



DEPARTAMENTO  
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico I

September 2, 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

<b>1</b>	<b>Introducción</b>	<b>2</b>
<b>2</b>	<b>Problema 1: Telégrafo</b>	<b>2</b>
2.1	Idea general del problema . . . . .	2
2.2	Explicación y pseudocódigo . . . . .	3
2.3	Deducción de la cota de complejidad temporal . . . . .	4
2.4	Demostración formal . . . . .	6
2.5	Experimentaciones . . . . .	6
<b>3</b>	<b>Problema 2: A Medias</b>	<b>7</b>
3.1	Idea general del problema . . . . .	7
3.2	Explicación y pseudocódigo . . . . .	7
3.3	Deducción de la cota de complejidad temporal . . . . .	8
3.4	Demostración formal . . . . .	9
3.5	Experimentaciones . . . . .	9
<b>4</b>	<b>Problema 3: Girls Scouts</b>	<b>10</b>
4.1	Idea general del problema . . . . .	10
4.2	Explicación y pseudocódigo . . . . .	10
4.3	Deducción de la cota de complejidad temporal . . . . .	11
4.4	Demostración formal . . . . .	12
4.5	Experimentaciones . . . . .	12
<b>5</b>	<b>Código</b>	<b>13</b>

## 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 kilómetros 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 kilómetros 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
    int start ← 0
    int actual ← 0
    int aux ← 0
    mientras (la longitud del cable sea > 0 y v[actual] no sea el ultimo) {
        en el peor caso el ciclo se ejecuta n veces
        aux ← longitud del cable
        longitud del cable - (elemento próximo - elemento actual).
        si (la longitud del cable sigue ≥ 0) {
            la complejidad de la sentencia del if es de O(1)
            incrementamos en 1 resTemp
            incrementamos en 1 actual
        }
    }
    int conectadas ← resTemp
    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.
    }
    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
    }
    mientras (el elemento actual no sea el último) {
        en el peor caso el ciclo se ejecuta n veces
        si(resTemp == 0) {
            la complejidad de la sentencia del if es de O(1)
            conectadas ← 0;
            incrementamos start en 1;
            incrementamos actual en 1;
        } si no {
            decrementamos conectadas en 1;
            a longitud del cable le sumamos (v[start+1]-v[start]);
            incrementamos start en 1;
        }
        mientras(longitud del cable ≥ 0 y v[actual] no es el último) {
            en el peor caso el ciclo se ejecuta longitud del cable veces
            aux ← longitud del cable
            a longitud del cable le restamos (v[actual+1]- v[actual]);
            si (longitud del cable ≥ 0) {
                la complejidad de la sentencia del if es de O(1)
                incrementamos conectadas en 1
            }
        }
    }
}

```

```

        incrementamos actual en 1                                O(1)
    }
}
    si (longitud del cable < 0 ) {
la complejidad de la sentencia del if es de O(1)
        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)
    }
    devolvemos resTemp                                          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. Nuestro algoritmo cuenta con dos while independientes: El primer while sirve para ver hasta donde llega el cable empezando por la ciudad del kilómetro 0. Se ejecuta como mucho  $n$  veces, en ese caso (que es cuando el cable es lo suficientemente largo como para conectar todas las ciudades), el segundo while no se ejecutara pues ya se llegó al final, en caso contrario, si no se alcanza la última ciudad en el primer ciclo, este segundo ciclo desconecta la primer ciudad y se fija hasta que ciudad se llega ahora, sumándole al cable la distancia correspondiente entre la primer ciudad conectada y la segunda. Se ejecuta como mucho  $n$  veces (ese sería el peor caso, que ocurre cuando se da un cable muy corto y en cada iteración se avanza una o ninguna ciudad) dentro de él hay otro while que se ejecuta mientras que la longitud del cable sea positiva y hace lo dicho anteriormente, con la nueva longitud del cable se fija hasta que ciudad se llegaría, por ende mientras más largo el cable más ciudades se conectan y se van reduciendo las iteraciones del segundo while, ya que éste se ejecutaba hasta que se llegue a la última ciudad.

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

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 así 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:

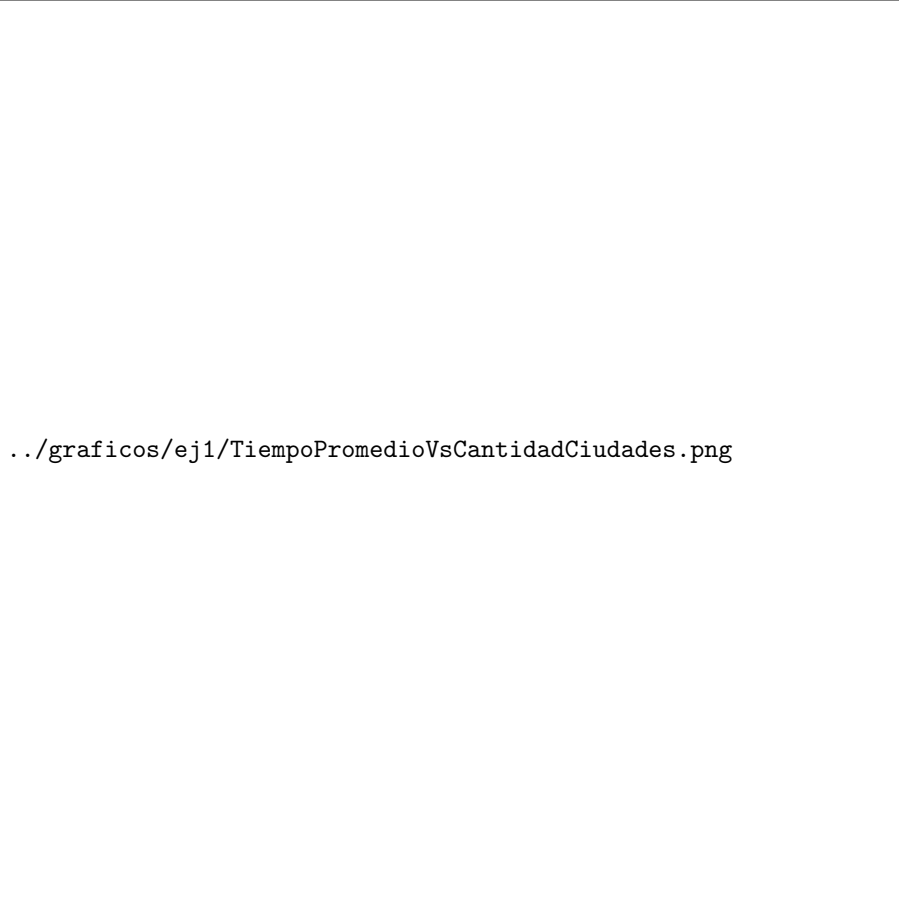
---

```
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.



../graficos/ej1/TiempoPromedioVsCantidadCiudades.png

Figure 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 ordenáramos 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:

- Si el arreglo original no contiene elementos, entonces devolvemos un vector vacío
- 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 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.

A continuación mostraremos su pseudocódigo y deduciremos su complejidad:

---

```
vector<int> medianas(vector<int> a){
    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(log(tamaño de masGrandes))
    para cada (int i = 1; i < al tamaño de a; i++){
        en el peor caso el ciclo se ejecuta tamaño de a veces
        si (el elemento i de a ≥ el elemento i-1 de res){
            la complejidad de la sentencia del if es de 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) {
la complejidad de la sentencia del if es de 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) {
la complejidad de la sentencia del if es de 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) {
la complejidad de la sentencia del if es de 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) {
la complejidad de la sentencia del if es de O(1)
        maximo ← * masChicos.rbegin()                            O(1)
        posición i de res ← maximo                                O(1)
    }
    }
    devolvemos res                                              O(1)
}

```

---

### 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 **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 Demostración formal

### 3.5 Experimentaciones

## 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 sucesión de amistades separadas por “;”. Por ejemplo, en la primer línea, el grupo de exploradoras está 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” esté en el conjunto de amigas de “y”, pero “y” no esté en el de “x”, debemos interpretarlo como que cada una está en el grupo de amigas de la otra.

Las salidas que corresponden a los ejemplos recién 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 más 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 utilizamos la función *next\_permutation*, perteneciente a la librería standard de c++. Mediante esta función vamos generando todas las rondas posibles en orden alfabético. 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 alfabético, 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:

---

```

tupla<int, vector<char>> > mejorRonda(vector<char> exploradoras, vector<vector<char>>> amigas){
    crear tupla<int, int> sumaMinima                                O(tamaño de a)
    crear vector<char> rondaOptima                                  O(tamaño de a)
    ordenar alfabeticamente exploradoras                            O(1)
    rondaOptima ← posibles                                          O(1)
    sumaMinima ← calcularSuma(rondaOptima, exp, amigas)             O(1)
    maxDist ← maximaDistancia(rondaOptima, exp, amigas)            O(1)
    mientras (hay nueva permutacion de posibles) {                  O(1)
        crear int nuevaSuma ← calcularSuma(posibles, exp, amigas)    O(1)
        si calcularSuma(posibles, exp, amigas)                        O(1)

```

El calculo 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). Se repite el procedimiento hasta que el vector se recorre completamente, y así obtenemos la suma de las distancias.

### 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 vacía) 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 verificar 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.

La complejidad de *completarAmigas* es  $O(e^3)$  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 i exploradoras.  
Costo:  $O(e)$  pues como maximo  $e = i$ .
- Por cada una de las exploradoras recorrer su grupo de amigas.  
Costo:  $O(e)$  pues como maximo son  $e - 1$  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 búsquedas lineales con costo  $O(e)$  en el peor caso cada una.

Finalmente la ecuación de la complejidad del algoritmo es  $O(e) \times O(e) \times O(e) = O(e^3)$ .

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.  
Costo:  $O(e)$  pues siempre las rondas tienen  $e$  exploradoras.
- Por cada exploradora buscar su grupo de amigas en el diccionario y copiarlo a un vector auxiliar.  
Costo:  $O(e)$  pues buscar en el diccionario es  $O(\log(e))$  (extraído de [cppreference.com](http://cppreference.com)) y copiarlo es  $O(e)$ .
- 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:  $O(e)$  pues es buscar su posición linealmente sobre  $e$  elementos, y el cálculo es  $O(1)$ .

Finalmente la ecuación de la complejidad del algoritmo es  $O(e) \times O(e) \times O(e) = O(e^3)$ .

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 *next\_permutation* de la STL de C++, por lo que la complejidad es  $O(e)$  en el peor caso (extraído de [cppreference.com](http://cppreference.com)).
- *ordenAlfabetico*; que ordena la ronda alfabéticamente. Para ello usamos la función *sort* de la STL, con complejidad  $O(e \log(e))$  (extraído de [cppreference.com](http://cppreference.com)).

Ahora analicemos la complejidad de la principal operación del ejercicio

## 4.4 Demostración formal

El procedimiento que utilizamos resuelve el problema porque, dada una ronda, al combinar las exploradoras de todas las maneras posibles

## 4.5 Experimentaciones

## 5 Código