DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

4 de septiembre de 2015

Algoritmos y Estructuras de Datos III
Segundo Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Federico Sebastián Sassone	602/13	fede.sassone@hotmail.com
Ignacio Manuel Lebrero Rial	751/13	ignaciolebrero@gmail.com
Kevin Fuksman	682/13	kfuksman@gmail.com
Damián Fixel	512/13	damianfixel@gmail.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 General

1 Ejercicio 1

1.1 Problema: Telégrafo

La comunicacion es el progreso! decididos a entrar de lleno en la nueva era el pais decidio conectar telegraficamente todas las estaciones del moderno sistema ferreo que recorre el pais en abanico con origen en la capita (el kilometro 0). Por lo escaso del presupuesto, se ha decidido ofrecer cierta cantidad de kilometros de cable a cada ramal. Pero para maximizar el impacto en epocas electorales se busca lograr conectar la mayor cantidad de ciudades con los metros asignados (sin hacer cortes en el cable).

Se busca optimizar las conexiones entre estaciones de un sistema de trenes. Para esto contamos con los siguientes datos:

El sistema está dividido en ramales.
Cada ramal comienza en la capital, que se encuentra en el kilómetro 0.
En cada ramal contamos con una cantidad de cable para conectar estaciones.
Se busca conectar la mayor cantidad de estaciones para cada ramal, sabiendo que el cable conector no puede dividirse.

Se nos pide devolver por cada ramal un entero con la cantidad máxima de estaciones conectables respetando la complejidad $\mathcal{O}(n)$ siendo n la cantidad de estaciones del sistema.

1.1.1 Ejemplos

Lineas Entrada	Linea Salida
6	3
6 8 12 15	
35	6
8 14 20 40 45 54 60 67 74 89 99	
100	4
35 87 141 163 183 252 288 314 356 387	
90	14
6 8 16 19 28 32 37 45 52 60 69 78 82	
4	0
5 13 19 26 35	
5	2
5 13 19 26 35	
5	2
7 16 19 27 33	
8	4
2 5 8 14 18	
8	3
3 6 9 15 19	

Tabla 1: Ejemplos Telégrafo

1.2 Desarrollo

Para comenzar, tengamos en cuenta que el cable no puede dividirse, es decir que las estaciones conectadas deben ser consecutivas. Sabiendo esto, encaramos el problema mediante un algoritmo goloso.

La idea es conectar todas las estaciones consecutivas posibles en un arreglo hasta que no tengamos mas cable disponible.

Comenzamos recorriendo las estaciones pasadas por parámetro en orden, partiendo de una estación central imaginaria en el kilometro 0 y conectando cada una con su siguiente.

Cuando decimos que "*conectamos*" una estación con la siguiente, lo que hacemos en realidad es restarle al cable disponible la distancia entre ambas (su distancia es a la vez la resta entre el entero que representa a la estación siguiente menos el entero que representa a la estación actual).

Si recorremos todas las estaciones con el cable disponible, devolveremos la cantidad total de estaciones mas uno (por la estación central); si esto no sucede es porque nos quedamos sin cable, es decir a la hora de conectar una estación con la siguiente, la distancia entre ambas fue mayor al cable restante.

Cuando esto ocurra, tendremos que verificar que obtuvimos la mayor cantidad de estaciones conectables de la siguiente forma:

Guardar la máxima cantidad de estaciones conectables obtenida hasta ahora y desconectar la primer estación del arreglo para asi intentar conectar la proxima estación (si es que existe) con el cable ahora sobrante después de desconectar la primera del arreglo.

Si la distancia a la proxima estación (si es que existe) es mayor al cable restante, seguiremos desconectando de a una las estaciones del arreglo (siempre desconectando la primera) para obtener más cable y reintentar conectar.
Puede ocurrirr que la distancia a la próxima estación a conectar sea mayor que la cantidad de cable disponible (inclusive luego de desconectar todas las del arreglo). Si esto sucede, comenzaremos a completar el arreglo con el mismo metodo partiendo desde esta estación.

Cuando lleguemos a la última estación, devolveremos el máximo entre la cantidad de estaciones en el arreglo de estaciones conectadas actual y la máxima cantidad que fuimos guardando cada vez que nos quedábamos sin cable.

1.3 Justificación y Complejidad

Utilizaremos dos subíndices para recorrer las estaciones del problema. Ambos comienzan seteados en 0, o sea en la posición de la estación central. Uno corresponde a la primera estación conectada en el arreglo actual de estaciones conectadas, (A), y otro a la última estación conectada o agregada al arreglo de estaciones conectadas, (B).

El algoritmo termina cuando el índice (B) llega a la última estación. Esto puede ocurrir de varias maneras y aunque el algoritmo tiene complejidad lineal, hay un mejor y peor caso:

1. O bien la cantidad de cable inicial alcanzó para recorrer todas las estaciones y el algoritmo termina luego de hacer n conexiones.

   Este es el mejor caso ya que se recorren las n estaciones y se devuelve la mayor cantidad conectable en $\mathcal{O}(n)$.

2. O se hacen uno o más guardados de la máxima cantidad de estaciones conectadas actualmente y se comienza con un nuevo arreglo de estaciones conectadas.

   Este es el peor caso; ya que si bien el problema se resuelve en tiempo lineal $\mathcal{O}(n)$, se realizan hasta el doble de operaciones que en el caso anterior.

Veamos el peor caso posible:

Imaginemos que tenemos n estaciones: n-1 estaciones en los kilómetros 1,2...n-1 y la última en el kilómetro 2n. Tambien supongamos que tenemos n kilómetros de cable. Nuestro algoritmo conectaría las estaciones 1 a n-1 en n-1 pasos. Sin embargo, cuando intente conectar n-1 con n, tardará n pasos en ver que es imposible; para luego comenzar otro arreglo en la última estación de la entrada de parámetros y terminar.

A continuación analizaremos el algoritmo:

Algoritmo 1.1 - Vista general - parametros: estaciones[], cableRestante

```
FOR i desde 1 hasta estaciones.length()  // O(n)
   distanciaActual = estaciones[i] - estaciones[i-1] //  O(1)

    IF cableRestante - distanciaActual >= 0
        .
        Algoritmo 1.2
        .
    ELSE
        .
        Algoritmo 1.3
        .
    ENDIF
ENDFOR
IF max > 0 : // O(1)
    max++  // O(1)
ENDIF
```

El bucle general recorre el arreglo linealmente, por lo que hasta el momento lleva $\mathcal{O}(n)$. Veamos que ocurre en el algoritmo 1.2:

Algoritmo 1.2

```
cableRestante -= distanciaActual  // O(1)
cantEstaciones++  // O(1)
max = maximo(max, cantEstaciones) //  O(1)
ultimaEstacionSumo[i-1] = true //  O(1)
i++;  O(1)
```

En el caso de que con el cable que tenemos podamos agregar la proxima estación, simplemente se agrega y se resta al cable. La complejidad de este caso es $\mathcal{O}(1)$ por lo que hasta el momento seguimos teniendo $\mathcal{O}(n)$

veamos el algoritmo 1.3:

Algoritmo 1.3

```
WHILE cableRestante - distanciaActual <= 0 && ultimaEstacion < i // O(n)
    IF ultimaEstacionSumo[ultimaEstacion] // O(1)
        distanciaASacar = estaciones[ultimaEstacion+1] -
        estaciones[ultimaEstacion]  // O(1)

        cableRestante += distanciaASacar // O(1)
        cantEstaciones-- // O(1)
    ENDIF
    ultimaEstacion++ // O(1)
ENDWHILE
IF ultimaEstacion == i // O(1)
    i++ //  O(1)
ENDIF
```

a simple vista parece ser $\mathcal{O}(n)$ este caso, con lo que quedaria peor caso $\mathcal{O}(n^2)$ la solución, pero observando la condición del bucle podemos ver que pide $ultimaEstacion < i$ por lo que no ira mas alla del indice en el que se encuentre el bucle exterior(algoritmo 1.1) y la variable $ultimaEstacion$ siempre avanza, por lo tanto este bucle hara como mucho n pasos en el tiempo total en que corra el algoritmo. dado que es sobre el total de tiempo de corrida del algoritmo 1.1 lo tomaremos como una suma con respecto al bucle exterior, con esto la complejidad queda:

$$\mathcal{O}(n + n)$$

$$\mathcal{O}(2n)$$

$$\mathcal{O}(n)$$

Que es lo que queriamos demostrar.

1.4 Correctitud

Probemos el algoritmo por inducción
Quiero probar P(n) = ($\forall n : entero > 0$) mi algoritmo devuelve la mayor cantidad de las n+1 estaciones consecutivas que se puedan conectar con una distancia de cable determinada o de no poder hacerlo devuelva cero.
Veamos el caso base, n=1:
resto estaciones[1] con estaciones[0] y pregunto si el cable alcanza.

1. De alcanzar (algoritmo 1.2) agrego esa estación y pasa a ser la mas larga hasta el momento, luego termina el agoritmo ya que no vuelve a cumplir la condición del algoritmo 1.1. El resultado son dos estaciones.

2. De no alcanzar (algoritmo1.3) entro en el bucle, la primera condición se cumple automáticamente por hipótesis de este caso, y como entra con i = 1, última estación será 0, por lo que la condición da true y pasa. Como el arreglo ultimaEstacion empieza en false, no se cumple la condición y paso a sumar a ultimaEstacion, como esta es igual a i sumo 1 a i y vuelvo al bucle exterior. En este momento la condición no se cumple y sale, dando como resultado cero estaciones.

Con esto queda probado que el caso base funciona, nuestra hipótesis inductiva sera que mi algoritmo funciona para $p(n)$, ahora suponiendo que vale $p(n)$ veamos si vale para $p(n+1)$.

Primero podemos observar que el arreglo contiene $n+2$ elementos, los cuales pueden ser tomados como un arreglo de $n+1$ elementos concatenado con el elemento $n+2-esimo$, sabiendo esto puedo aplicar mi hipótesis inductiva sobre el arreglo de $n+1$ elementos y comenzar la demostración en el elemento $n+2-esimo$ en el algoritmo 1.1, por hipótesis inductiva hasta el momento $\exists\ s : entero$ una posible solución maxima que conecta k estaciones, con $0 <= k <= n$ y hay una cantidad de cable dispnible m con $0 <= m <= cablemaximo$. sigamos con el algoritmo:

El elemento entra al bucle del algoritmo 1.1, pasa ya que es el ultimo elemento, calculo la distancia entre la $n+1-esima$ y $n+2-esima$ estaciones, ahora:

1. Si es menor a m, agrego esta estación, sumo uno mas y en la proxima iteracion salgo:

   (a) si el maximo local que se estaba calculando hasta la anterior iteracion mas la nueva estación es mayor a s, entonces el maximo sera el maximo local mas uno, ya que agregue la nueva estación.

   (b) si el maximo local que se estaba calculando hasta la anterior iteracion mas la nueva estación es menor o igual s, entonces el maximo sera s.

2. si no, entro en el else y comienzo a sacar estaciones del princio de la subsolucion actual:

(a) Si saco todas las estaciones que tenia hasta el momento, me quedara solo la nueva estación que agregue, con lo que significa que no me alcanza el cable, suma uno a i y en la proxima iteracion del algoritmo 1.1 no entrara en el bucle principal, devolviendo el resultado s previamente calculado.

(b) Si saco algunas de las estaciones que tenia hasta el momento, me quedara un subarreglo de estaciones y al salir del bucle, volvera a entrar en el algoritmo 1.1 entrando por la primer condición y caera en el caso 1 de esta demostracion.

Con esto vale $P(n+1)$. Luego $(\forall\ n : entero > 0)$ mi algoritmo devuelve la mayor cantidad de las n+1 estaciones consecutivas que se puedan conectar con una distancia de cable determinada o de no poder hacerlo devuelve cero.
Con esto queda demostrado que el algoritmo es correcto.

1.5 Tests

Para los Tests analizamos su mejor y peor caso.

1.5.1 Mejor Caso

El mejor caso es que nunca tenga que sacar estaciones de la subsolucion que este calculando, para esto armamos un arreglo cuya suma de distancias sea menor a la longitud maxima de cable, de esta manera siempre agrega siendo costo $\mathcal{O}(n)$.

```
public void generateAllbestCase() {
    // Calentamos la maquina si queremos medir tiempos.
        String string;
        double tiempo ;
        double[] tiempos;
        for (int i = 0; i < 100; i++) {
                tiempos = new double[5];
                string = "1";
                for (int j = 0; j < i ; j++) {
                        string = string + " 1";
                }
                for (int j = 0; j < tiempos.length; j++)
                    {
                        tiempo = System.nanoTime();
                        tp1.main.Main.testCatedraEj1Params(string,
                        Integer.MAX_VALUE);
                        tiempo = System.nanoTime() -
                            tiempo;
                        tiempos[j] = tiempo;
                }

                System.out.print(obtenerPromedio(tiempos)
                    + ";");
                }
        }
```

1.5.2 Peor Caso

El peor caso es cuando debe sacar todas las estaciones del arreglo ya que termina siendo costo $2 * \mathcal{O}(n)$, para esto armamos arreglos en donde la distancia entre todas las estaciones sea menor que la longitud total de cable excepto la ultima, cuya logitud es mayor que todo el cable, de esta manera, el algoritmo tendra que realizar 2*n operaciones.

```
public void generateAllbestCase() {
    // Calentamos la maquina si queremos medir tiempos.
        String string;
        double tiempo ;
        double[] tiempos;
        for (int i = 0; i < 100; i++) {
                tiempos = new double[5];
                string = "1";
                for (int j = 0; j < i ; j++) {
                        string = string + " 1";
                }
                for (int j = 0; j < tiempos.length; j++)
                   {
                        tiempo = System.nanoTime();
                        tp1.main.Main.testCatedraEj1Params(string,
                        Integer.MAX_VALUE);
                        tiempo = System.nanoTime() -
                           tiempo;
                        tiempos[j] = tiempo;
                }

                System.out.print(obtenerPromedio(tiempos)
                   + ";");
                }
        }
```

1.5.3 Performance

Observamos el contraste de nuestro peor y mejor caso. La variación en el peor caso está en la constante 2 que multiplica a la cantidad de operaciones realiza en el mejor caso.

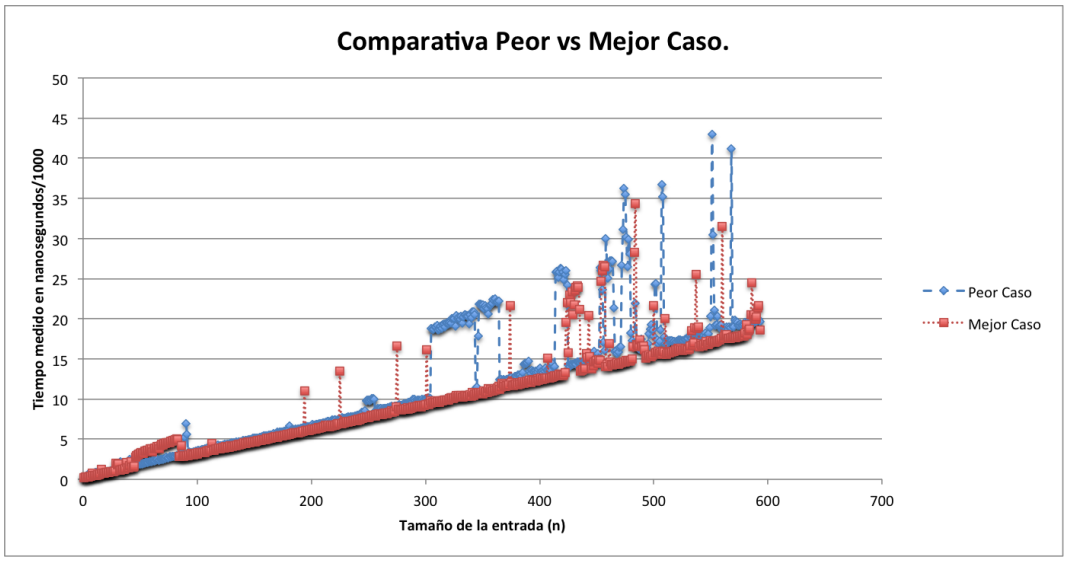

Figura 1: Comparativa Mejor vs Peor Caso

Se observa que nuestro mejor caso tiene un comportamiento lineal y mantiene ese comportamiento a medida que las entradas se hacen más numerosas.

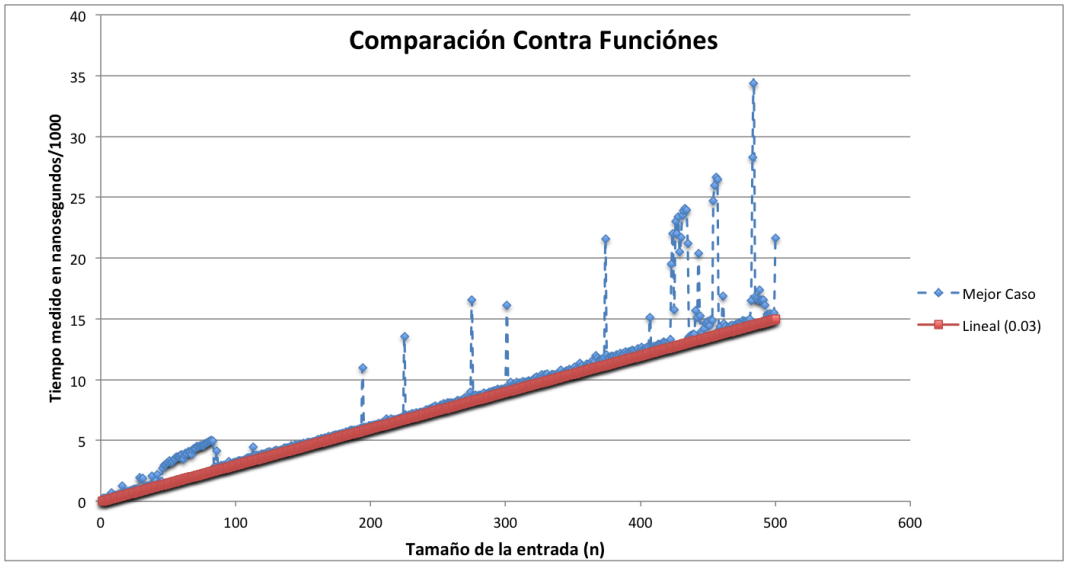

Figura 2: Comparativa Contra Funcion Lineal, Cuadratica y constante

2 Ejercicio 2

2.1 Problema: A Medias

Nos proveen la defición de mediana de un conjunto ordenado de n elementos como:
$x_{(n+1)/2}$ si n es impar
$(x_{n/2} + x_{n/2+1)}/2$ si n es par.

Se nos pide dada una entrada de n números enteros no ordenados devolver otros n números donde el $i$ésimo de ellos represente la parte entera de la función mediana aplicada a los primeros i números de la entrada, luego de ser ordenados.
Además la resolución de este problema debe obtenerse en una complejidad estrictamente menor a $\mathcal{O}(n^2)$ siendo n el número total de enteros en la entrada.
Luego, la entrada será una linea de n números, al igual que la salida.

2.1.1 Ejemplos

Linea Entrada	Linea Salida
2 3 4 1 2	2 2 3 2 2
2 7 2 8 4 9 1 6 5	2 4 2 4 4 5 4 5 5
1 87 4	1 44 4
4 0 32 6 8 10	4 2 4 5 6 7

Tabla 2: Ejemplos A Medias

2.2 Desarrollo

La complejidad intrinseca de este ejercicio esta en ir recorriendo un vector de números, e ir de alguna manera ordenando este nuevo array para obtener fácilmente la mediana, que sabemos que si se cumple la anterior propiedad es muy fácil calcularlo. Para lograr esto hicimos uso de una estructura que nos permite obtener el valor del mínimo/máximo barato y la insercion y borrado relativamente tambien. La idea es mantener siempre la idea de puntero al medio de lo que seria el array actual, por lo que decidimos usar un minHeap, un maxHeap y una variable, lo que hace que nuestro array se convierta en la siguiente estructura.

$$[1 , 2 , 3 , 4 , 5]$$

$$\text{maxheap}(1,2) \text{ ++ } 3 \text{ ++ } \text{minHeap}(4,5)$$

La idea del "++" es que si vamos extrayendo el maxheap incrementando las posiciones , agregamos la variable del medio y luego extraemos el minHeap volvemos a obtener nuestro array inicial, este vendria a ser nuestro funcion de Abstraccion que matchea nuestra estructura con un array convencional.

Teniendo esta estructura es muy fácil ir agregando nuevos elementos y mantenerla consistente es muy sencillo. La idea es la siguiente:

Si el elemento nuevo es menor a la variable (esta es el "medio"), se agrega al maxHeap, caso contrario se la agrega al minHeap. Luego de esto se llama a una funcion que readjusta la estructura de ser necesario para luego simplemente, si el vector es de tamaño par, se devuelve la suma de la variable mas el min/max del heap que tenga mas elementos, caso contrario se devuelve la variable.

Esta insercion puede generar la necesidad de ajustar la estructura, pero como veremos en la siguiente seccion esto a lo sumo necesita realizar dos pasos.

2.3 Justificación y Complejidad

El algoritmo crea las estructuras necesarias para trabajar y son las siguientes.

leftHeap $\leftarrow$ $maxHeap$ *de tama*no n
rigthHeap $\leftarrow$ $minHeap$ *de tama*no n
resultado $\leftarrow$ $intArray$ *de tama*no n
middle $\leftarrow intVariable$

Luego tenemos un for que va a hacer las llamadas correspondientes para resolver el problema.

Algoritmo 2.1

```
FOR i desde 0 hasta n
    resultado[i] <- calcularMediana(array[i],i)
ENDFOR
```

Aca ya empezamos a observar que el costo de esta funcion al menos va a tener como cota $\mathcal{O}(n)$, ya que a lo sumo vamos a recorrer 1 vez el array entero pero nos falta analizar el costo de la función calcularMediana, por lo tanto la complejidad que podemos asegurar por ahora es $\mathcal{O}(n * \mathcal{O}(calcularMediana))$

Para analizar la función anterior vamos a dividirlo en 3 partes, pero antes debemos asegurarnos que las operaciones que utilicemos en los Heaps sean de la complejidad que afirmamos que son.

Para esto fuimos a la documentación oficial de oracle, en donde citamos la siguiente frase :

"Implementation note: this implementation provides O(log(n)) time for the enqueing and dequeing methods (offer, poll, remove() and add);"

Y la misma fue extraida del siguiente link:
http://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html

Luego de esta aclaracion podemos continuar con el analisis de las funciones, la interfaz de calculateMediana es :
int calculateMediana(int number, int i).

Agregar nuevo elemento:

Esta seccion es la encargada de ingresar un nuevo elemento, para luego poder mas adelante calcular la mediana del array el cual deberiamos calcular. El pseudo-codigo es bastante sencillo, ya que tenemos un primer caso, en el cual no hay ningun elemento añadido, es decir la estructura

esta vacia, y por lo tanto la opreación simplemente consta en asignar la variable middle, caso contrario, debemos ver en que heap deberiamos insertar el elemento, sin temor a descompensar la estructura, ya que luego llamaremos a una funcion encargada de recomponer la misma de ser necesario. Es importante observar que si ingresa un elemento repetido, y este es igual al middle, por como se armaron los casos, va a ir al rigthHeap, lo que no causaria ningun tipo de inconveniente.

Algoritmo 2.2

```
IF i == 0 :
    middle <- number O(1)
ELSE :
    IF number < middle :
        leftHeap.add(number) // O(log(n))
    ELSE :
        rigthHeap.add(number) // O(log(n))
    ENDIF
ENDIF
```

Por lo tanto podemos acotar esta seccion de la función con costo : $\mathcal{O}(log(n))$

ajustar la estructura:

Esta seccion es la que se encarga de en otras palabras, hacer valer nuestro invariante de representacion, que es mantener la variable middle en lo que pasando por nuestra funcion de abstraccion es el centro del array, en caso de ser impar y sino uno de los dos elementos del "medio", en caso de ser par.

El pseudo codigo de la funcion es el siguiente:

Algoritmo 2.3

```
IF rigthHeap.size() - leftHeap.size() > 1 :
    leftHeap.add(middle)   // O(log(n))
    middle = rigthHeap.remove()  // O(log(n))
ELSE IF leftHeap.size() - rigthHeap.size() > 1 :
    rigthHeap.add(middle)  // O(log(n))
    middle = leftHeap.remove() // O(log(n))
ENDIF
```

Olvidandonos de que hay casos en los cuales, esta parte de la funcion tenga costo $\mathcal{O}(1)$ ya que no entrariamos en ningun caso, si podemos asegurar que en el caso de necesitar relaizar un adjuste, solo vamos a realizar uno, esto se traduce a ejecutar una sola parte del if. Vamos a ver que por absurdo estas condiciones no pueden cumplirse simultaneamente:

Sea a = rigthHeap.size() y b = leftHeap.size() .

Asumimos que : a - b $>$ 1 y queremos llegar a que b - a $>$ 1

Podemos hacer un pasaje de terminos en la primera por lo que nos quedaria : a $>$ b + 1

Y de la segunda sabemos que b $>$ a + 1, por lo tanto si juntamos las inecuaciones nos queda:

b $>$ a + 1 $>$ a $>$ b + 1 Abs!

Como llegamos al absurdo de que b $>$ b + 1 podemos concluir que si sucede a - b $>$ 1, no puede suceder b - a $>$ 1 . Esto lo podemos hacer sin perdida de generalidad y vamos a llegar a la misma conclusion.

Por lo tanto este analisis no lleva a poder afirmar que una cota para esta seccion es $\mathcal{O}(2 * log(n))$, pero si nos lanzamos a calcular la complejidad sin ningun analisis, la complejidad estaria acotada por $\mathcal{O}(4 * log(n))$ y como 2 y 4 son constantes llegamos a la cota final de **O(log(n))**

Devolver el resultado:

En esta ultima seccion, lo unico que debemos es identificar en que caso de array estamos, es decir par o impar, por lo que el pseudo-codigo seria el siguiente:

Algoritmo 2.4

```
IF (i+1) % 2 == 0) :
    IF (rigthHeap.size() > leftHeap.size()) :
      return ((rigthHeap.peek() + middle) /2) // O(1)
    ELSE :
       return ((leftHeap.peek() + middle) /2) // O(1)
    ENDIF
ELSE :
    return middle // O(1)
ENDIF
```

La funcion peek, lo que hace es ver el min/max elemento, y esto lo hace en $\mathcal{O}(1)$, por lo tanto la complejidad final de esta seccion es **O(1)** para cualquier caso, de entrada posible.

Sumando todas las complejidades de las tres divisiones que hicimos, nos queda que la funcion calculateMediana tiene una complejidad de $\mathcal{O}(log(n))$.

Al tener resuelta esta parte, podemos completar lo que antes no podiamos asegurar y por lo tanto la complejidad del problema es :

$\mathcal{O}(n * log(n))$

Y como $\mathcal{O}(log(n)) < \mathcal{O}(n^2)$, podemos concluir que :

$\mathcal{O}(n * log(n)) < \mathcal{O}(n^2)$

Por lo tanto nuestro algoritmo cumple con la complejidad pedida.

2.4 Correctitud

Para empezar a mostrar la correctitud de nuestro algoritmo, vamos a escribir nuestro invariante de representacion en un pseudo lenguaje.

```
1) FOREACH leftHeap as element:
    element <= middle
ENDFOREACH
2) FOREACH rigthHeap as element:
    element >= middle
ENDFOREACH
3) | rigthHeap.size() - leftHeap.size()| < 2
```

Suponiendo que este invariante de representacion, es muy fácil ver que la solucion va a hacer correcta, por la definicion de mediana. Veamos que la definicion de mediana, es igual a nuestra funcion de calcular resultado.

	Solucion Mediana	Solucion Nuestra
Caso PAR	$(x_{n/2} + x_{n+1/2})/2$	((rigthHeap.peek() + middle) /2) V ((leftHeap.peek() + middle) /2)
Caso IMPAR	$x_{(n+1)/2}$	middle

Tabla 3: Comparación de soluciones

Para no ser redundantes con la copia del codigo, vamos a explicar brevemente que en el caso par, al no existir un "medio", nuestra estructura tiene en la variable middle, uno de los 2 "medios", y con observar el tamaño de los heaps, podemos determinar cual de ellos tiene al otro "medio".

Ahora que sabemos que de nuestro invariante podemos llegar a la solucion pedida, nos queda ver que haciendo la unica operacion posible con este algotirmo que es agregar un nuevo elemento, la estructura sigue manteniendo el invariante al finalizar su ejecución.

Veamos primero nuestro case base, las estructuras vacias, y nuestra variable middle sin inicializar. Al no haber elementos en ninguno de los 2 heaps , el FOREACH se cumple trivialmente ya que no hay elementos para ver si la condicion que pedimos es valida. y el tercer punto tambien se cumple trivialmente, si no tiene elementos los tamaños son 0, por lo tanto la diferencia entre ellos es 0 y menor que 2.

Luego nos queda ver que pasa al agregar un nuevo elemento, la parte 1 y 2 del invariante de Representacion nunca deja de valer. En la seccion anterior, en la parte que titulamos "Agregar nuevo elemento"y que sería el Algoritmo 2.2 , vemos ese IF hace que los elementos nuevos se agregen respetando nuestros requerimientos, ya que seria la union de los 2 primeros puntos del invariante plasmados a las condiciones que utilizamos.

Nos falta ver que se sigue cumpliendo la tercera, y para esto nos vamos a apoyar en nuestra funcion de readjuste, que es Algoritmo 2.3. Comprobemoslo con un pequeño ejemplo pero sin perder la generalidad de en que caso estemos.

Para esto vamos a suponer que nuestro variable middle vale j y vamos a agregar un nuevo elemento llamado k, sin perdida de generalidad podriamos suponer que j $<$ k y ahora tenemos que ver varios casos en los cuales los heaps

tienen distintas cantidad de elementos. Para esto vamos a llamar a nuestro factor de balanceo como | rigthHeap.size() - leftHeap.size()| .

CASO 1, FACTOR DE BALANCEO 0:

Este caso es trivial, ya que si nuestro factor de balanceo es 0, agregando un elemento en cualquiera de los heaps, y luego recalculando nuestro factor va a hacer 1, por lo tanto menor a 2, es decir cumple el invariante de representacion.

CASO 2, FACTOR DE BALANCEO 1:

Que nuestro factor de balanceo sea 1 significa que uno de los heaps tiene un elemento mayor que el otro, por lo tanto podemos llamarlo H. Y ahora nuevamente tenemos dos casos. que el elemento j haya sido agregado a H o no.

CASO 1, j es agregado a H:

En este caso H ya tenia un elemento mas, y como le estamos agregando j, el factor de balanceo pasa a ser 2. por lo tanto lo que debemos hacer y lo que hace el algoritmo realiza es un balanceo. Por lo tanto nuestro algoritmo haria las siguientes acciones :

Vamos a llamar a nuestro otro heap T, el distinto de H.

```
T.add(middle)   // O(log(n))
middle = H.remove()  // O(log(n))
```

Por lo tanto podemos ver que antes de hacer estas dos operaciones estabamos en un factor de balanceo 2, pero agregamos un elemento en T y quitamos uno en H. Por lo tanto nuestro nuevo factor de balanceo en caso de haber hecho un rebalanceo pasa a ser de 0.

CASO 1, j no es agregado a H:

Si H tenia un elemento mas pero agregamos un nuevo elemento en el heap T, es facil ver que el factor de balanceo vuelve a tener valor 0, por lo tanto no es necesario hacer ningun readjuste, ya que el elemeto mejora el factor por si solo.

factor de balanceo estaba en 1, lo que significa que uno de los 2 heaps, tenia un elemento mas que el otro, para el proposito de esta demostración vamos a suponer que este es el rigthHeap, ya que sino no se desbaleancearia la estructura, y no podriamos observar lo que queremos demostrar.

	Tamaño leftHEAP	Tamaño rigthHEAP	valor Middle	Balanceo
Antes de agregar	3	4	5	1
Despues de agregar	3	5	5	2

Si entramos a la funcion encargada de ajustar la estructura, es fácil notar que vamos a entrar en este if:

```
IF rigthHeap.size() - leftHeap.size() > 1 :
        leftHeap.add(middle)
        middle = rigthHeap.remove()
```

	Tamaño leftHEAP	Tamaño rigthHEAP	valor Middle	Balanceo
Despues de agregar	3	5	5	2
Despues de ajustar	4	4	mínimo(rigthHeap)	0

Entonces volvamos a ver como nos queda nuestro cuadro luego de aplicar esta seccion de la funcion:

Hay que destacar que mínimo(rigthHeap) es el elemento mínimo que se extrajo en el algoritmo, y por eso su tamaño bajo en 1.

Luego vemos que la tercera parte del invariante se cumple, sin perder generalidad podemos ver que como en cada iteracion se llama a esta funcion y solo se agrega a un elemento a la vez, nunca vamos a tener una diferencia mayor a 2, por lo tanto el adjuste de mover un elemento desde el que mas tiene hasta el que menos tiene, teniendo el cuidado de hacer la transicion correspondiente para mantener el punto 1 y 2, balancea la estructura y mantiene los tres puntos de nuestro invariante.

Al haber probado que :

- Una estructura que cumple el invariante de representacion podemos ir a la solucion del problema.
- Al agregar un elemento a una instancia que cumple el invariante de representacion, nos lleva a otra instancia valida con ese elemento.

Podemos concluir que al agregar un nuevo elemento podemos llegar a una instancia valida, que luego podremos convertirla en una solucion del problema.

2.5 Tests

Para los Tests analizamos su mejor y peor caso.

2.5.1 Mejor Caso

El mejor caso es que el primer elemento se el elemento del medio y que la distribucion de la entrada sea (<middle),(>middle),(<middle) ... lo que va a generar que nunca se necesite balancear la estructura quedando cada iteracion con el costo de insertar un nuevo elemento en un heap, y eso nos cuesta: $\mathcal{O}(log(n))$.

```
public void generateAllBestCase() {
    // Calentamos la maquina si queremos medir tiempos.
                double tiempo ;
                double[] tiempos = null;
                String string;
                for (int i = 0; i < 1000; i += 2) {
                        string = "2";
                        for (int j = 0; j < i ; j++) {
                                if (j % 2 == 0){
                                        string = string
                                           + " 1";
                                } else {
```

```
                                string = string
                                    + " 3";
                        }
                    }
                    tiempos = new double[5];
                    for (int j = 0; j <
                        tiempos.length; j++) {
                            tiempo =
                                System.nanoTime();
                            tp1.main.Main.testCatedraEj2Params(string);
                            tiempo =
                                System.nanoTime() -
                                tiempo;
                            tiempos[j] = tiempo;
                    }
                    System.out.println(obtenerPromedio(tiempos)
                        + ";");
                }
        }
```

2.5.2 Peor Caso

El peor caso es cuando la distribucion de la entrada es monotona creciente/decreciente lo que generaria la necesidad de reajustar la estructura siempre luego de la segunda iteracion, y recordemos que el adjuste de la estructura nos cuenta $\mathcal{O}(2 * log(n))$ y a esto hay que sumarle el costo que tenemos siempre de agregar el elemento para luego revisar si hay que rebalancear que es $\mathcal{O}(log(n))$, lo que nos da un total de $\mathcal{O}(3*log(n))$ para cada iteracion luego de la segunda..

```
public void generateAllWorstCase() {
    // Calentamos la maquina si queremos medir tiempos.
                double tiempo ;
         double[] tiempos = null;
                String string;
                for (int i = 0; i < 1000; i += 2) {
                        string = "0";
                        tiempos = new double[5];
                        for (int j = 0 ; j < i ; j++) {
                                        string = string
                                            + " " + j;
                        }

                        for (int j = 0; j <
                            tiempos.length; j++) {
                                tiempo =
                                    System.nanoTime();
                                tp1.main.Main.testCatedraEj2Params(string);
                                tiempo =
                                    System.nanoTime() -
```

```
                        tiempo;
                    tiempos[j] = tiempo;
                }
                System.out.print(obtenerPromedio(tiempos)
                    + ";");
            }
        }
```

2.5.3 Performance

Comparación de mejor peor caso. Vemos que crecen asintóticamente igual, sólo que con distintas pendientes. La constante 3 que afecta al peor caso, surge de que en este caso se suma a la misma inserción del mejor caso, dos mas que son causa de mover el middle a un heap, y el heap que tenia mas elementos a middle.

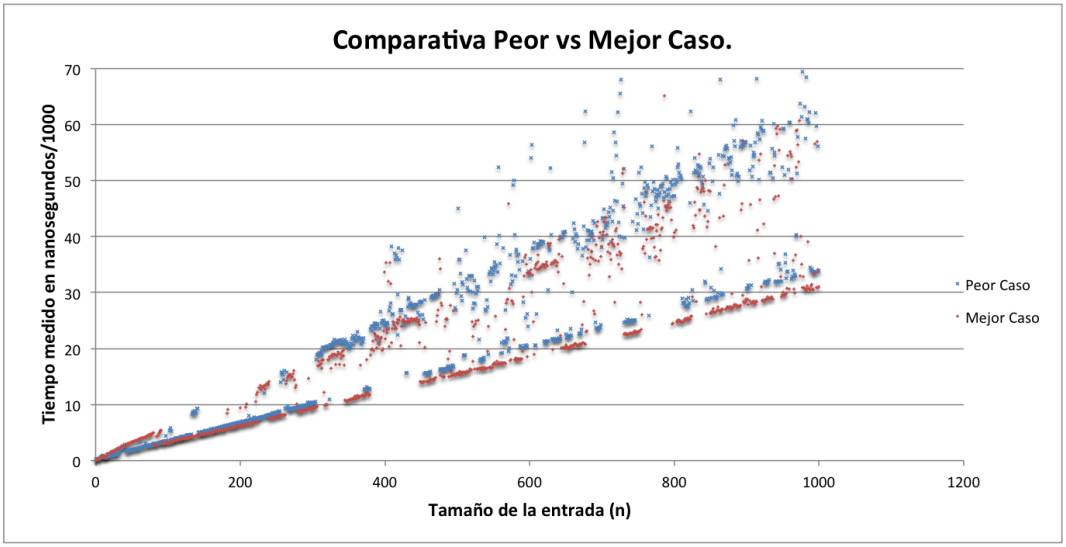

Figura 3: Comparativa Peor vs Mejor Caso

Luego podemos ver que la cota que justificamos como $\mathcal{O}(n * log(n))$ está acotada inferiormente por una función lineal a partir de un n.

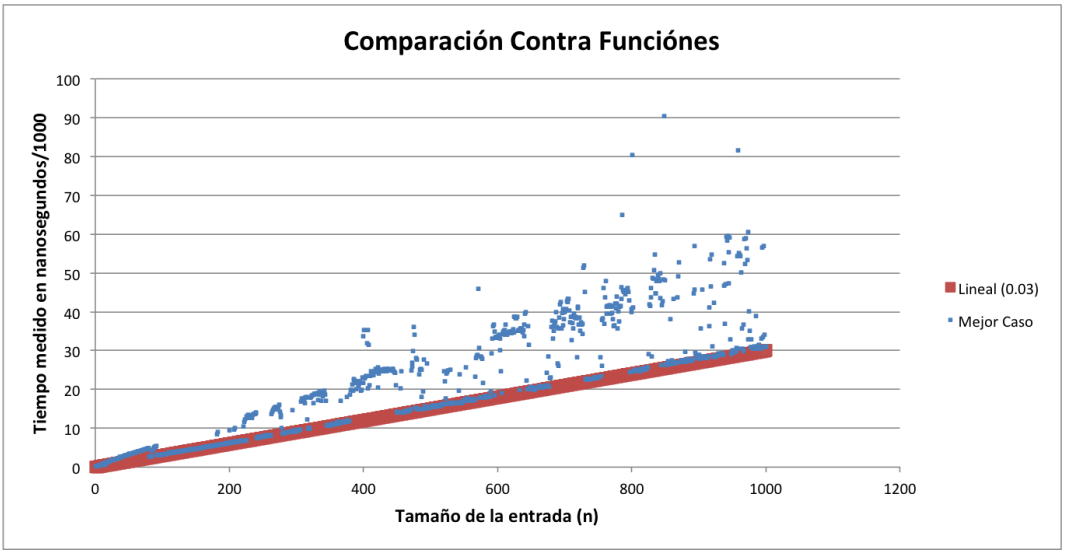

Figura 4: Comparativa Contra Funcion Lineal y Cuadratica

3 Ejercicio 3

3.1 Problema: Girls Scouts

El capitan de Las Girasoles quiere evitar los problemas en el fogoon del ano anterior cuando las exploradoras rompieron en llanto por no poder sentarse en la ronda junto a sus mejores amigas. En secreto les asigno una letra a cada nina y relevo quien se llevaba con quien.
Su idea es organizar la ronda de manera que exista la menor distancia posible entre cada amistad.

Con esto se sabe que trabajara con un conjunto de personas y un conjunto de amistades entre ellos, para esto decidimos modelar al conjunto de personas como un arreglo para determinar las posiciones que tendran cada una. Por otro lado las combinaciones de posibles lugares a saltarce simplemente seran permutaciones del arreglo.

La entrada consta de un conjunto de personas y sus respectivas amistades(las cuales tambien perteneceran al conjunto). Se quiere que, al sentarlas a todas en una ronda, se minimize la suma de distancias entre todos los pares de amistades dentro de la misma.

3.1.1 Ejemplos

Linea Entrada	Linea Salida
a bcde;b acde;c abde;d abc;e abc	2 abdce
a bcd;b ae;c ad;d ac;e b	2 abecd
a fb;b gc;d gc;f agh;e hd	3 abgcdehf
x yz	1 xyz

Tabla 4: Ejemplos Girls Scouts

3.2 Desarrollo

Dado que no es necesario generar las permutaciones que incluyan elementos repetidos, dado e un conjunto de personas, si genero permutaciones con elementos repetidos tendria e^e posibles combinaciones(ya que tendriamos e posibilidades para la primer posicion, e para la segunda y asi hasta la $e - esima$ posicion). A la hora de generar permutaciones sin repeticion de elementos tendremos e posibilidades en la primera posicion, $e - 1$ en la segunda y asi susecivamente hasta la llegar a la $e - esima$ posicion donde solo nos quedara una posibildad. De esta manera redujimos el espacio de permutaciones posibles a:

$$\underline{e}\ \underline{e-1}\ \underline{e-2}\ \underline{e-3}\ \underline{...}\ \underline{1}$$
Posibles combinaciones

$$= e * (e-1) * (e-2) * ... * 1$$

$$= \prod_{i=1}^{e} i$$

$$= e!$$

De esta manera se genera primero la permutacion inicial(con los elementos ordenados lexicograficamente, ej: a,b,c,d,e) y luego se crearan nuevas permutaciones quedandose siempre con la que cumpla las condiciones (es importante notar que el calculo de la suma de distancias se hara solo al momento que la permutacion sea generada en su totalidad, mas adelante se entrara en detallle sobre este aspecto). Como no es necesario calcular todas las permutaciones al mismo tiempo para luego compararlas, se iran creando de a una y solo se mantendra la mejor hasta el momento, evitando asi tener mayor costo espacial.

3.3 Justificación y Complejidad

Como explicamos recien, vamos a tener $e!$ permutaciones para revisar. Para cada una de estas permutaciones, lo que vamos a hacer es calcular la suma de la longitud de las exploradoras que estan relacionadas por una amistad. A este proceso lo llamaremos calcular el *costo* de la ronda.

Algoritmo 3.1 Caso base (creacion de Ronda)

```
rondaAGuardar <- copiarRonda(rondaActual); O(e)
calcularDistancias(rondaAGuardar); O(e^3)
```

$$\mathcal{O}(e + e^3)$$

$$\mathcal{O}(e^3)$$

Con esto tenemos costo cubico para crear una ronda, veamos las complejidades de calcularDistancias:

Algoritmo 3.2 calcularDistancias

```
sumaDistancias  <- 0
AmigaMasLejana  <- 0
FOR i <- 0 to ronda.length  O(e)
    mejoresAmigas <- ronda[i] O(1)
    posNina <- obtenerPos(mejoresAmigas[0]) O(e)
    j <- 2
    WHILE j < mejoresAmigas.length() O(a)
        posicionAmiga   <- obtenerPos(mejoresAmigas[j]) O(e)
      distancia      <- absoluto(posNina - posicionAmiga) O(1)
      distanciaMinima <- minimo(distancia, ronda.length - distancia) O(1)
```

```
        sumaDistancias  <- sumaDistancias + distanciaMinima; O(1)
        if (amigaMasLejana > distanciaMinima) amigaMasLejana <- distanciaMinima O (1)
          j++
      ENDWHILE
    ENDFOR
    return <- res
```

$$\sum_{i=1}^{e}(e + \sum_{i=1}^{a_i} e)$$

con e, cantidad de personas en la ronda y a_i, cantidad de amigas de la $i - esima$ persona

Como la cantidad de amigas puede ser a lo sumo e se puede acotar por

$$= \sum_{i=1}^{e}(e + \sum_{i=1}^{e} e)$$

$$= \sum_{i=1}^{e}(e + e^2)$$

$$= e * (e + e^2)$$

$$\mathcal{O}(e^2 + e^3)$$

$$\mathcal{O}(e^3)$$

Calculada la complejidad del caso base pasemos a ver el caso general:

Algoritmo 3.3 parametros: ronda, permutacion

```
mejorRondaActual <- mejorPermutacion(ronda, permutacion + 1);
FOR i <- permutacion + 1; i < ronda.length; i++  O(e - permutacion)
    swap(ronda, permutacion, i); O(1)
   mejorRondaPermutada <- mejorPermutacion(ronda, permutacion + 1);
    swap(ronda, permutacion, i); O(1)

   if (mejorRondaPermutada.sumaDistancias < mejorRondaActual.sumaDistancias)  O(1)
     mejorRondaActual = mejorRondaPermutada; O(1)
    else

      if (mejorRondaPermutada.sumaDistancias ==  mejorRondaActual.sumaDistancias
         AND (comparar(mejorRondaPermutada, mejorRondaActual)) )O(e)
            mejorRondaActual = mejorRondaPermutada; O(1)
       ENDIF
    ENDIF
ENDFOR
return <- mejorRondaActual
```

Veamoslo paso a paso, en el primer caso de entrada el bucle tendra $e-1$ repeticiones de costo $e-1$ cada una pero como forzamos una llamada antes de entrar al mismo, quedan e lladas de costo $e-1$, la segunda vez sera analoga con $e-1$ repeticiones, la tercera $e-2$ y asi sucesivamente. Si defino la funcion de recurrencia queda:

$$T(e) = \begin{cases} e * T(e-1) & si \quad e > 1 \\ 1 & si \quad e = 1 \end{cases}$$

Por el momento no tengamos en cuenta el costo e de cada bucle para comparar las rondas, veamos solo que cantidad de llamadas tendremos.

$$\begin{aligned} T(e) &= e * T(e-1) \\ &= \prod_{i=1}^{e} i \\ &= e! \end{aligned}$$

Por lo que la copmlejidad de la recursion sera $e!$ agregandole e por cada llamada nos queda $\mathcal{O}(e! * e)$, a continuacion se agregaran los casos bases suponiendo que ocurren siempore (cosa que en realidad solo pasa en las hojas del arbol del recursion).

$$\begin{aligned} &= \mathcal{O}(e! * e * e^3) \\ &= \mathcal{O}(e! * e^4) \end{aligned}$$

Ahora veamos que cumple con la complejidad pedida(por simplicidad solo escribiremos el contenido de la complejidad)

$$e^e * a^2 > e! * e^4$$

tomando el primer termino

$$e^e * a^2 > e^e$$

veamos si $e^e > e! * e^4$

$$e^{e-4} * e^4 > e! * e^4$$

$$e^{e-4} > e!$$

expandiendo ambos terminos

$$e * e * e... * e > e * (e-1) * (e-2) * ... * 5 * 4 * 3 * 2 * 1$$

Como $4 * 3 * 2 * 1$ es constante puede ser eliminado de la ecuacion

$$e * e * e... * e > e * (e-1) * (e-2) * ... * 5$$

Ahora cada termino tiene exactamente $e - 4$ elementos, basta ver que solo el primero es igual (e) y el resto son todos menores del lado derecho. De esta manera:

$$e * e... * e > (e-1) * (e-2) * ... * 5$$

Luego

$$\mathcal{O}(e^e * a^2) > \mathcal{O}(e^e) > \mathcal{O}(e! * e^4)$$

$$\mathcal{O}(e^e * a^2) > \mathcal{O}(e! * e^4)$$

Que es lo que queriamos demostrar.

3.4 Correctitud

Como dijimos Previamente existen $e!$ combinaciones posibles de elementos sin repetir y el algoritmo las recorre todas. Como se puede ver en el algoritmo 3.3 por cada entrada recorre desde la posicion *permutacion* hasta el final de la ronda swapeando el elemento actual del bucle contra la posicion de la permutacion, llama a la recursion y cuando vuelve lo swapea al lugar donde estaba previamente para probar swapear en el siguiente en la proxima iteracion, de esta manera en el primer caso de la recursion intercambiara el primer elemento del arreglo con todo el resto, dejara ese estatico y en el proximo paso recursivo intercambiara a partir del segundo elemento contra todos los siguientes y asi

hasta llegar al ultimo elemento que quedara estatico y esa sera una posible solucion. Con esto quedan generadas todas las posibles permutaciones ya que el primer elemento se prueba en e posiciones, el segundo en $e-1$ posiciones (ya que fue probado en el primer lugar cuando fue swapeado con el primero durante su iteracion) y asi sucesibamente.
Ahora veamos que calcula correctamente la suma de distancias y la distancia maxima entre amigas. El algoritmo 3.2 recorre todas las personas y todas las amigas de esas personas por lo que acarrea al maximo hasta terminar toda la busqueda con lo que efectivamente lo encuentra y la suma de distancias suma siempre la distancia minima entre dos amistades(3.2). Como recorre todas las combinaciones de amistades en la ronda podemos decir que su implementacion es correcta.
Veamos si el algoritmo acarrea la mejor solucion, en 3.3 podemos ver que luego de llamar a la recursion dentro del arreglo, se chequea contra la mejor hasta el momento (en el primer caso se compara contra la ronda sin modificaciones de esta recursion, la cual por la llamada recursiva previa al bucle fuerza a generar la primer rama de la recursion) y, de ser menor estricto en suma de distancias simplemente se queda con ese. De ser iguales en suma de distancias compara cual es menor lexicograficamente y de ser asi, cambia la mejor hasta el momento por la nueva permutacion, la cual se ira acarreando hasta que ocurra una mejor. De esta manera podemos concluir que el algoritmo resuelve el problema propuesto como queriamos ver.

3.5 Tests

En este ejericio, el mejor y peor caso es mas evidente y mas sencillo que el anterior, como en nuestro algoritmo, siempre vamos a generar todas las permutaciones posibles, podando solo los casos con elementos repetidos, la unica funcion variante va a hacer la del costo, y como esta es $\mathcal{O}(e*a)$, podemos notar que esta muy ligado a la cantidad de amistades presentes.

3.5.1 Mejor Caso

El mejor caso seria que no haya ninguna amistad, pero para tener un analisis un poco mas realista, asumimos que nuestro mejor caso es que solo exista una sola amistad, para que el algoritmo tenga un poco mas de sentido.

```
public void generateAllbestCase() {
                String string;
                double tiempo ;
                double[] tiempos;
                for (int i = 1; i < 10; i++) {
                        tiempos = new double[5];
                        string = generateFriendships(i,
                            1);
                        for (int j = 0; j <
                            tiempos.length; j++) {
                                tiempo =
                                    System.nanoTime();
                                tp1.main.Main.testCatedraEj3Params(string);
```

```
                        tiempo =
                            System.nanoTime() -
                            tiempo;
                        tiempos[j] = tiempo;
                    }
                    System.out.print(obtenerPromedio(tiempos)
                        + ";");
                }
            }
```

3.5.2 Peor Caso

El peor caso siguiendo el hilo en el que veniamos, seria que todas las exploradoras sean amigas entre si, que aunque todas las soluciones tienen el mismo costo, por la forma en que el algoritmo funciona, este es nuestro peor escenario.

```
public void generateAllWorstCase() {
                double tiempo ;
                double[] tiempos = null;
                String string;
                for (int i = 0; i < 1000; i += 2) {
                        string = "0";
                        tiempos = new double[5];
                        for (int j = 0 ; j < i ; j++) {
                                        string = string
                                            + " " + j;
                        }
                        for (int j = 0; j <
                            tiempos.length; j++) {
                                tiempo =
                                    System.nanoTime();
                                tp1.main.Main.testCatedraEj2Params(string);
                                tiempo =
                                    System.nanoTime() -
                                    tiempo;
                                tiempos[j] = tiempo;
                        }
                        System.out.print(obtenerPromedio(tiempos)
                            + ";");
                }
        }
```

3.5.3 Performance

Para la siguiente tabla de tiempos, siguiendo la definición que dimos de nuestros mejores y peores casos, pasamos a ver como se diferencian gráficamente.

Peor Caso	Mejor Caso	N
198	182	1
108	95	2
253	216	3
765	406	4
1431	1245	5
2845	2678	6
14181	10159	7
89646	48423	8
1255595	491801	9
13266693	4900001	10

Tabla 5: Comparación de mejor vs peor Caso con valores redondeados

Para el siguiente gráfico los tiempos fueron medidos en nanosegundo/1000 y luego fue aplicada la función Logaritmo en base 2, para que los datos fueran más legibles.

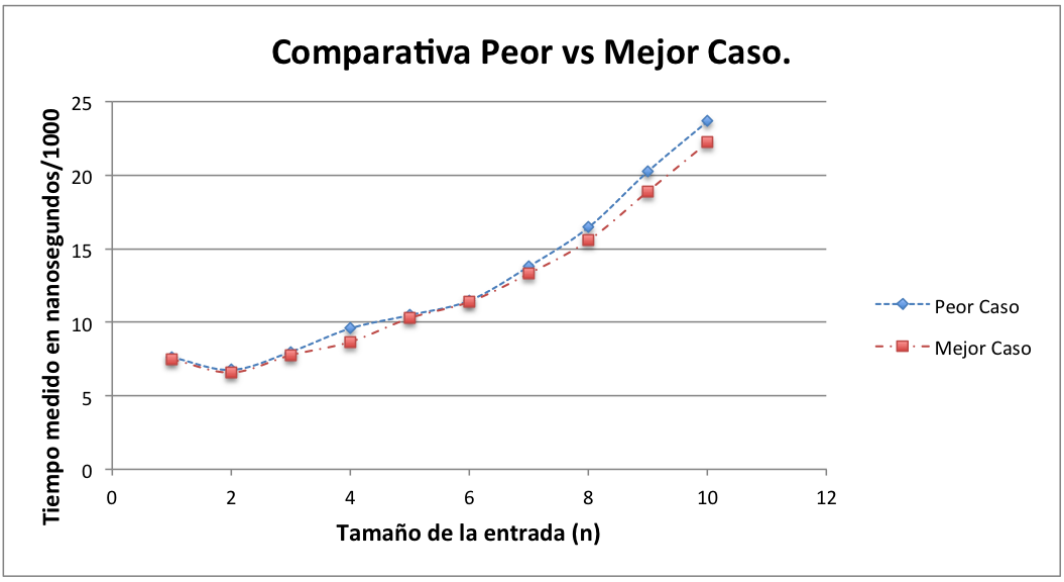

Figura 5: Comparativa Peor vs Mejor Caso

En el siguiente grafico pudimos realizar una acotacion de nuestro algoritmo. Por lo que pudimos acotarla por $\mathcal{O}(e!)$ por debajo y $\mathcal{O}(e! * (e^2))$ por arriba.

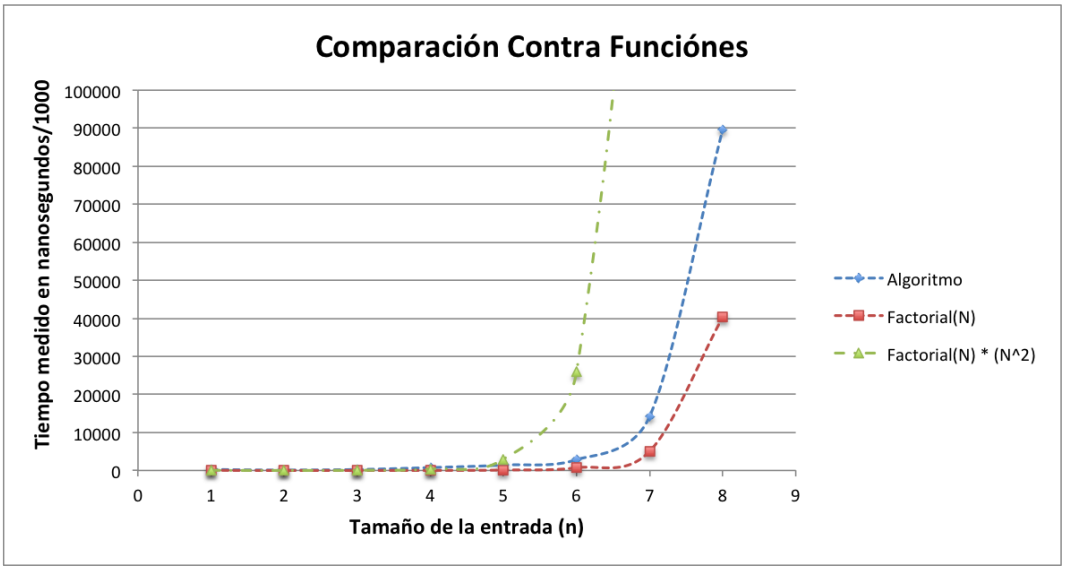

Figura 6: Comparativa Contra Funcion Lineal y Cuadratica

4 Apendice

4.1 Codigo Ejercicio1

```
package tp2.exercises;

import java.util.List;

public class Exercise1 {
        private static int[][] matriz;
        private static int[]  mejores;

        public  Exercise1(int pisos,List<Portal>
            portales) {
                pisos++;
                matriz = new int[pisos][pisos];
                mejores = new int[pisos];
                for (int i = 0; i < matriz.length; i++) {
                        for (int j = 0; j <
                            matriz.length; j++) {
                                if (j == i){
                                        matriz[i][j] = 0;
                                } else {
                                        matriz[i][j] =
                                            -2;
                                }
                        }
                }

                for (int i = 0; i < portales.size();
                    i++) {
                        matriz[(Integer)
                            portales.get(i).getDesde()][(Integer)
                            portales.get(i).getHasta()] =
                            -1;
                }
        }

        static int solve() {
                for (int i = 0; i < matriz.length -1;
                    i++) {
                        if (matriz[i][matriz.length-1]
                            == -1){
                                matriz[i][matriz.length-1]
                                    = 1;
                                mejores[i] = 1;
                        } else {
                                mejores[i] = -2;
                        }
                }
```

```
                for (int i = matriz.length -2; i >= 0;
                    i--) {
                        for (int j = matriz.length - 2;
                            j > i; j--) {
                                if (matriz[i][j] == -1
                                    && mejores[j] > 0 ){
                                        matriz[i][j] =
                                            mejores[j] +
                                            1;
                                        mejores[i] =
                                            Math.max(mejores[i],
                                            matriz[i][j]);
                                }
                        }
                }
                return mejores[0];
        }
}
```

4.2 Codigo Ejercicio2

```
package tp2.exercises;

import java.util.List;

public class Exercise2 {
        private Grafo2 grafo;
        private int piso;

        public Exercise2(int pisos, int longitud,
            List<Portal<Baldoza>> portales) {
                piso = pisos;
                Baldoza p = portales.get(0).getDesde();
                grafo = new Grafo2(portales, pisos,
                    longitud);

        }

        public int solve() {
                return grafo.solve("0,0", piso + "," +
                    piso, 0);
        }

}
```

4.3 Codigo Ejercicio3

```
package tp2.exercises;
```

```
import java.util.ArrayList;

public class Exercise3 {
        private Grafo grafo;
        private UnionFind union;

        public Exercise3(java.util.List<Pasillo>
            pasillos) {
                grafo = new Grafo();
                for (int i = 0; i < pasillos.size();
                    i++) {
                        Pasillo pasillo =
                            pasillos.get(i);
                        grafo.addVertice(pasillo.getExtremo1(),
                            pasillo.getExtremo2(),
                            pasillo.getLongitud());
                }
                union = new
                    UnionFind(grafo.getVertices().size()
                    + 1);
        }
        public int solve() {
                ArrayList<Vertice> vertices =
                    grafo.getSortedVertices();
                int i = 0;
                int peso = 0;
                while (i < vertices.size()) {
                        if
                            (union.find((Integer)vertices.get(i).getNodo1())
                            !=
                            union.find((Integer)vertices.get(i).getNodo2())){
                                union.union((Integer)vertices.get(i).getNodo1()
                                    vertices.get(i).getNodo2());
                                peso +=
                                    vertices.get(i).getPeso();
                        }
                        i++;
                }
                return -((grafo.getPeso()*-1) - peso);
        }
}
```

4.4 Informe de modificaciones

4.4.1 Ejercicio 1

Agregamos el enunciado del problema al principio del ejercicio.
Modificamos los gráficos.

4.4.2 Ejercicio 2

Se modificó la sección de correctitud de éste problema.

4.4.3 Ejercicio 3

En el código:

Dentro de la recursión ya no chequeamos que la amiga más lejana sea menor. El objeto ronda calcula la amiga más lejana y la suma de distancias al mismo tiempo con la función "$calcularDistancias$".

En el informe:

Se hicieron cambios respecto a la descripción del algoritmo para explicar cómo funciona ahora el mismo.

Se modificó la tabla de valores en dónde mostábamos la performance del mejor vs el peor caso.

Se agregó el enunciado al principio del ejercicio.

Se modificaron los gráficos.