DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

September 18, 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

Contents

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

El algoritmo empieza ubicando dos índices en la primera ciudad, "start" y "actual" . Luego, si alcanza el cable, avanza "actual" hasta la segunda ciudad, y le resta al cable esa distancia (entre la primer ciudad y la segunda), asi continúa (restando la distancia entre la tercera y la segunda, luego entre la cuarta y la tercera, etc.) hasta que ya no se puedan conectar mas ciudades, esto ocurre cuando el cable pasa a ser negativo. Cada vez que una nueva ciudad es conectada, la variable "conectadas" aumenta en uno y se aumenta en uno el indice "actual", en caso de llegar a la última ciudad (cuando "actual" apunta a la ultima ciudad) termina y retorna la variable "restemp" que es igual a "conectadas" . Si no, calcula la distancia entre la ciudad apuntada por "start" y su siguiente, esa distancia se le suma al resto del cable que quedo disponible y avanza el indices "start" , ademas le resta uno a la variable "conectadas" (siempre que esta sea mayor a 0) pues este proceso simboliza que una ciudad fue desconetada. Con esta nueva cantidad de cable se fija ahora si se puede avanzar más el puntero "actual" , repitiendo el proceso de restarle al cable la distancia entre la última ciudad conectada (apuntada por "actual") y su siguiente. Este proceso se repite hasta que se termine el cable otra vez (repitiendo el proceso anterior) o se llegue a la última ciudad. Si cuando se termina el cable o se llega al final, el valor de "conectadas" es mayor que "restemp" , se actualiza "restemp" .

A continuación mostramos un pseudocódigo de la implementación explicada anteriormente.

```
int conectar(vector<int> v , int cable) {
    int resTemp ← 0                                                          O(1)
    int start ← 0                                                            O(1)
    int actual ← 0                                                           O(1)
    int aux ← 0                                                              O(1)
    mientras (actual no apunte al último elemento del vector) {              O(n)
      mientras(cable ≥ 0 y actual no apunte al ultimo elemento del vector) { O(cable)
        Guardamos en aux el cable, por si este pasa a ser negativo           O(1)
        Le restamos al cable la distancia entre lo apuntado por actual y su siguiente; O(1)
        si (longitud del cable ≥ 0) {                                        O(1)
            Incrementamos conectadas en 1 y avanzamos el puntero actual      O(1)
        }
        Si conectadas es igual a uno se lo cambia por dos;                   O(1)
      }
      Si el cable en negativo, se actualiza su valor por el de aux
      Si conectadas es mayor que restemp se actuliza el valor de restemp por el de conectadas
      si(resTemp == 0) {                                                     O(1)
        Conectadas ← 0;                                                      O(1)
        Avanzamos los dos indices, start y acutal;                           O(1)
      } si no {
        Decrementamos conectadas en 1, ya que desconectamos la primer ciudad
        Sumamos al cable la distancia entre lo apuntado por start y su siguiente;1
        Incrementamos start en 1;                                            O(1)
      }
    }
    Devolvemos restemp, en caso de que sea uno, se cambia su valor por dos;  O(1)
}
```

2.3 Deducción de la cota de complejidad temporal y correctitud

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. El algoritmo termina cuando el segundo puntero, "actual", llega al final. Para esto cuenta con dos ciclos (explicados anteriormente), el primero se ejecuta como mucho n veces, ese caso ocurre cuando se da un cable muy corto y en cada iteración se avanzan los dos índices solo en uno, pasando asi por todos los elementos, en este caso el segundo while solo iteraria una vez pues el cable pasaria a ser negativo instantaneamente, por otro lado el primer while podria llegar a ejecutarse solo una vez si se da un cable muy largo, en este caso el segundo while es el que iteraria n veces, llevando el puntero "actual" hasta el final, pero sin mover "start" por lo que seria considerado como el mejor caso, aunque este tambien tiene una complejidad O(n).
En otro caso, el segundo while del algoritmo se ejecuta mientras que la longitud del cable sea positiva y el indice "actual" no apunte a la ultima ciudad, este es el ciclo que va sumando ciudades al ramal y guarda cuantas ciudades se conectaron. Mientras que va conectando ciudades, el algoritmo avanza el segundo puntero, cuando el cable pasa a ser negativo recien ahi se avanza el primer puntero ,"start", suma esa diferencia al cable, se fija si hay que actualizar el resultado, y vuelve a comenzar el segundo ciclo. Este procezo continua hasta que "actual" apunte a la ultima ciudad. Notar que en cada iteracion, "actual" no siempre avanza, pero cuando no avanza, "start" si lo hace y si "start" llega a alcanzaz a "actual" , en la proxima iteracion ,si "start" vuelve a avanzar, lo empuja. Aca se puede obervar el peor caso, que ocurre cuando "actual" llega hasta cierto punto arbitrario, y luego "start" lo alcanza, si esto pasa todo el tiempo, se recorreria linealmente el vector dos veces, y aunque sigue siendo O(n), es el caso que mas iteraciones produce y por eso es considerado el peor caso.

El algoritmo termina cuando el segundo puntero llega al final, en cada iteración el segundo puntero, "actual" avanza en uno, y cuando no avanza, avanza "start" tambien en uno. Notar que si "start" alcanza a "actual", en la siguiente iteracion el algoritmo avanza los dos. Como ninguno de los dos indices retrocede nunca, y si "start" alcanza a "actual" en la siguiente iteracion, pase lo que pase "actual" avanza (ya que si no da el cable avanza "start" y empuja a "actual" y si alcanza el cable simplemente avanza "actual") se puede deducir que el algortimo es lineal y por lo tanto tiene una complejidad O(n). Por esta razón decimos que nuestro algoritmo es correcto, ya que siempre llega al final, y siempre devuelve el resultado esperado

Las funciones que usamos: $>$, $<$, $\geq$, $+$, $-$, $==$, tienen complejidad constante al igual que las asignaciones. También sabemos por la documentación de C++, que el *operator*[] del vector es de complejidad O(1) al igual que la función del vector *.back*().

2.4 Casos de test, experimentación, y gráficos

2.4.1 Caso random

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

El archivo que contiene dicha función se puede encontrar en la carpeta Ejercicio 1, y se llama: generadorArchivosInRandom.py.

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 de 100 corridas de nuestro algoritmo, en milisegundos. Este es el tiempo que tarda en calcular la máxima cantidad de estaciones conectadas de cada ramal. Decidimos calcular el tiempo promedio porque al ejecutarlo un par de veces nos dimos cuenta que el tiempo varía aunque resuelva la misma instancia, entonces decidimos correrlo 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.

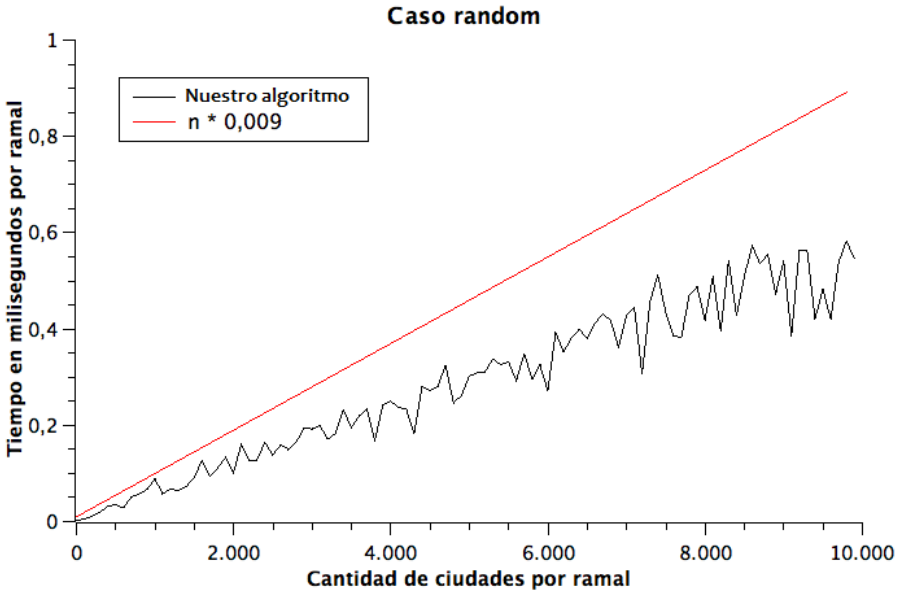

Figure 1: Comparación con O(n). Dado un archivo de entrada con longitud de cable y kilometrajes de estaciones random.

2.4.2 Peor caso

El próximo test generado consiste en tener una ciudad al final muy lejos, lo que produciria que "actual" llegue hasta el anteultimo, luego start lo alcance y esto produzca que se pase por todas las ciudades dos veces. Esto lo consideramos un Peor caso ya que el resto de los casos no haces falta pasar dos veces por todas las ciudades. Para probar esto, escribimos un script que nos asegure que la ultima ciudad este muy alejada. En cuanto a las estaciones del ramal, configuramos números random y el ultimo muy grande. Estos archivos se encuentran en la carpeta "Ejercicio 1" con prefijo "2doEjemplo", y"generadorArchivosInRandom.py" es el script con los distintos tests, comentados. Veamos un ejemplo:

- La longitud del cable no alcanza para conectar ninguna ciudad:
  in:

1
3 6 8 12 15 20 24 49 58 70 90 123
out:
0

En este caso la longitud del cable es 1, y todas las distancias son mayores que 1, por lo tanto está bien que nuestro algoritmo devuelva que conectó 0 ciudades.

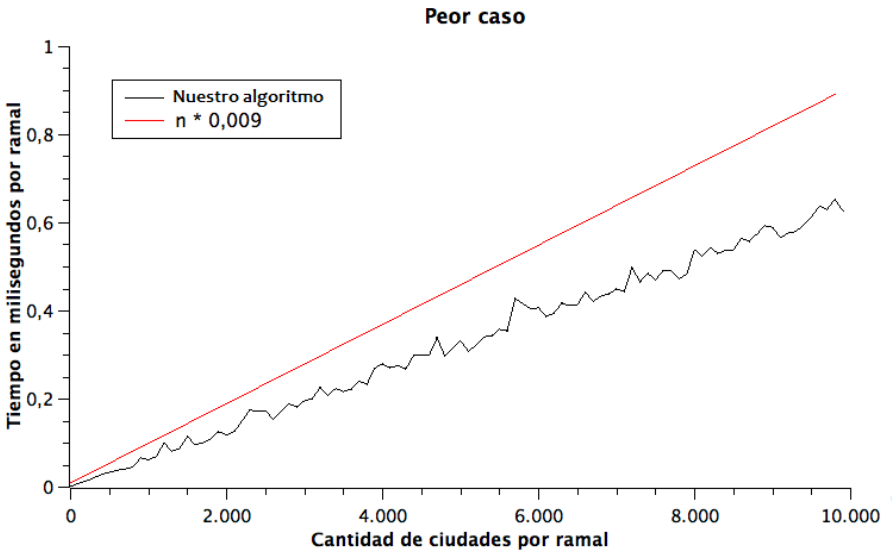

Figure 2: Comparación con O(n). Dado un archivo de entrada con un cable corto y kilometrajes de estaciones random.

En la figura 2 vemos como nuestro algoritmo sigue respentando el límite de complejidad propuesto por la cátedra, a pesar de que éste sea un caso “border”.

2.4.3 Mejor caso

Otro test que decidimos hacer fue el caso contrario a este, que sería tener un cable demasiado largo, que permita conectar todas las ciudades de los ramales. Esto lo consideramos como el mejor caso, ya que al tener un cable suficientemente largo como para conectar todas las ciudades, el primer puntero (”start”) no es necesario que se mueva, solo avanza el segundo. Para ver esto , escribimos un script de manera tal que la cantidad de kilómetros de cable siempre supere la distancia entre la primera ciudad y la ciudad del último kilómetro del ramal. Al medir los tiempos promedios vemos que nuestro algoritmo sigue tardando menos que O(n), y lo podemos observar en la figura que sigue, pero antes veamos un ejemplo.

- El cable alcanza para conectar todas las ciudades:
  in:
  10000
  1 4 67 78 90 95 120 150 270 380 456 900 1300 1809 5546 8403
  out:

17

En este caso al tener un cable de longitud = 10000km y todas las estaciones estar a distancia menor que 10000km, el algoritmo devuelve 17 que son la cantidad de estaciones del ramal + la ciudad de kilómetro 0, pues la distancia entre ésta ciudad y la primera es de 1km. Veamos el grafico 3.

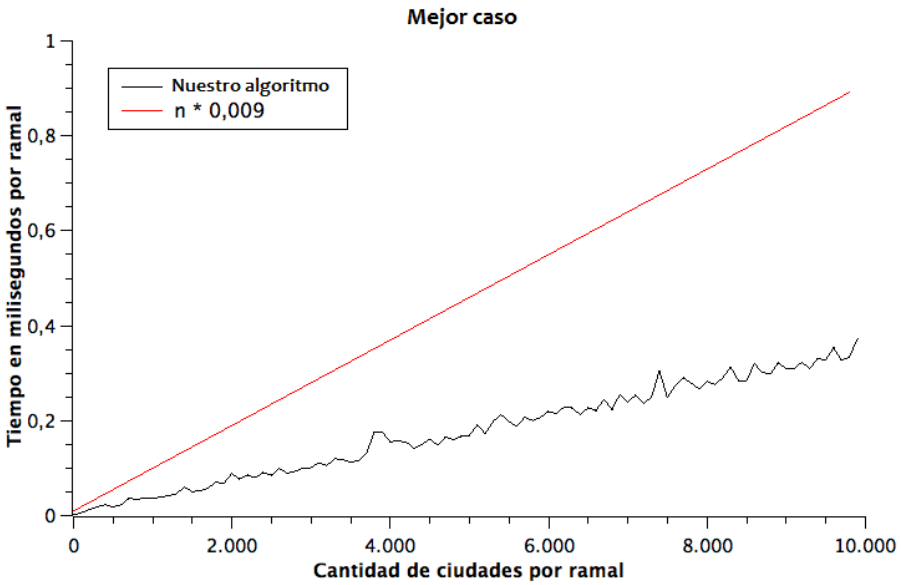

Figure 3: Comparación con O(n). Dado un archivo de entrada con un cable muy largo y kilometrajes de estaciones random.

2.4.4 Peor caso vs Mejor caso vs O(n)

A continuación ilustramos un gráfico en donde comparamos el tiempo que tarda el algoritmo en el mejor caso, con el tiempo del peor caso y con O(n). Para realizar el gráfico definimos la línea O(n) como n $*$ 0,01.

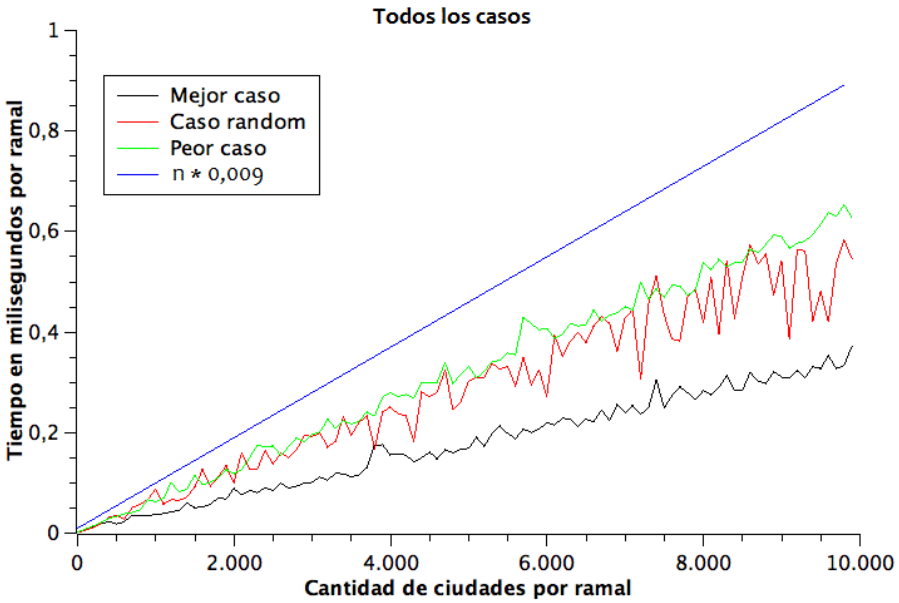

Figure 4: Comparación con O(n) del mejor y peor caso anteriormente explicados.

2.4.5 Otros casos interesantes:

- Cuando el cable alcanza para conectar solo las primeras ciudades:
  in:
  9
  1 2 3 4 5 40 55 68 79 99 130 139 200 259 2889
  out:
  6

Al tener un cable de longitud 9km, las primeras 5 estaciones estar a 1 kilómetro de distancia entre ellas, y las siguientes distancias superar los 9km, si contamos estas 5, y a la ciudad del kilómetro 0, no devuelve las 6 ciudades que conecta.

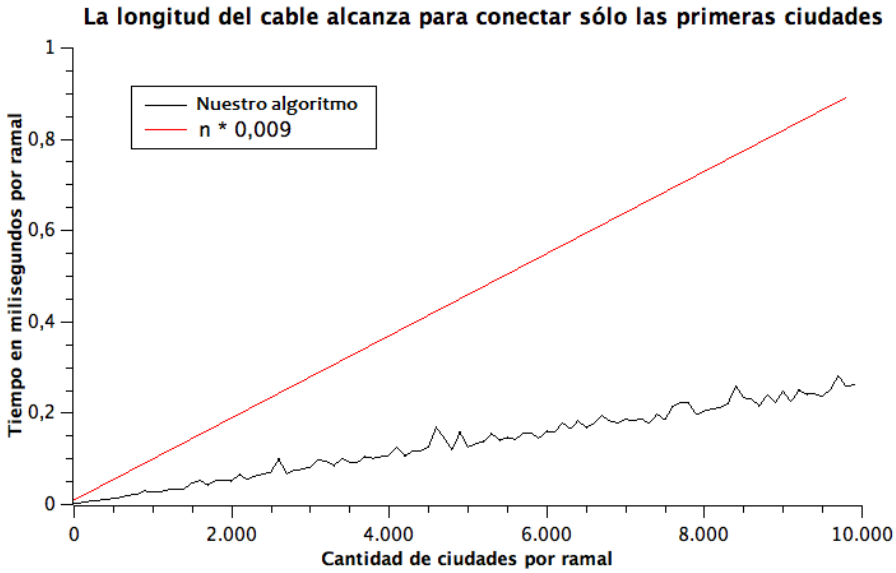

Figure 5: Comparación con O(n). Dado un archivo de entrada donde entre las primeras ciudades hay una distancia corta.

- Cuando el cable alcanza para conectar sólo las últimas ciudades:
  in:
  8
  10 20 35 47 82 99 120 143 155 200 267 298 299 300 301
  out:
  4

Al ser un cable de longitud 8km, y las primeras 12 estaciones estar a distancia mayor que 8km, no conecta ninguna de ellas, pero si lo hace con las últimas 4 ya que la distancia de las mismas es menor que 8km.

- Cuando el cable no alcanza y no se conectan ciudades:
  in:
  1
  4 8 12 16 20 24 28 32
  out:
  0

  La distancia entre todas las estaciones es de mas de 1km, y el cable tiene un largo de 1km, por lo tanto, al faltarle siempre cable para poder conectar al menos dos ciudades, devuelve 0. Esto es correcto ya que no alcanza el cable y no "conecta" ciudades.

  A continuación mostramos el gráfico que resulta de medir el tiempo que tarda el algoritmo al correr un archivo de entrada que generamos con ciudades que están a una distancia mayor a un, con un cable de longitud 1 y compararlo con O(n), en particular para este ejemplo, lo comparamos con $0{,}006 * O(n)$. Los archivos que usamos son: "generadorArchivosInRandom.py" (ejemplo 6), "2doEjemploTiempoEnMilisegundos.txt",

"2doEjemploCableCorto.in", "2doEjemploCableCorto.out" y , "ejemploTamCiudades.txt"

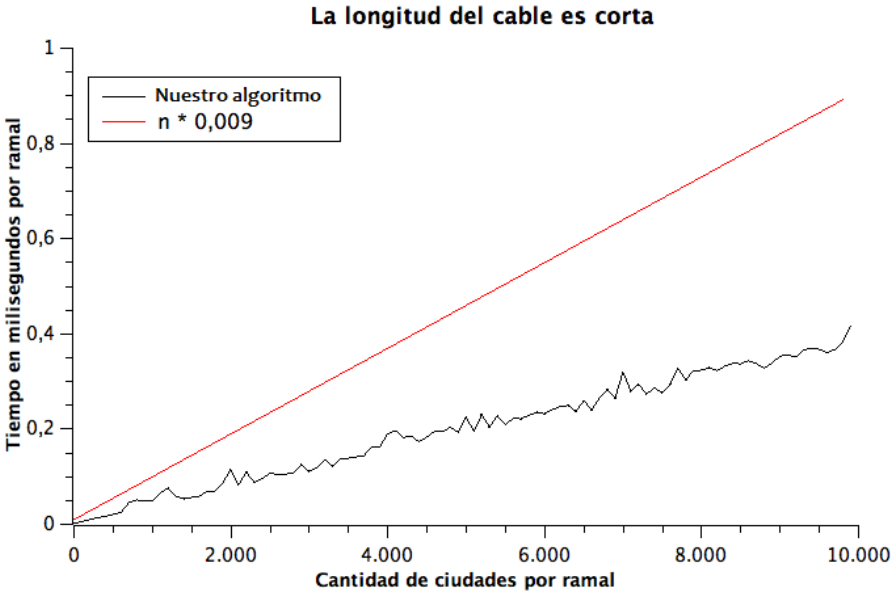

Figure 6: Comparación con O(n). Dado un archivo de entrada donde se da un cable corto para conectar las ciudades.

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. Un ejemplo de entrada puede ser:

2 7 2 8 4 9 1 6 5

Su salida deberia ser:

2 4 2 4 4 5 4 5 5

3.2 Explicación y pseudocódigo

Nuestro algoritmo lo que hace es:

- Si el arreglo original no contiene elementos, entonces devolvemos un vector vacio
- Creamos dos conjuntos vacios, uno con los elementos mas chicos que la mediana (A), y otro con los elementos mas grandes (B). En caso de tener un elemento igual a la mediana, puede ir en cualquier conjunto
- Insertar el primer elemento 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, utilizando la propiedad que se encuentra en "Correctitud y Deducción de la cota de complejidad temporal" podemos 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.

A continuación mostraremos su pseudocódigo y deduciremos su complejidad:

```
vector<int> medianas(vector<int> a){
        si (a es vacio) {                                              O(1)
            devolvemos un vector vacio                                 O(1)
    }
    creamos el vector res de tamaño igual que a, inicializado con ceros   O(tamaño de a)
    res[0] ← a[0]                                                      O(1)
    creamos el multiset vacío: masGrandes                              O(1)
    creamos el multiset vacío: masChicos                               O(1)
    int mínimo                                                         O(1)
    int máximo                                                         O(1)
    insertamos a masGrandes el primer elemento del vector              O(1)
     para cada (int i = 1; i < al tamaño de a; i++){                   O(tamaño de a)
        si (el elemento i de a ≥ el elemento i-1 de res){              O(1)
            insertamos a masGrandes el elemento i de a   O(log(tamaño de masGrandes))
         } si no {
            lo insertamos en masChicos                   O(log(tamaño de masChicos))
```

```
        }
        si (el tamaño de masGrandes - tamaño de masChicos == 2) {
O(1)
            minimo ← ∗ masGrandes.begin()                                   O(1)
            borramos mínimo de masGrandes                  O(log(tamaño de masGrandes))
            insertamos minimo a masChicos                    O(log(tamaño de masChicos))
        } si no, si(el tamaño de masGrandes - tamaño de masChicos == -2) {
la complejidad de la sentencia del if es de O(1)
            maximo ← ∗ masChicos.rbegin()                                   O(1)
            borramos máximo de masChicos                   O(log(tamaño de masChicos))
            insertamos máximo a masGrandes                 O(log(tamaño de masGrandes))
        }
        si (el tamaño de masGrandes == el tamaño de masChicos) {            O(1)
            minimo ← ∗ masGrandes.begin()                                   O(1)
            maximo ← ∗ masChicos.rbegin()                                   O(1)
            posicion i de res ← (maximo + minimo)/2;                        O(1)
        } si no, si (el tamaño de masGrandes - el tamaño de masChicos == 1) {
O(1)
            minimo ← ∗ masGrandes.begin()                                   O(1)
            posición i de res ← minimo                                      O(1)
        } si no, si (el tamaño de masGrandes - tamaño de masChicos == -1) {
O(1)
            maximo ← ∗ masChicos.rbegin()                                   O(1)
            posición i de res ← maximo                                      O(1)
        }
    }
    devolvemos res                                                          O(1)
}
```

3.3 Correctitud y Deducción de la cota de complejidad temporal

¿Por que nuestro algoritmo es correcto?
Utilizamos la siguiente propiedad: Sea L un arreglo de tamaño n. Dividamos sus elementos en dos conjuntos (multiconjuntos, ie: pueden tener repetidos), uno con los elementos menores a cierto número (A) y otro con los mayores (B), donde los elementos iguales a ese número pueden ir en cualquiera de los dos. Entonces, 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 ciclo tiene un invariante que nos asegura que en todo momento (al comenzar y finalizar el ciclo) la diferencia entre los conjuntos es menor a dos. Entonces en cada iteracion vamos a poder encontrar la mediana obteniendo el minimo y/o maximo de los conjuntos respectivos. Esto, sumado a que el ciclo termina, nos garantiza su correctitud

¿Por que nuestro algoritmo cumple la complejidad pedida?
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 **multisets**, 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. Además usamos la función **size** de los **multisets** 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 Experimentación

Vamos a analizar los tiempos de nuestro algoritmo para entradas de distinto tamaño, divididos en tres casos (todos O(n log n)):

- Peor caso: Corresponde al caso donde en la mitad de las iteraciones del ciclo, la diferecia entre la cantidad de elementos entre los conjuntos masGrandes y masChicos es igual a dos. Entonces hay que realizar en cada una de estas un borrado y una insercion. No existe peor caso que este, ya que al realizar una insercion y borrado, los conjuntos quedan con la misma cantidad de elementos y se necesitan dos iteraciones mas ciclo para tener que volver a reacomodar los elementos. Entradas crecientes y decrecientes son ejemplos de peor caso, que es lo que utilizaremos
- Mejor caso: Resulta cuando en ninguna iteracion la diferecia entre la cantidad de elementos entre los dos conjuntos es igual a dos, y no hay que hacer borrado e insercion. Para que pase esto, nuestra entrada consistira en una cadena de numeros, donde la subcadena de numeros en posicion par es creciente, y la subacadena de numeros en posicion impar es decreciente
- Caso random: Utilizando la funcion random de python, generamos entradas al azar de diferentes tamaños

Los tiempos se miden de la misma forma que en el ejercicio 1.

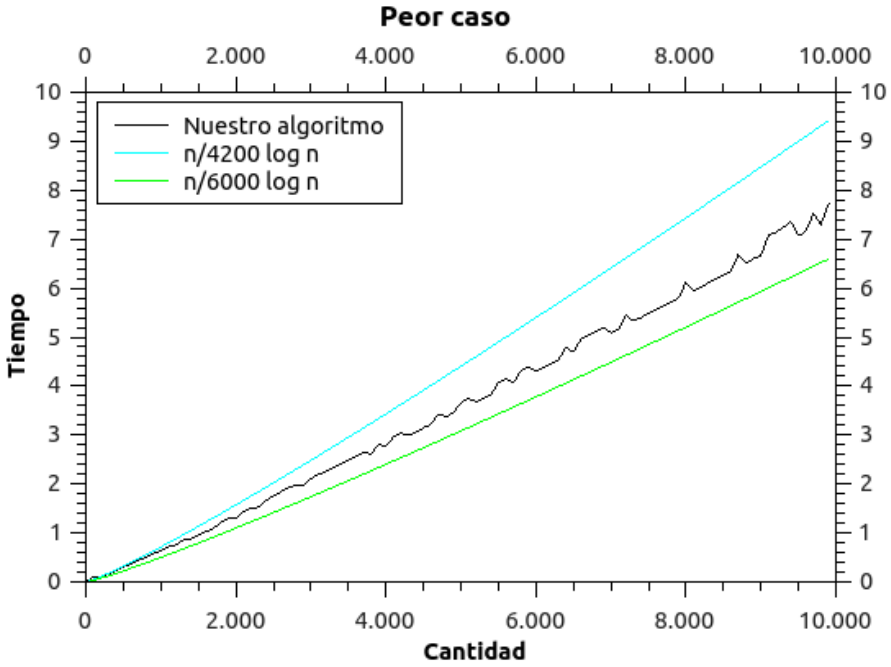

Figure 7: Comparación del peor caso con O((n/6000)log(n)) y O((n/4200)log(n)). Cantidad representa la cantidad de numeros pasados como parametro y el tiempo se mide en milisegundos

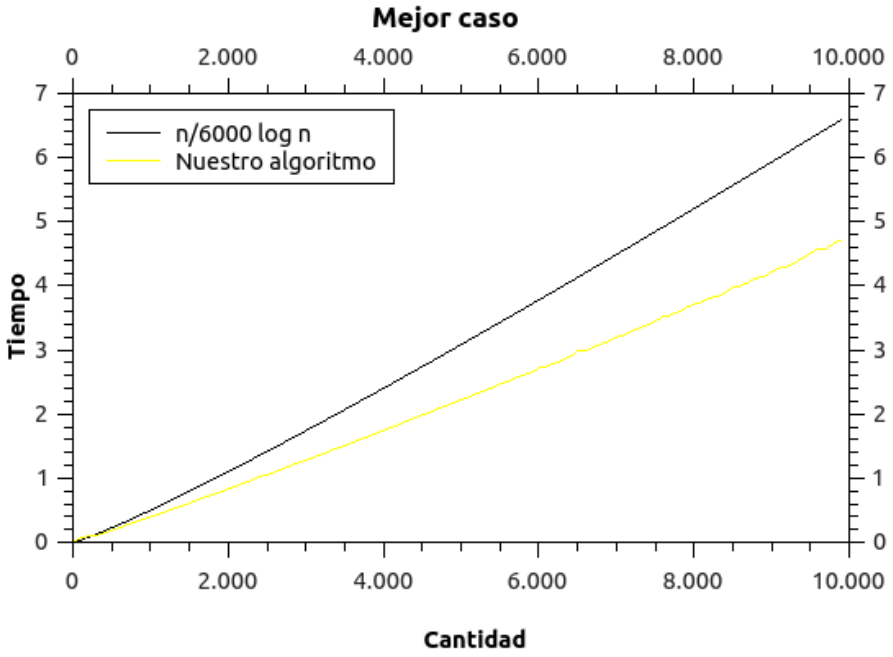

Figure 8: Comparación del mejor caso con O((n/6000)log(n)). Cantidad representa la cantidad de numeros pasados como parametro y el tiempo se mide en milisegundos

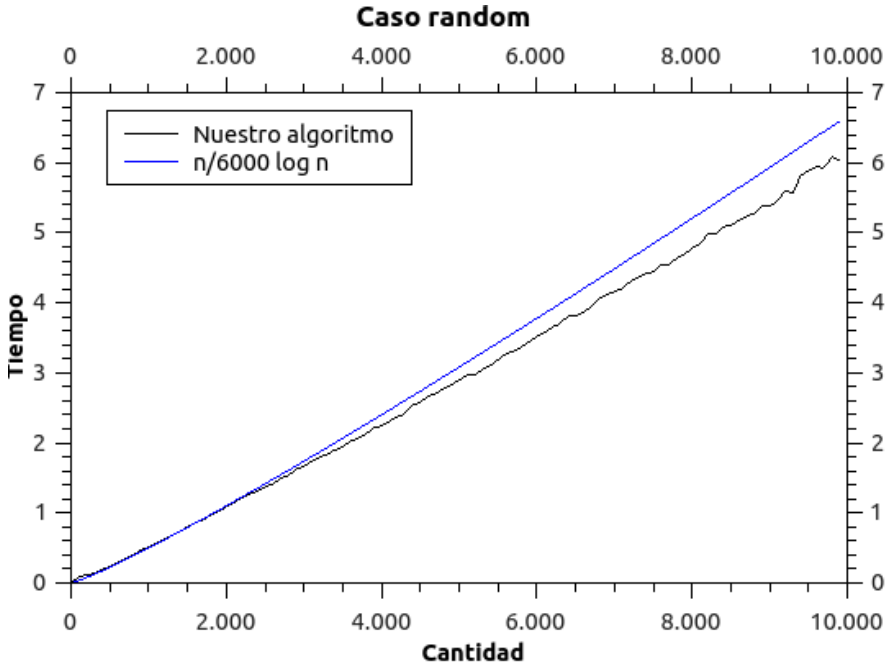

Figure 9: Comparación de un caso random con O((n/6000)log(n)). Cantidad representa la cantidad de numeros pasados como parametro y el tiempo se mide en milisegundos

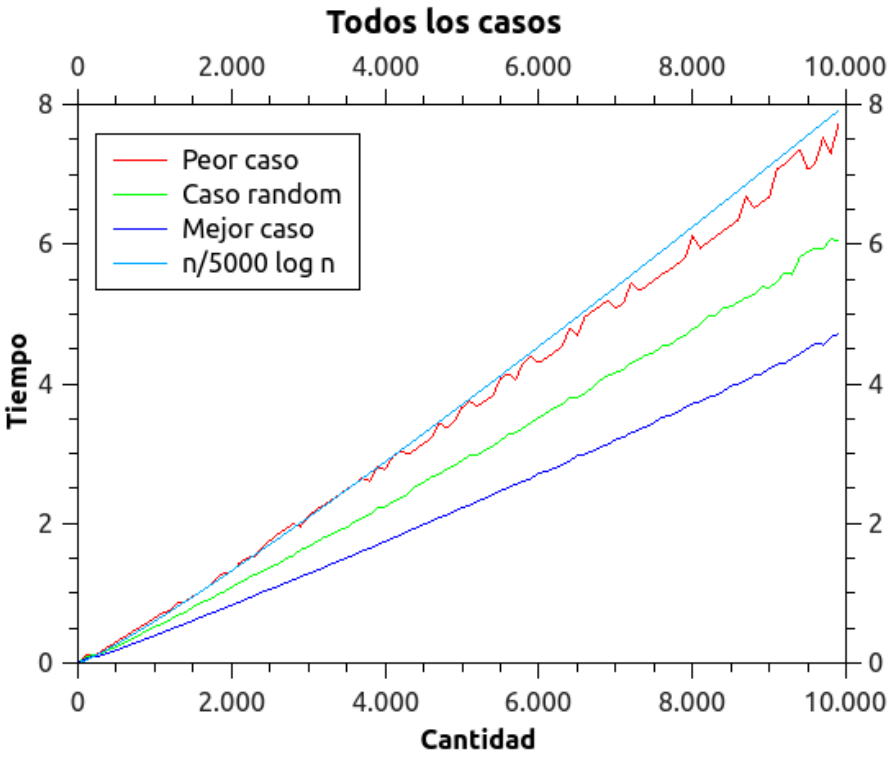

Figure 10: Comparación de los tres casos anteriores con O((n/5000)log(n)). Cantidad representa la cantidad de numeros pasados como parametro y el tiempo se mide en milisegundos

4 Problema 3: Girls Scouts

4.1 Idea general del problema

El ejercicio nos propone diseñar un algoritmo para resolver el siguiente problema: dado un grupo de exploradoras, y el conjunto de amigas de cada una de ellas, organizar una ronda de manera tal que exista la menor distancia posible entre cada amistad, es decir, minimizar la suma de las distancias entre todos los pares de amigas.
La complejidad de la solución debe ser estrictamente menor que $O(e^e.a^2)$, donde "e" es la cantidad de exploradoras en cada grupo, y "a" la cantidad de amistades.

Algunos ejemplos de posibles datos de entrada del problema son:
a bcde;b acde;c abde;d abc;e abc
a bcd;b ae;c ad;d ac;e b
a fb;b gc;d gc;f agh;e hd
x yz

Cada línea corresponde a un grupo de exploradoras; y se compone de una exploradora, seguida por una sucesion de amistades separadas por ";". Por ejemplo, en la primer línea, el grupo de exploradoras esta compuesto por "a" cuyas amigas son [bcde], "b" con [acde], "c" con [abde] y por último "d" y "e" con [abc]. Asumimos que las amistades son simétricas, es decir, si " a" es amiga de "b", entonces "b" es amiga de "a". Por lo tanto, aunque ocurra que "x" este en el conjunto de amigas de "y", pero "y" no este en el de "x", debemos interpretarlo como que cada una esta en el grupo de amigas de la otra.

Las salidas que corresponden a los ejemplos recien dados son las siguientes (mismo orden):
2 abdce
2 abecd
3 abgcdehf
1 xyz

La sucesión de caracteres representa la solución del problema, es decir, la ronda en la que exista la menor distancia posible entre cada amistad. El número que está delante, es la máxima distancia que hay entre dos amigas en la ronda solución. Si es que existe mas de una ronda óptima, se debe dar la que esté primera alfabéticamente.

4.2 Explicación y pseudocódigo

Para resolver el problema diseñamos un algoritmo que consiste en, dado un grupo de exploradoras, generar todas las rondas posibles, e ir almacenando aquella que hasta el momento es la "mejor" entre las ya calculadas (con "mejor" nos referimos a aquella que minimiza la suma de las distancias entre amigas).

Para armar las permutaciones, construimos la funcion *permutar*. Mediante esta funcion vamos generando todas las rondas posibles, pues cada vez que la llamamos, genera la siguiente permutacion (siguiente en orden alfabético) de la ronda, hasta que ya no hayan nuevas. Su algoritmo consiste en buscar el mayor índice k tal que a[k] $<$ a[k+1]; luego buscar el mayor índice i tal que a[k] $<$ a[i]; swapear a[k] con a[i]; y por último aplicar reverso a partir del elemento k (sin incluirlo) hasta el final de la lista. Entonces, a medida que vamos armando las distintas rondas posibles, calculamos la suma de distancias y luego la comparamos con la suma de la que tenemos almacenada. Si la suma de la nueva ronda es menor entonces nos guardamos la nueva, pues es "mejor" que la que teníamos. Caso contrario, pasamos a calcular la siguiente ronda, pues si la suma es mayor esa ronda no nos interesa, y si es igual tampoco, porque como las rondas se van calculando en orden alfabetico, entonces la que ya tenemos guardada va a estar primera teniendo en cuenta dicho orden.

El algoritmo termina una vez que se hayan calculado todas las rondas posibles. La última ronda que quedó guardada va a ser nuestra solución. Por otra parte, la máxima distancia entre dos amigas de la ronda, se calcula en simultáneo con la suma de las distancias, y siempre la almacenamos junto con la "mejor" ronda durante todo el algoritmo.

Para entender mejor la idea dejamos el algoritmo en pseudocódigo:

```
mejorRonda (dado un conjunto de exploradoras (exp) y sus amistades)
    crear int sumaMinima
    crear int maxDist
    crear Ronda rondaOptima
    ordenar alfabeticamente exp
    rondaOptima ← exp
    sumaMinima ← calcular suma de distancias de rondaOptima
    maxDist ← calcular maxima distancia entre 2 amigas en rondaOptima
    mientras (hay nueva permutacion de exp) {
        crear int nuevaSuma ← calcular suma de distancias de exp
        crear int nuevaDist ← calcular maxima distancia entre 2 amigas en exp
        si (nuevaSuma < sumaMinima) {
            sumaMinima ← nuevaSuma
            maxDist ← nuevaDist
            rondaOptima ← exp
        }
    }
    exp ← rondaOptima
    devolver maxDist
```

El cálculo de la suma de las distancias lo hacemos mediante un algoritmo iterativo que, lo que hace, es recorrer el vector (que representa a la ronda), y para cada exploradora calcula cuál es la distancia entre ella y cada una de sus amigas (mediante otro ciclo interno). Si bien al aplicar *completarAmigas* se agregan todas las amistades para todas las exploradoras, en el algoritmo se tienen en cuenta qué amistades ya fueron sumadas, por lo tanto no vuelven a sumarse (sino estaríamos calculando el doble de la suma de distancias). Se repite el procedimiento hasta que el vector se recorre completamente, y así obtenemos la suma de las distancias.
Este algoritmo es correcto pues no dejamos de lado ninguna ronda existente; siempre guardamos la mejor ronda calculada hasta el momento; y generamos las rondas en orden alfabético, por lo tanto siempre vamos a obtener la solución deseada.

4.3 Deducción de la cota de complejidad temporal

Para la resolución de este ejercicio construimos la clase Ronda en c++. La ronda se representa, en la parte privada de la clase, mediante un vector<char>, donde los char representan a las exploradoras y cómo están ubicadas en la ronda. También posee un diccionario(utilizamos <map> de la STL de c++), donde están asociadas las exploradoras a sus respectivos grupos de amigas.
La clase cuenta con el constructor por defecto (ronda vacia) y otro constructor que recibe como parámetros un vector<char> y un vector<vector<char>>; donde el primero representa el conjunto de exploradoras, y el segundo las amigas de cada una (asociadas por el orden de los vectores). Como el enunciado del ejercicio permite que mismos grupos de exploradoras sean escritos de distintas formas (por ejemplo a bc;b ac;c ab también se puede escribir como a b;b c;c ab), construimos la función *completarAmigas* que se encarga de verficar que ninguna amistad esté ausente en la lista de una exploradora, y que estén presentes la totalidad de las exploradoras, cada una con su grupo de amigas. Esta función la utilizamos en el constructor, de manera que se almacene en la

parte privada toda la información completa.
A continuación analizamos paso a paso la complejidad temporal de las funciones que implementamos. Para ello, tener en cuenta que "e" representa la cantidad de exploradoras de cada grupo y "a" la cantidad de amistades distintas.
La complejidad de *completarAmigas* es O(e.a) en el peor caso; y el algoritmo consiste en los siguientes pasos (recibe los mismos parámetros de entrada que el constructor):

- Recorrer el vector<char> que contiene a las exploradoras y por cada exploradora recorrer su grupo de amigas.
  Costo: 2a = O(a) pues son a amistades y cada una de ellas esta en el grupo de las 2 exploradoras amigas.

- Por cada amiga, verificar si está presente en el vector de exploradoras y en caso positivo (si no está presente se agrega, que toma O(1)), verificar si la exploradora i está presente en el grupo de dicha amiga(si no está presente se agrega, que toma O(1)).
  Costo: O(e) pues son dos busquedas lineales con costo O(e) en el peor caso cada una.

Finalmente la ecuación de la complejidad del algoritmo es O(e) x O(a) = O(e.a).
Otra operación pública de nuestra clase es *sumaDistancias*, que devuelve una dupla de enteros, donde la primer componente representa la suma de las distancias entre las exploradoras que son amigas, y la segunda, la mayor distancia entre dos amigas en la ronda. A continuación analizamos la complejidad de *sumaDistancias*:

- Recorrer el vector<char> que contiene a las exploradoras y por cada exploradora buscar su grupo de amigas en el diccionario y copiarlo a un vector auxiliar.
  Costo: $O(e^2)$ pues son e exploradoras; buscar en el diccionario es O(log(e))(extraído de cppreference.com) y copiar el vector es O(e) (O(e) + O(log(e)) = O(e)).

- Por cada amiga buscamos su posición en la ronda y calculamos la distancia entre ella y la exploradora a la que le corresponde el vector de amigas que estamos recorriendo.
  Costo: O(e.a) pues es buscar su posicion linealmente sobre e elementos (lo hacemos 2.a veces, pero 2.a = O(a)), y el cálculo de la distancia es O(1).

Finalmente la ecuación de la complejidad del algoritmo es $O(e^2)$ + O(e.a) = O(e.a) (la cantidad de amistades es mayor o igual que la cantidad de exploradoras, pues cada exploradora tiene al menos una amiga, entonces $O(e^2)$ = O(e.a)).

Ahora veamos cuál es la complejidad de la función *permutar*:

- Busqueda lineal del mayor indice k tal que explorers[k] < explorers[k+1].
  Costo: O(e).

- Busqueda lineal del mayor indice i tal que explorers[k] < explorers[i].
  Costo: O(e).

- Swapear explorers[k] y explorers[i].
  Costo: O(1).

- Aplicar reverso a partir del elemento k (sin incluirlo) hasta el final del vector (utilizamos *reverse* de la STL de c++).
  Costo: O(e) (extraído de cppreference.com).

Finalmente la ecuación de la complejidad del algoritmo es O(e) + O(e) + O(1) + O(e) = O(e).

Otras operaciones que implementamos en la clase Ronda son:

- *cambiarOrden*; que cambia las posiciones de las exploradoras en la ronda (lo hace en orden alfabético). Utilizamos la función *permutar*, por lo que la complejidad es O(e) en el peor caso.

- *ordenAlfabetico*; que ordena la ronda alfabéticamente. Para ello usamos la función *sort* de la STL, con complejidad O(e.log(e)) (extraido de cppreference.com).
- *amigasDe*; que devuelve un vector que contiene a las amigas de la exploradora ingresada como parámetro(O(e)).
- *rondaActual*; que devuelve un esquema de la ronda, mediante un vector (O(e)).
- *imprimirRonda* e *imprimirAmistades*; imprimen en consola las ronda y conjunto de amistades respectivamente (utilizadas principalmente para la prueba y desarollo del trabajo práctico).

Ahora analicemos la complejidad de la principal operación del ejercicio, a la cual llamamos *mejorRonda*, que modifica la Ronda de manera tal que minimice la suma de las distancias entre amigas, y devuelve la maxima distancia entre dos amigas en la ronda solución:

- Ordenamos la ronda mediante *ordenAlfabetico*.
  Costo: O(e.log(e))
- Hacemos *sumaDistancias* sobre la ronda ordenada, y guardamos los valores en una dupla de enteros.
  Costo: O(e.a)
- Copiamos la ronda actual en una nueva variable.
  Costo: O(e)
- Hacemos un ciclo que finaliza cuando hayamos generado todas las permutaciones posibles de la ronda.
  Costo: O(e!) (cantidad de iteraciones)
- Por cada iteracion del ciclo realizamos lo siguiente (en el peor de los casos):
  -*cambiarOrden* (toma O(e))
  -*sumaDistancias* (toma O(e.a))
  -Copiamos una ronda (toma O(e))
- Copiamos la mejor ronda en la parte privada de la Ronda.
  Costo: O(e)
- Copiamos un entero a una nueva variable.
  Costo: O(1)

Entonces, la complejidad de la función es:

$$\underbrace{O(e.\log(e)) + O(e.a) + O(e)}_{O(e.a)} + \underbrace{O(e!) \text{ x } \big(O(e) + O(e.a) + O(e) \big)}_{O(e!.e.a)} + \underbrace{O(e) + O(1)}_{O(e)}$$

O(e.a) + O(e!.e.a) + O(e) = O(e!.e.a)

Ahora, demostraremos que la complejidad de nuestro algoritmo cumple con la cota dada:

Aclaraciones: si bien *completarAmigas* y *mejorRonda* son dos operaciones distintas en nuestro proyecto, ambas forman parte de la solución del ejercicio, por lo tanto para demostrar que cumplimos con la cota dada en el enunciado, tenemos en cuenta las complejidades de las dos operaciones (de todas formas, la complejidad teórica no cambia).

O(e!.e.a) + O(e.a) = O(e!.e.a)

Hechas las aclaraciones, comenzamos con la demostración:

- Queremos ver que $O(e!.e.a) \subseteq O(e^e.a^2)$
- Para demostrarlo, probemos que $e!.e.a \in O(e^e.a^2)$ (pues si $f \in O(g) \Rightarrow O(f) \subseteq O(g)$)
- Por definicion de O; $e!.e.a \in O(e^e.a^2) \Leftrightarrow \exists\ n_0 > 0, c > 0\ /\ \forall\ (n > n_0)\ e!.e.a \leq c.e^e.a^2$
- Entonces, veamos que $e!.e.a \leq c.e^e.a^2\ \forall\ (n > n_0)$
- Y para probar lo anterior basta con ver que $e!.e.a \leq c.e^e\ \forall\ (n > n_0)$:
  $e! = \underbrace{1.2.3.....e}_{e\ productos} \leq \underbrace{e.e.e.....e}_{e\ productos} = e^e$

  Sabemos que la anterior desigualdad es cierta, pues todos los factores de la multiplicación del lado izquierdo son menores o iguales que los del lado derecho.
  Entonces veamos si vale lo siguiente: $e!.e.a = \underbrace{1.2.3.....e}_{e\ productos}\ .\ e.a \leq c.\ \underbrace{e.e.e.....e}_{e\ productos} = c.e^e$

  ¡Si, vale! Pues si acomodamos un poco los productos, acotamos superiormente a por e^2 (pues la maxima cantidad de amistades posibles es $\sum_{i=1}^{e-1} i = \frac{(e-1).e}{2} \leq e^2$), y tomamos c = 1.2.3 = 6 nos queda que:

  $e!.e.e^2 = 6.\ \underbrace{4.5.6.....e.e.e.e}_{e\ productos} \leq 6.\ \underbrace{e.e.e.....e}_{e\ productos} = c.e^e$

  Y con el mismo razonamiento de antes (todos los factores de la multiplicación del lado izquierdo son menores o iguales que los del lado derecho) concluimos que vale la desigualdad. Entonces $e!.e.a \in O(e^e.a^2) \Rightarrow O(e!.e.a) \subseteq O(e^e.a^2)$, y concluimos que nuestro algoritmo respeta la cota dada por el enunciado.

4.4 Experimentación

Para analizar el comportamiento de nuestro algoritmo, evaluamos su desempeño mediante varios casos de test, que nos permitieron realizar los siguientes gráficos y conclusiones. Pusimos pocos valores (de tamaño de entrada) en el gráfico, pues la complejidad y el tiempo de ejecución del algoritmo crece muy rápidamente, por lo que no podemos testearlo con casos más grandes.

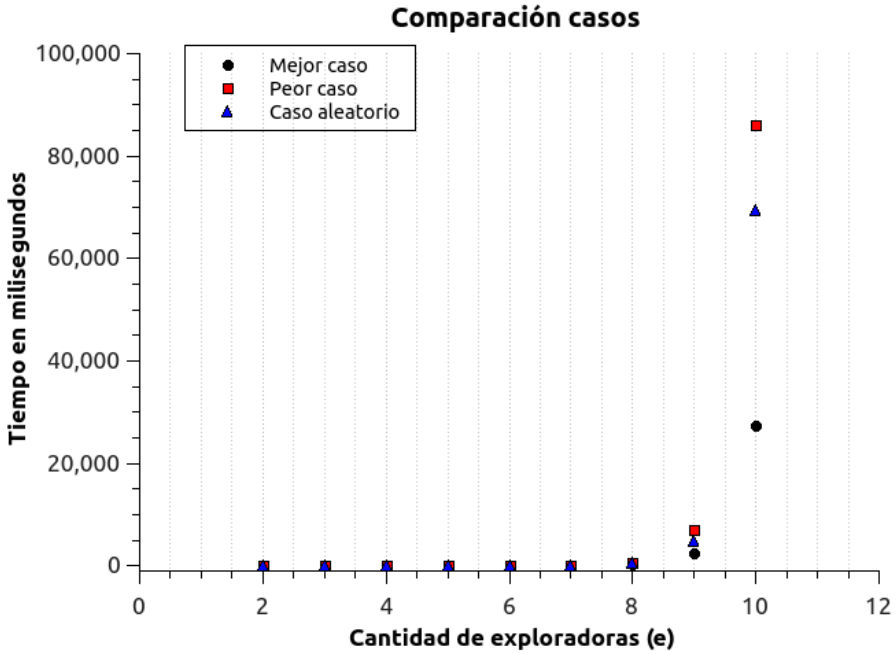

Figure 11: Comparación de los mejores y peores casos para cada tamaño de entrada. También incluimos casos aleatorios.

Los mejores casos son aquellos en los que las exploradoras tienen solo una amiga cada una, pues a la hora de calcular las distancias, hay menos amistades que sumar a la suma total. Al contrario, los peores casos son aquellos en los que todas las exploradoras son amigas con todas. Damos un ejemplo de peor y mejor caso para un ronda de 6 exploradoras:
a bcdef (mejor caso)
a bcdef;b cdef;c def;d ef;e f (peor caso)
Las instancias de mejores y peores casos las generamos manualmente (ver Mejor_PeorCaso.txt), mientras que para las de casos aleatorios hicimos un generador que utiliza la función *rand* de c++ (ver generador.cpp).
Para apreciar mejor las diferencias entre los distintos casos hicimos el siguiente gráfico, que es un "zoom" del anterior en las instancias más pequeñas.

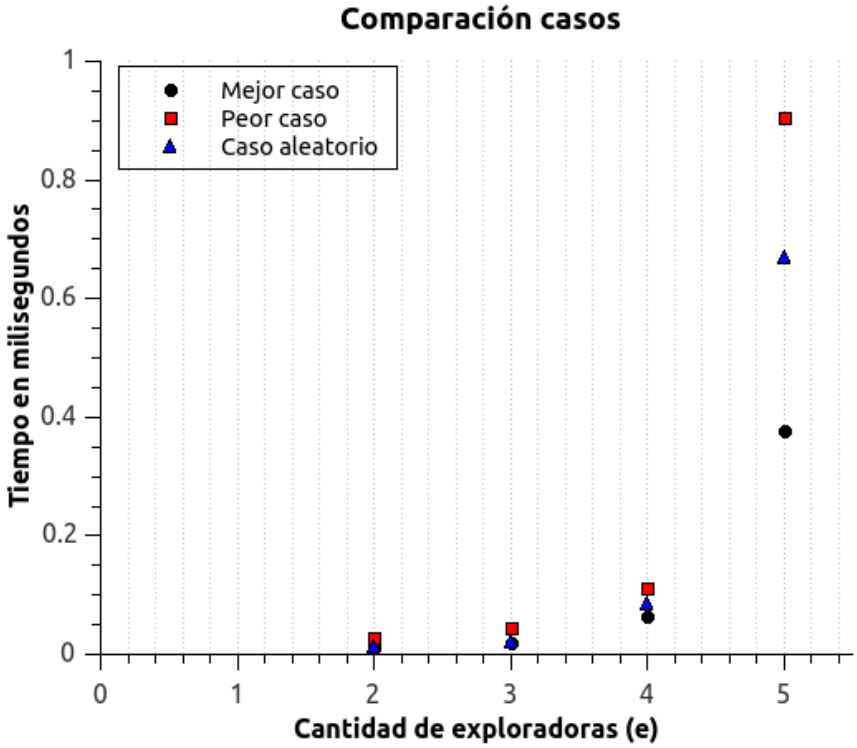

Figure 12: Comparación de los mejores y peores casos para cada tamaño de entrada. También incluimos casos aleatorios.

Si bien en instancias de entrada pequeñas los tiempos de ejecución de la computadora son muy rápidos, de todas formas se puede ver la diferencia entre los distintos casos.

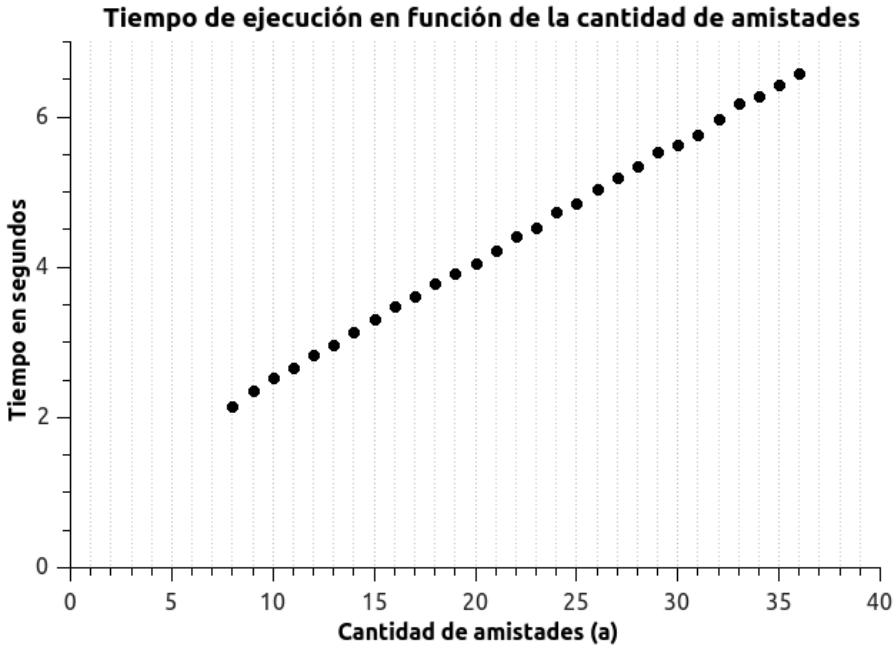

Figure 13: Tiempo de ejecución en función de la cantidad de amistades en un grupo de 9 exploradoras.

El gráfico recien expuesto nos muestra cómo aumenta el tiempo de ejecución en función de la cantidad de amistades del grupo de exploradoras. Para realizarlo, fijamos la cantidad de exploradoras en 9 (por eso los puntos comienzan en 8, pues es la mínima cantidad de amistades posibles) y fuimos agregando amistades hasta que ya no había ninguna por agregar. Este gráfico nos permite dar cuenta que la complejidad de nuestro algoritmo claramente depende de a.

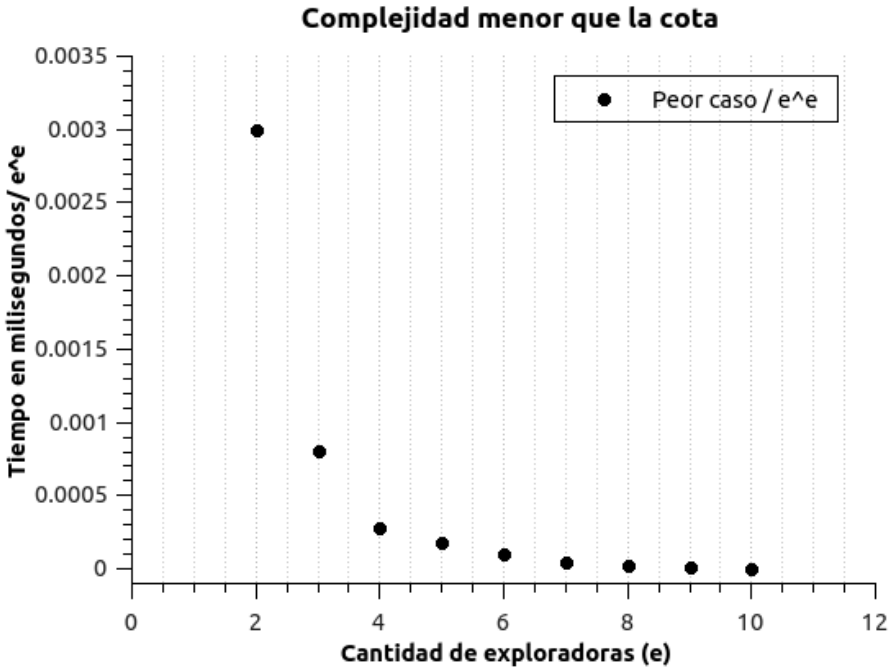

Figure 14: Complejidad de nuestro algoritmo acotada por la del enunciado.

En este gráfico dividimos los tiempos de los peores casos por e^e y como la curva tiende a 0, concluimos que la complejidad de nuestro algoritmo es menor que e^e.

Aclaración: en los archivos y carpetas de código se pueden ver muchos ejemplos para testeo.

5 Conclusión

En este trabajo realizamos 3 algoritmos distintos que resuelven los ejercicios que nos presentaron, también medimos sus complejidades y las analizamos generando tests de mejor y peor caso, y de casos random. Ilustramos estos resultados con gráficos. Para realizarlos, siempre tomamos el tiempo promedio de 100 ejecuciones del algoritmo. Además de explicarlos con ejemplos y pseudocódigos, analizamos la correctitud de estos algoritmos.