longitud de cable disponible de cada ramal, decidimos que sea un número random entre 1 y la cantidad de estaciones de cada ramal.
- Para los kilometrajes de las estaciones tuvimos en cuenta, que estos debían estar ordenados de menor a mayor, sin contener el kilómetro 0. Para definir esto hacemos algo de la pinta:

  ciudad = ciudad + (random.randrange(i+1)+1)

  random.randrange(i+1) da un número random entre 0 e $i + 1$, como inicialmente i es 0 tuvimos que sumarle 1. A su vez a esto lo incrementamos en 1 porque un número random entre 0 e $i + 1$ puede llegar a darnos 0, y no queremos que esto pase, ya que el kilómetro 0 no debe figurar en el archivo de entrada.

  Por otro lado, al hacer "ciudad = ciudad + ..." nos aseguramos que los kilometrajes esten en orden creciente.

Podemos ver el código de este test implementado en python acá:

```
file = open('ejemploConNumerosRandom.in', 'w+')
file2 = open('ejemploTamCiudades.txt', 'w+')
for x in xrange(1,10000, 100):
    i = 0
    ciudad = 0
    file.write(str(random.randrange(x)) + ' \n')
    file2.write(str(x) + ' \n')
    while i < x:
```

```
            ciudad = ciudad + (random.randrange(i+1)+1)
            file.write(str(ciudad) + ' ')
            i = i + 1
    file.write(' \n')
file2.close()
```

Una vez que generamos el archivo del input con dicho test, ejecutamos nuestro algoritmo obteniendo el archivo de salida con la cantidad de ciudades conectadas para cada ramal, e imprimimos por pantalla el tiempo promedio en milisegundos que tardó nuestro algoritmo en calcular la máxima cantidad de estaciones conectadas de cada ramal. ¿Por qué el tiempo promedio? Bueno, al ejecutarlo un par de veces nos dimos cuenta que se obtenían valores parecidos pero no idénticos, entonces decidimos correr el algoritmo una cierta cantidad de iteraciones (en este caso 100), e ir acumulando los tiempos para luego dividir este acumulador por 100 y así obtener un valor promedio de los tiempos en los que se tarda en resolver el problema para cada ramal.

Con estos tiempos creamos el gráfico de la figura 1 que mostramos a continuación, en el que hacemos una comparación con la gráfica de O(n) y observamos como nuestro algoritmo cumple dicha complejidad.

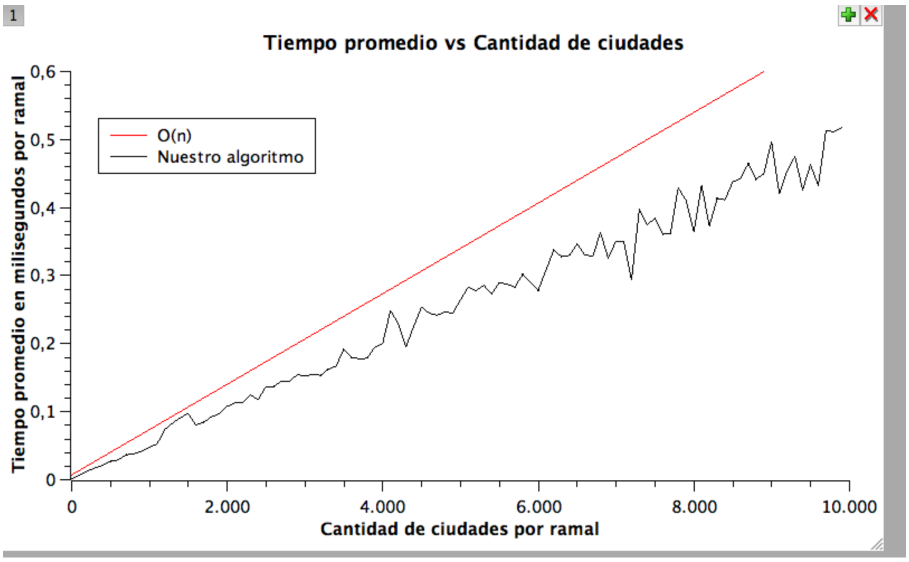

Figura 1: Comparación con O(n). Dado un archivo de entrada con longitud de cable y kilometrajes de estaciones random.

2.4. Demostración formal

2.5. Experimentaciones

3. Problema 2: A Medias

3.1. Idea general del problema

Se nos propone realizar un algoritmo en el que dados n número enteros en cualquier orden se debe devolver otros n números, donde el i-ésimo de ellos represente la parte entera de la mediana de los primeros i números de la entrada. La mediana de un conjunto ordenado de n números se define como $x_{(n+1)/2}$ si n es impar, o como $(x_{n/2} + x_{n/2+1)})/2$ si n es par.

3.2. Explicación y pseudocódigo

Lo que hicimos fue dividir un arreglo (L) de tamaño n en dos conjuntos (multiconjuntos, ie: pueden tener repetidos), uno con los elementos menores a cierto número (A) y otro con los mayores (B). Los elementos iguales a ese número pueden ir en cualquiera de los dos. Entonces sabemos que, si ordenaramos el arreglo, L[|A|] = max(A) y L[|A|+1] = min(B). Tres casos importantes:

- Si ambos conjuntos tienen la misma cantidad de elementos entonces L[|A|] = L[n/2] = max(A) y L[n/2+1] = min(B). Entonces la mediana(L) = (max(A)+min(B))/2
- Si |A|= |B|+ 1 (n es impar), entonces L[(n+1)/2] = max(A). Entonces la mediana(L) = max(A)
- Si |A|+ 1 = |B|(n es impar), entonces L[(n+1)/2] = min(B). Entonces la mediana(L) = min(B)

Nuestro algoritmo lo que hace es:

- Insertar el primer elemento (asumimos que no puede ser vacío el arreglo original) en el conjunto de los elementos más grandes (B) e insertarlo también en la posición 1 del resultado, ya que es mediana.
- Hasta que no queden elementos en el arreglo original, se agrega el elemento i del arreglo original en el conjunto A si es menor a la mediana del paso anterior y al conjunto B si es mayor. Si la diferencia entre la cantidad de elementos de ambos conjuntos es igual a 2, entonces quitamos el mayor elemento de A o el menor elemento de B y lo insertamos en el otro para que nos queden dos conjuntos de igual tamaño. Entonces podemos utilizar la propiedad de arriba y sacar fácilmente la mediana, obteniendo el máximo y/o mínimo de los conjuntos correspondientes. Ponemos la mediana en la posición i del resultado y avanzamos a la siguiente iteración.

3.3. Deducción de la cota de complejidad temporal

Los conjuntos utilizados son <**multiset**> cortesía de C++. La inserción y borrado cuesta O(log(n)) siendo n la cantidad de elementos del conjunto, ya que usamos las funciones de C++: **insert** y **erase** de los **multiset**, respectivamente. La función **erase** cuesta O(log(n)) + O(m) siendo n = cantidad de elementos del conjunto, y m el número de elementos eliminados, en nuestro caso, siempre es uno porque llamamos a erase una vez con el elemento mínimo y otra con el máximo.
Obtener máximo y mínimo nos cuesta O(1), pues usamos **begin()** y **rbegin()** de complejidad constante.

Tenemos como entrada un vector de tamaño n. Los pasos a realizar son:

1) Crear el vector donde se guardará y devolverá el resultado: se realiza en O(n).

2) Insertar el primer elemento del arreglo original en el arreglo resultado es O(1), porque usamos **operator[]** de complejidad constante.

3) Luego, se realizan n-1 iteraciones, en las cuales:

- Se realiza una comparación O(1) y se hace una inserción en alguno de los conjuntos en O(log(n)).
- Si la diferencia entre la cantidad de elementos de ambos conjuntos es igual a 2, se elimina el mayor elemento de A o el menor elemento de B en O(log(n)) y lo insertamos en el otro en O(log(n)).
- Obtenemos el máximo y/o mínimo de los conjuntos correspondientes para calcular la mediana y la insertamos en el resultado en O(1).

Aclaración: en este ciclo, en realidad, las complejidades no son O(log(n)) sino que son O(log(i)) siendo i el número de iteración. Pero como $i < n \; \forall \; i$, entonces lo acotamos por O(log(n)).

Entonces la complejidad total del algoritmo es O(n + (n-1)log(n)) = O(n log(n)).

3.4. Demostración formal

3.5. Experimentaciones

4. Problema 3: Girls Scouts

4.1. Idea general del problema

4.2. Explicación y pseudocódigo

4.3. Deducción de la cota de complejidad temporal

4.4. Demostración formal

4.5. Experimentaciones

5. Código