Algoritmos y Estructura de Datos III

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 1

Grupo 1

Integrante	LU	Correo electrónico
Candioti, Alejandro	784/13	amcandio@gmail.com
Maldonado, Kevin	018/14	maldonadokevin11@gmail.com
Shaurli, Tomas	671/10	tshaurli@gmail.com
Tripodi, Guido	843/10	guido.tripodi@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Contents

1 Ejercicio 1

1.1 Descripción de problema

¡La comunicación es el progreso! Decididos a entrar de lleno en la nueva era, el país decidió conectar telegráficamente todas las estaciones del moderno sistema férreo que recorre el país en abanico con origen en la capital (el kilómetro 0). Por lo escaso del presupuesto, se ha decidido ofrecer cierta cantidad de kilómetros de cable a cada ramal. Pero para maximizar el impacto en épocas electorales se busca lograr conectar la mayor cantidad de ciudades con los metros asignados (sin hacer cortes en el cable)

Resolver cuántas ciudades se pueden conectar para cada ramal en O(n), con n la cantidad de estaciones en cada ramal, y justificar por qué el procedimiento desarrollado resuelve efectivamente el problema.

Entrada **Tp1Ej1.in**

Cada ramal ocupa 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 cosiderar el 0.

Salida **Tp1Ej1.out**

Para cada ramal de entrada, se debe indicar una línea con la cantidad de ciudades conectables encontradas.

1.2 Explicación de resolución del problema

Para resolver el problema, primero convertimos el input a una lista de enteros $Diferencias[]$, de modo que para cada i válido, $Diferencias[i]$ representa la distancia entre la estación i y la estacion $i+1$ (donde la estación 0 es la capital). El problema así descripto se reduce a conseguir la mayor cantidad de elementos consecutivos de $Diferencias[]$ tal que su suma sea menor o igual a un número fijo $L \geq 0$ (la longitud del cable).

Para ello, se mantienen dos índices sobre la lista, i, j de modo que se cumple siempre $i \leq j$. Sea n la longitud de la lista, y para cada $0 \leq i \leq j \leq n$, sea $sum(i, j)$ la suma de los elementos desde $Diferencias[i]$ hasta $Diferencias[j-1]$, ambos inclusive, con $sum(i, i) = 0$. Comenzamos con $i = j = 0$.

Siguiendo un algoritmo *greedy*, vamos a buscar en cada paso la mayor cantidad de elementos consecutivos tal que el último elemento es el j-ésimo, y la suma de los elementos es menor o iqual que L. En cada iteración, los elementos a considerar son los elementos desde el i hasta el $j-1$, ambos inclusive.

Mantenemos en una variable la cantidad actual de elementos, la suma de los elementos actuales, y la mayor cantidad de elementos conseguidos hasta el momento; los tres inicializados en 0. Vamos a iterar j sobre los números de 1 a $n-1$, y en cada paso vamos a hacer lo siguiente:

- Movemos j un lugar a la derecha. Esto hace que aumentemos en 1 la cantidad de elementos actuales, y que la suma de los elementos actuales se incremente en $Diferencias[j-1]$.
- Ahora bien, esto puede hacer que $sum(i, j) > L$. Para mantener la suma menor o igual, lo que hacemos es mover i a la derecha hasta la primera posición i' que cumple que $sum(i', j) \leq L$. Notar que como $sum(j, j) = 0$, tenemos que $i' \leq j$. Por cada posición i en que nos movemos a

la derecha, debemos restar 1 a la cantidad de elementos actuales, y restar $Diferencias[i]$ a la suma de los elementos actuales.

Al cabo de cada iteración, vale que $i \leq j$ y $sum(i, j) \leq L$. Luego, si la cantidad de elementos actuales es mayor o igual que el máximo conseguido hasta ese momento, actualizamos dicho máximo y seguimos con la siguiente iteración.

Para probar que el algoritmo resuelve el problema, probemos que para cada j, luego de la iteración j-ésima, el índice i es el menor tal que $sum(i, j) \leq L$. Para $j = 0$, tenemos $i = 0$. Supongamos inductivamente que para $0 \leq j < n - 1$ al cabo de la j-ésima iteración, el índice i es el menor tal que $sum(i, j) \leq L$. Sea i' el índice óptimo para $j + 1$. Es claro que $i' \geq i$, pues si $i' < i$ y $sum(i', j + 1) \leq L$, entonces $sum(i', j) \leq L$, contradiciendo la minimalidad de i. Luego, $i' \geq i$, y como en el algoritmo movemos al indice i hasta el primer índice i' tal que $sum(i', j+1) \leq L$, tenemos que al cabo de la iteración $j + 1$ el índice izquierdo es óptimo para el índice $j + 1$, como queríamos probar.

Resumiendo, hemos demostrado que para todo $j+1$, encontramos el mejor subarray que termina en el indice $j + 1$.

Luego, como la mayor cantidad de elementos tal que su suma sea menor o igual que L se alcanza entre dos elementos, y para cada elemento encontramos el óptimo a izquierda, el máximo que encuentra el algoritmo es efectivamente el máximo del problema, por lo que el algoritmo es correcto.

1.3 Algoritmos

Algoritmo 1 TELEGRAFO

```
1: function SOLVE(in cableSize : Integer, in stationDistances : List<Integer>)→ out res :
   Integer
2:     List<Integer> distanceDifferences ← List<Integer>(vacio)          //O(1)
3:     Integer lastStation ← 0                                           //O(1)
4:     Integer distance ← 0                                              //O(1)
5:     while i < stationDistances do                                     //O(N)
6:         distance ← stationDistances                                   //O(1)
7:         distanceDifferences.Agregar(distance - lastStation)           //O(1)
8:         lastStation ← distance                                        //O(1)
9:         i++                                                           //O(1)
10:    end while
11:    res ← getMaxRangeLength(cableSize, distanceDifferences)           //O(N)
12: end function
```

Complejidad: O(n)

Algoritmo 2 TELEGRAFO (Función Auxiliar)

```
1: function getMaxRangeLength(in cableSize : Integer, in distanceDifferences :
   List<Integer>)→ out res: Integer
2:     Integer rangeSize ← 0                                                    //O(1)
3:     Integer sum ← 0                                                          //O(1)
4:     Iterator<Integer> leftIter ← distanceDifferences.iterator()              //O(1)
5:     while i < distanceDifferences do                                         //O(N)
6:         distance ← distanceDifferences                                       //O(1)
7:         sum ← sum + distance                                                 //O(1)
8:         rangeSize++                                                          //O(1)
9:         while sum > cableSize do                                             //O(N)
10:            rangeSize−−                                                      //O(1)
11:            sum ← sum - leftIter.next()                                      //O(1)
12:        end while
13:        ret ← Math.max(ret, rangeSize)                                       //O(1)
14:        i++                                                                  //O(1)
15:    end while
16:    res ← ret                                                                O(1)
17: end function
```

Complejidad: O(n)

1.4 Análisis de complejidades

1. **Solve**

   Comenzamos por convertir el *input* provisto (una lista de N enteros tal que el lugar i-ésimo es la distancia a la capital de la estación i-ésima) por una lista $Differences[]$ tal que $Differences[i]$ es la distancia entre las estaciones i y $i+1$, que tiene una complejidad $O(N)$.

   En la implementación del algoritmo, se itera por cada elemento de la lista. En cada iteración, actualizar la suma y la cantidad de elementos actualmente considerados es $O(1)$.

   Ahora bien, como tanto el índice izquierdo (implementado con el *iterator*) siempre está a izquierda del derecho (implementado con el *foreach*), y ambos índices sólo se mueven hacia la derecha, tenemos que el índice izquierdo sólo se mueve a derecha a lo sumo N veces. Y cada vez que movemos el índice izquierdo, actualizar la cantidad de elementos y la suma actual es $O(1)$. Además, actualizar el máximo logrado hasta el momento, también es $O(1)$.

   Por lo tanto, en total realizamos $O(N)$ iteraciones para el índice derecho, más $O(N)$ iteraciones para el índice izquierdo, y cada iteración de ambos índices es $O(1)$, lo que nos da una complejidad total de $O(N)$.

   Calculemos el mejor y peor caso del algoritmo. Dado que las iteraciones del índice derecho se realizan siempre de 1 a N, el mejor caso se va a dar cuando no sea necesario mover el índice izquierdo. Esto se da, por ejemplo, cuando la suma de todas las distancias es menor que la longitud del cable (el cable alcanza para conectar todas las estaciones).

   Análogamente, es fácil ver que el peor caso se da cuando es necesario llevar al índice izquierdo hasta el final de la lista, pues en este caso la cantidad de iteraciones es máxima. Esto se puede dar, por ejemplo, cuando la longitud del cable es menor que $Differences[i]\ \forall i$.

1.5 Experimientos y conclusiones

1.5.1 Test

Por medio de los tests dados por la cátedra, desarrollamos nuestros tests, para corroborar que nuestro algoritmo era el indicado.

A continuación enunciaremos 4 tipos de casos de nuestros tests:

Rollo de cable cubre todas las estaciones

Para este tipo de testeo mostraremos a continuación un ejemplo del mismo, exponiendo su respectivo resultado.

Con un:

Rollo de Cable : 90
Lista de estaciones : 80 81 82 83 84 85 86 87 88 89 90

Obtuvimos el siguiente resultado:

Cantidad de estaciones conectadas : 11

Rollo de cable no cubre ninguna de las estaciones

Para este tipo de testeo mostraremos a continuación un ejemplo del mismo, exponiendo su respectivo resultado.

Con un:
Rollo de Cable : 60
Lista de estaciones : 80 150 220 290 360

Obtuvimos el siguiente resultado:

Cantidad de estaciones conectadas : 0

Rollo de cable con estaciones de igual o similar Kilometraje en referencia a la distancia

Para este tipo de testeo mostraremos a continuación un ejemplo del mismo, exponiendo su respectivo resultado.

Rollo de Cable : 50
Lista de estaciones : 50 60 69 70 130 190

Obtuvimos el siguiente resultado:

Cantidad de estaciones conectadas : 4

Estaciones con bastante distancia intercaladas con estaciones más cercanas

Aquí veremos, un ejemplo del conjunto de test de este tipo, exponiendo su respectivo resultado.

Rollo de Cable : 200
Listadeestaciones : 15 16 17 40 41 42 70 71 72 100 101 102 140 141 142 170 171 172 200 201 202 230 231 232

Obtuvimos el siguiente resultado:

Cantidad de estaciones conectadas : 21

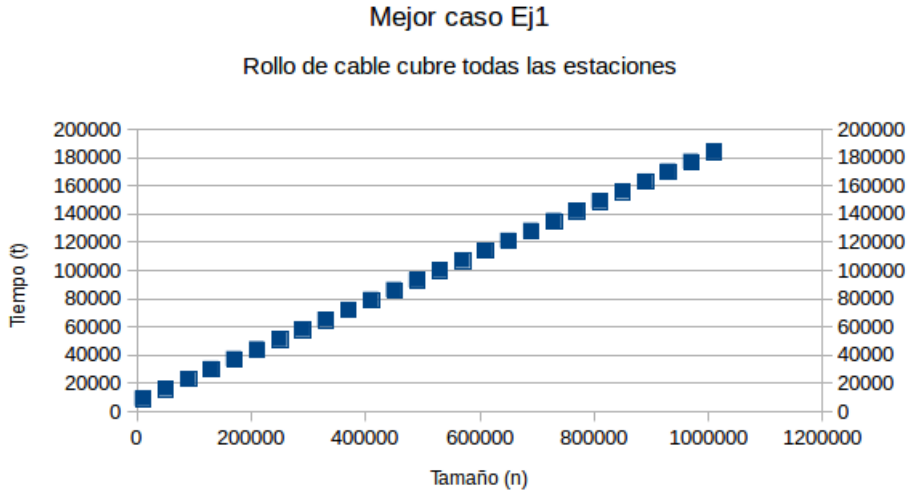
1.5.2 Performance De Algoritmo y Gráfico

Por consiguiente, mostraremos buenos y malos casos para nuestro algoritmo, y a su vez, daremos el tiempo estimado según la complejidad del algoritmo calculada anteriormente.

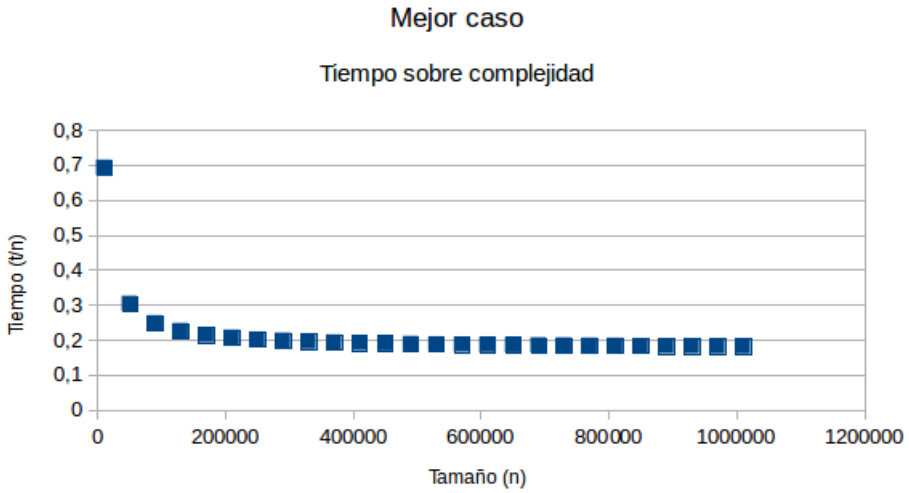
Luego de varios experimentos, pudimos llegar a la conclusión que uno de los tipos de casos que resulta más beneficioso para nuestro algoritmo es en el cual el rollo de cable llega a cubrir y conectar todas las estaciones.

Para llegar a dicha conclusión trabajamos con un total de 100 instancias y un n entre 1 y 1000000 obtuvimos que nuestro algoritmo finaliza lo solicitado demorando 184 milisegundos.

Para una mayor observacion desarrollamos el siguiente grafico con las instancias:



Si a esto lo dividimos por la complejidad propuesta obtenemos:



Para realizar esta división realizamos un promedio con el mismo input de aproximadamente 20 corridas tanto para la complejidad como para nuestro algoritmo y una vez calculado dicho promedio de ambas cosas realizamos la división para obtener resultados más consisos.

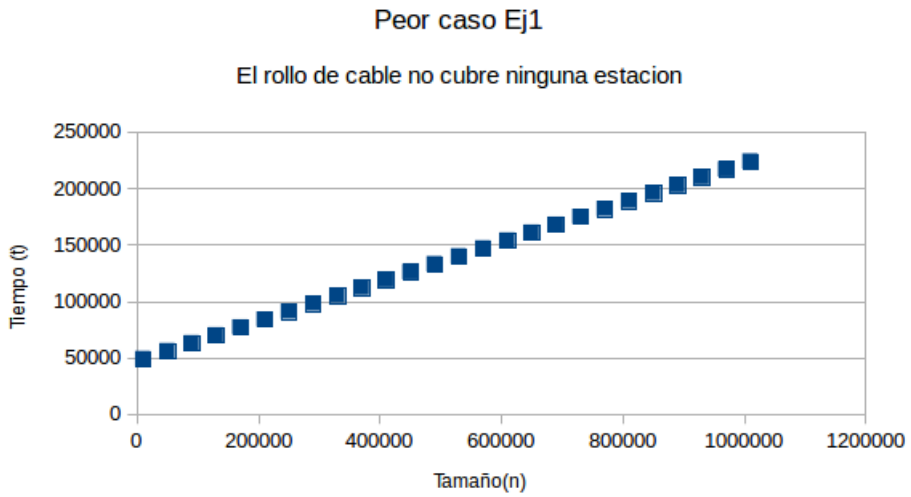
A continuación, adjuntamos una tabla con los considerados "mejor" caso que nos parecieron más relevantes

Tamaño(n)	Tiempo(t)	t/n
610000	114000	0,186
650000	121000	0,185
690000	128000	0,185
730000	135000	0,184
770000	142000	0,184
810000	149000	0,183
850000	156000	0,183
890000	163000	0,183
930000	170000	0,182
970000	177000	0,182
1010000	184000	0,182
Promedio		0.217

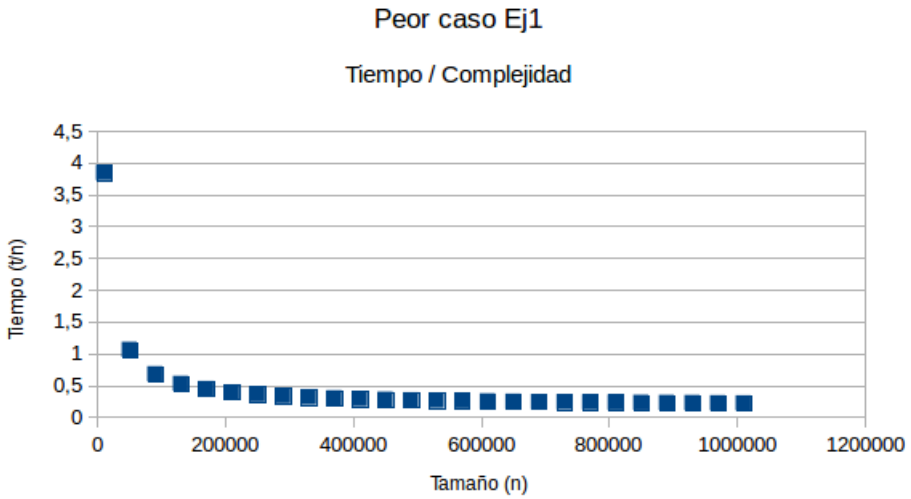
Dando un **promedio igual a 0.217**

Luego, uno de los peores casos para nuestro algoritmo es en el cual el rollo de cable no llega a cubrir ninguna distancia entre ciudades.

Para llegar a dicha conclusión trabajamos con un total de 100 instancias y un n entre 1 y 1000000 obtuvimos que nuestro algoritmo finaliza lo solicitado demorando 224 milisegundos.



Si a esto lo dividimos por la complejidad propuesta obtenemos:



Para realizar esta experimentación nos parecio acorde, realizar un promedio con el mismo input de aproximadamente 20 corridas tanto para la complejidad como para nuestro algoritmo y una vez calculado dicho promedio de ambas cosas realizamos la división para obtener resultados más relevantes.

La información de los 10 datos mas relevantes referiendonos al peor caso fueron:

Tamaño(n)	Tiempo(t)	t/n
610000	154000	0,251
650000	161000	0,247
690000	168000	0,243
730000	175000	0,239
770000	182000	0,236
810000	189000	0,233
850000	196000	0,230
890000	203000	0,227
930000	210000	0,225
970000	217000	0,223
1010000	224000	0,221
Promedio		0.469

Dando un **promedio igual a 0.469**

Aquí, podemos observar como la cota de complejidad del algoritmo y la de dicho caso tienden al mismo valor con el paso del tiempo.

2 Ejercicio 2

2.1 Descripción de problema

La mediana de un conjunto ordenado de n números se define como $x_{(n+1)/2}$ si n es impar, o como $\frac{1}{2}(x_{n/2} + x_{n/2+1})$ si n es par. Dados n números enteros en cualquier orden se deben 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.

Resolver en una complejidad estrictamente mejor que $O(n^2)$ donde n es el número total de enteros de entrada

Entrada Tp1Ej2.in y **Salida Tp1Ej2.out**
Tendrán una sucesión de enteros separados por espacios a razón de una instancia del problema por línea respectivamente.

PISTA: Mantener luego de cada entero leído de la entrada el conjunto actual dividido en dos mitades, los más chicos y los más grandes. Considerar una estructura de datos eficiente para manipular esos conjuntos.

2.2 Explicación de resolución del problema

Para la resolución de este problema se desarrolló una estructura llamada MedianCalculator que soporta dos operaciones: una de agregado de enteros y otra de calculo de la mediana de los elementos agregados. Con esta estructura la resolución del problema se vuelve directa. Simplemente consiste en una iteraración de la lista de entrada, haciendo en cada paso lo siguiente:

- Agregar el i-ésimo elemento a la estructura
- Calcular la mediana de los primeros i elementos
- Agregar la mediana calculada a una lista o vector de salida

Con esto es claro que la lista de salida cumple con los requisitos del problema, por lo que lo solo resta verificar que la estructura MedianCalculator realiza correctamente las operaciones indicadas.

Para la implementación de la estructura MedianCalculator se utilizó la estructura Heap, la cual admite las operaciones de agregado, tamaño y cálculo del menor (para algún orden dado). La estructura implementada consiste en dos Heaps: un MIN-Heap y un MAX-Heap. Es decir, el primero devuelve el menor número mientras que el segundo el mayor. A partir de ahora los llamaremos *menores* a la MAX-Heap y *mayores* a la MIN-Heap.

Nuestra estructura mantiene el siguiente invariante:

- La cantidad de elementos de *menores* es mayor o igual a la cantidad de elementos de *menores*
- La diferencia de cantidades de elementos entre *mayores* y *menores* es a lo sumo uno
- El mayor de *menores* es menor o igual al menor de *mayores*

Que puede escribirse más formalmente como:

$$0 \leq size(menores) - size(mayores) \leq 1 \wedge mayor(menores) \leq menor(mayores)$$

Para mantener este invariante a la hora del agregado miramos dos casos separados:

- *menores* y *mayores* con mismo tamaño: En este caso hacemos lo siguiente:
  - Agregamos el nuevo elemento a *mayores*: Este paso rompe el primer y tercer punto de nuestra invariante. Mientras que tampoco nos asegura que nos mantenga el segundo punto, ya que esto depende del numero agregado.
  - Sacamos el menor elemento de *mayores* y lo agregamos a *menores*: Podemos notar fácilmente que esto restablece el automáticamente el primer y tercer punto de nuestra invariante. Para verificar que esto reestablece el segundo punto basta con lo siguiente: Sea m el mayor elemento de *menores* y M el menor elemento de *mayores* (antes de agregar el nuevo elemento). Es claro que $m \leq M$. Luego si X es el elemento agregado, tenemos dos casos:
    * $X \geq M$: En este caso, sacaremos M de *mayores* y lo agregaremos en *menores*. Luego de esto, el mayor elemento de *menores* será M, mientras que el menor elemento de *mayores* será un M' y valdrá que $M \leq M'$, ya que M era el menor elemento de un conjunto que contenía a M'. Por lo tanto, se restablece el segundo punto del invariante.
    * X $<$ M: En este caso, sacaremos a X de *mayores* y lo agregaremos a *menores*. Luego de esto, el mayor elemento de *mayores* será el máximo entre X y m. Como para ambos vale que $m \leq M$ y $X \leq M$ entonces vale que $Max(m, X) \leq M$ y por lo tanto se recompone el segundo punto del invariante.

- *menores* y *mayores* de distinto tamaño: En este caso hacemos lo siguiente:
  - Agregamos el nuevo elemento a *menores*.
  - Sacamos el mayor elemento de *menores* y lo agregamos a *mayores*.

  La demostración de que esto reestablece el invariante es análoga al caso anterior.

Ahora bien, como nuestra estructura mantiene este invariante, calcular la mediana sólo se reduce a realizar las siguientes operaciones:

- Si el tamaño total es impar: Devolver el mayor elemento de *menores*.
- Si el tamaño total es par: Devolver el promedio entre el mayor elemento de *menores* y el menor elemento de *mayores*.

2.3 Algoritmos

Algoritmo 3 A MEDIAS

```
1:  function SOLVE(in list: List<Integer>)→ out res: List<Integer>
2:      if list == null then                                          //O(1)
3:          res ← Error                                               //O(1)
4:      end if
5:      List<Integer> ret ← List<Integer>(vacio)                      //O(1)
6:      Heap medianCalculator ← Heap(list.size)                       //O(log(N))
7:      Integer i ← 0                                                 //O(1)
8:      while i < list do                                             //O(N)
9:          Integer element ← list(i)                                 //O(1)
10:         medianCalculator.Agregar(element)                         //O(log(N))
11:         ret.Agregar(medianCalculator.getMedian())                 //O(log(N))
12:         i++                                                       //O(1)
13:     end while
14:     res ← ret                                                     //O(N.log(N))
15: end function
```

Complejidad: O(N.log(N))

2.4 Análisis de complejidades

Antes de calcular la complejidad del algoritmo debemos calcular la complejidad de las operaciones de Agregado y Calculo de Mediana.

Para el cálculo de la mediana, entre operaciones despreciables, realizamos como mucho dos veces la operacion "Cálculo de Menor". Como ésta es $O(1)$, el costo total del cálculo de la mediana es $O(1)$.

Para la inserción de un nuevo elemento, entre operaciones despreciables, realizamos como mucho dos inserciones a los heaps *menores* y *mayores*. Como el costo de cada inserción es $O(log(n))$, con n la cantidad de elementos presentes, la complejidad total es $O(log(n))$.

En nuestro algoritmo, efectuamos las siguientes operaciones:

- Cálculo de distancia entre puntos. (Costo $O(n)$)
- Para cada distancia: (Costo $O(n)$)
  - Inserción en MedianCalculator. (Costo $O(log(n))$)
  - Cálculo de la mediana (Costo $O(1)$)
  - Inserción en la lista de output (Usamos una ArrayList). Si bien su costo de inserción es $O(n)$, como fue construida con capacidad como párametro, sabemos que no cambia de tamaño (nunca nos excedemos de esa capacidad) y por lo tanto todas sus inserciones tienen costo $O(1)$.

Con esto nos alcanza para ver que nuestro algoritmo tiene complejidad $O(nlog(n))$ que cumple con lo pedido por el enunciado.

El mejor y pero caso de nuestro algoritmo estan fuertemente relacionados con el mejor y peor caso de la operacion de insercion en la estructura *heap*. El peor caso de insercion en un *heap* es cuando el elemento es el mayor y se lo tiene que mover desde la base hasta la punta. El mejor caso,

por el contrario, es cuando el elemento insertado permanece en la base.

El peor caso de nuestro algoritmo corresponde a cuando tenemos que insertar en las dos heaps (*menores* y *mayores*) y en ambas entramos en el peor caso. Para lograr este peor caso, generamos una lista de manera que el i-esimo elemento sea la mediana de los primeros i. Por ejemplo, con la lista $1, n, 2, n-2 \cdots$.
Por otra parte, el mejor caso de nuestro algoritmo se da cuando se inserta con el peor caso unicamente en una de las dos heaps, mientras que en la otra se inserta con el mejor caso (Es imposible evitar el peor caso de alguna de las heaps). Esto se puede lograr con el input $1, 2, 3, \cdots$.

2.5 Experimientos y conclusiones

2.5.1 Test

Por medio de los tests dados por la cátedra, desarrollamos nuestros tests, para corroborar que nuestro algoritmo era el indicado.

A continuación enunciaremos varios de nuestros tests:

Ambas ramas estan Ordenadas

Este caso se cumple cuando se recibe una lista con una sucesión con una pinta de la forma:
i ← n/2 [X_i, X_i+1]

Ambas ramas se encuentran desordenadas

Este caso se cumple cuando se recibe una lista con una sucesión con una pinta de la forma:
i ← n/2 [X_i, X_n-i+1]

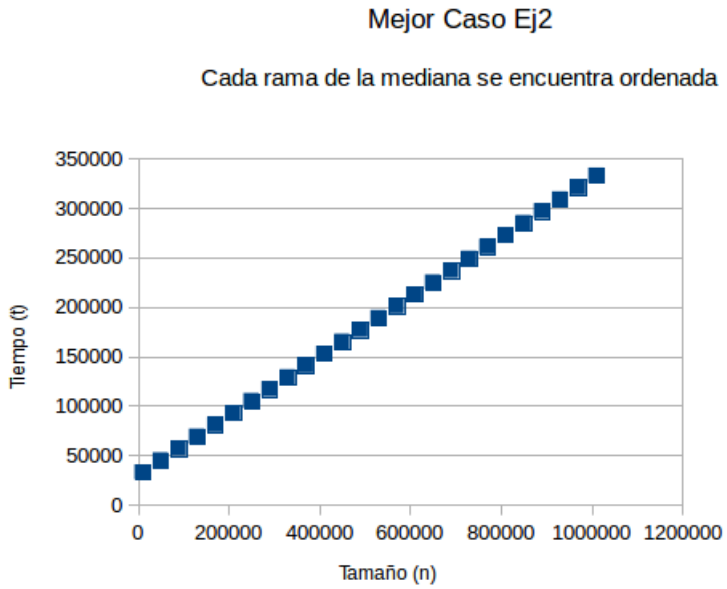
2.5.2 Performance De Algoritmo y Gráfico

Acorde a lo solicitado, mostraremos los mejores y peores casos para nuestro algoritmo, y además, daremos el tiempo estimado según la complejidad del algoritmo calculada anteriormente.

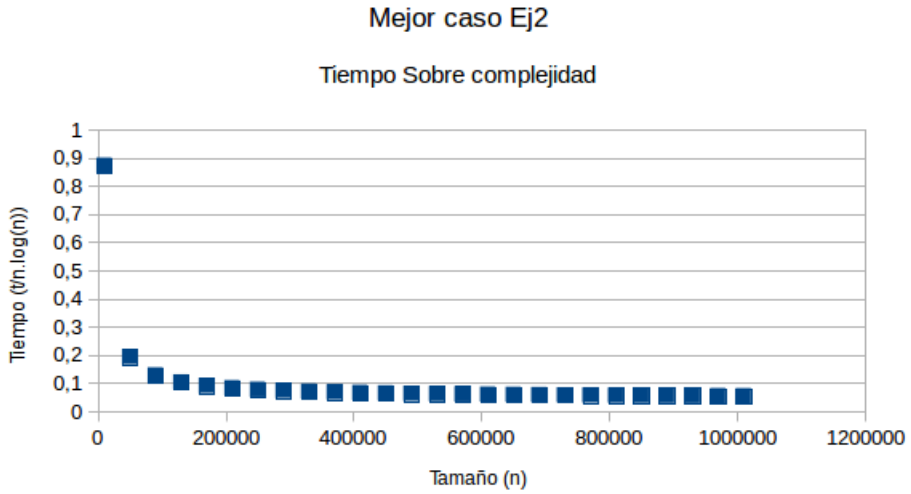
Luego de chequear varios instancias, pudimos llegar a la conclusión que uno de los tipos de casos que resulta más beneficioso para nuestro algoritmo es en el cual ambas ramas de la mediana se encuentran ya ordenadas

Para llegar a dicha conclusión trabajamos con un total de 100 instancias y un n entre 1 y 1010000 obtuvimos que nuestro algoritmo finaliza lo solicitado demorando 333 milisegundos.

Para una mayor observacion desarrollamos el siguiente grafico con las instancias:



Y dividiendo por la complejidad de nuestro algoritmo llegamos a:



Para realizar esta experimentación nos parecio prudente, realizar un promedio con el mismo input (n entre 1 y 1001000) de aproximadamente 20 corridas tanto para la complejidad como para nuestro algoritmo y una vez calculado dicho promedio de ambas cosas realizamos la división para obtener resultados más relevantes.

Se puede observar, como luego de realizar la división por la complejidad cuando el n aumenta el valor tiende a 0.

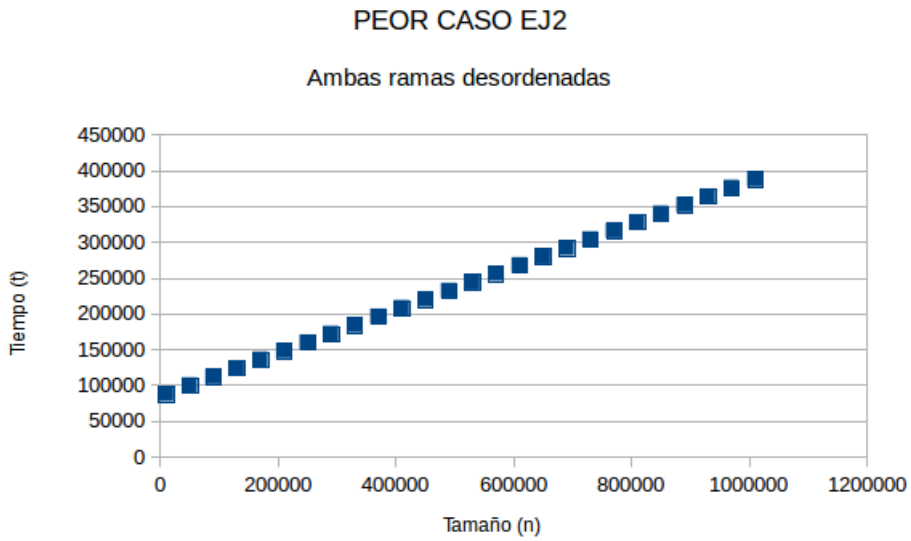
A continuación mostraremos una tabla con los 10 datos de medición mas relevantes y mostraremos un promedio de la totalidad de las instancias probadas.

Tamaño(n)	Tiempo(t)	$t/n.log(n)$
730000	249000	0,058
770000	261000	0,058
810000	273000	0,057
850000	285000	0,057
890000	297000	0,056
930000	309000	0,056
970000	321000	0,055
1010000	333000	0,055
Promedio		0,104

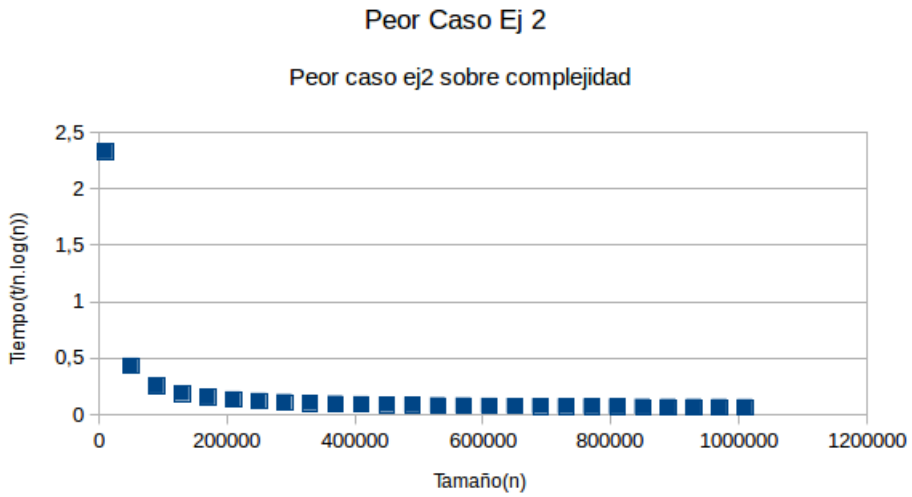
Promedio final de todas las instancias: 0,104

Verificando el peor caso, llegamos a la conclusión que el tipo de caso en el que resulta menos beneficioso trabajar con nuestro algoritmo será cuando ambas ramas se encuentran desordenadas.

Realizando experimentos con un total de 100 instancias con un n variando desde 1 hasta 1010000 obtuvimos que nuestro algoritmo, resuelve lo mencionado en 385 milisegundos, a continuación mostraremos un gráfico que ejemplifica lo enunciado.



Y dividiendo por la complejidad propuesta llegamos a:



Para realizar esta experimentación nos parecio acorde, realizar un promedio con el mismo input de aproximadamente 20 corridas tanto para la complejidad como para nuestro algoritmo y una vez calculado dicho promedio de ambas cosas realizamos la división para obtener resultados más relevantes.

Se puede observar que a pesar de tardar varios milisegundos este tipo de caso, al dividir por vuestra complejidad es propenso a tender a 0 quedando comparativamente por encima del mejor caso.

A continuación mostraremos una tabla de valores de lass ultimas 10 instancias y mostraremos el promedio total conseguido .

Tamaño(n)	Tiempo(t)	$t/n.log(n)$
610000	268000	0,076
650000	280000	0,074
690000	292000	0,073
730000	304000	0,071
770000	316000	0,070
810000	328000	0,069
850000	340000	0,068
890000	352000	0,067
930000	364000	0,066
970000	376000	0,065
1010000	388000	0,064
Promedio		0,195

Promedio total conseguido: 0,195

Se puede observar como el peor caso presenta un promedio mayor que el mejor caso, concluyendo lo que enunciamos inicialmente.

3 Ejercicio 3

3.1 Descripción de problema

El capitan de Las Girasoles quiere evitar los problemas en el fogón del año anterior, cuando las pequeñas exploradoras rompieron en llanto por no poder sentarse en la ronda junto a sus mejores amigas. En secreto les asignó una letra a cada niña y relevó quién se llevaba con quién.

Su idea es organizar la ronda de manera que exista la menor distancia posible entre cada amistad (Sí, mi querido computador minimizando la suma de las distancias entre todos los pares de amigas).
Resolver en una complejidad estrictamente mejor que $O(e^e a^2)$. Donde e es la cantidad de exploradoras en cada grupo, y a la cantidad de amistades.

Entrada Tp1Ej3.in
El archivo contiene una línea por cada grupo de exploradoras.
Cada línea se compone de una sucesión de amistades separadas por ; de la forma amistad[;amistad]
Cada amistad es un par x xs donde x es una letra y xs una cadena de letras.

Salida Tp1Ej3.out
Para cada grupo de exploradoras, se debe indicar un nro. entero correpondiente a la distancia máxima lograda, un espacio y la cadena de letras resultante. En caso de múltiples soluciones, dar la que este primera alfabéticamente.

3.2 Explicación de resolución del problema

La solución planteada utiliza la técnica algorítmica de *backtracking*. La idea es recorrer todas las configuraciones posibles manteniendo la mejor solución encontrada hasta el momento. Se representa a la ronda como un array donde cada posición equivale a un lugar para sentarse, y su valor en dicha posición indica quién esta sentado allí. Para evaluar todas las posibles configuraciones, vamos sentando a las niñas desde la posición 1 a la posición |ronda| en el array. Para esto se implementa una función que dado el índice que indica la siguiente posición a ocupar, prueba sentar a cada chica posible (es decir, dentro de las que no estan sentadas aún) en la posición vacía y llama recursivamente aumentando en uno el índice (Cuando finaliza cada llamada recursiva, se "vuelve hacia atras" y se coloca a la niña que estaba originalmente en esa posición). Cuando el array (o la ronda) está llena, ya se puede calcular la suma de las distancias y por lo tanto se puede decidir si es la mejor solución encontrada (hasta ese momento).

Se puede demostrar por inducción en la cantidad de personas que esta función evalúa todas las posibilidades.

- Caso base: Si hay solo una chica, la función calcula la suma de distancias para la única configuración posible.
- Paso inductivo: Si la cantidad de chicas es $n > 1$. Como la función coloca todos las posibles en la primera posicioń, y para las restantes llama recursivamente ($n-1$ chicas). Por nuestra hipótesis inductiva, sabemos que la función para $n-1$ chicas evalúa todas las posibles. Por lo tanto, como la función para n se llama recursivamente para $n-1$ para todas las particiones posibles de [chica de la primer posicion] - [chicas de las demas posiciones], podemos afirmar por inducción que f evalúa todas las permutaciones de las n chicas.

Como nuestro algoritmo pasa por todas las permutaciones posibles y estas son $n!$, su complejidad tiene un factor $n!$. Calcular la suma de distancias de todas las amigas tiene costo $n^2 * log(n)$, ya que para cada pareja de chicas se hace lo siguiente:

- Se fija si son amigas (Costo $log(n)$ por la implementacion de set)
- Si son amigas se calcula la distancia. (Costo $O(1)$)

Podas: Se puede mejorar el algoritmo propuesto anteriormente efectuando algunas podas. Es decir, si pensamos al *backtracking* como un árbol en el nos vamos moviendo por diversas posiciones intermedias (nodos internos) hasta llegar a las finales (hojas). Si a partir de un nodo intermedio podemos afirmar que todas las soluciones que derivan del mismo no son la óptima, podemos rechazarlas y pasar a analizar otras ramas, ahorrando asi un importante costo en cálculos. A esta técnica se la conoce como "podas" (o *backtrack*) en un *backtracking*.

- Poda de permutaciones circulares: Como estamos representando a la ronda en una array, existen permutaciones equivalentes a la misma ronda. Por ejemplo, ABC es equivalente a BCA. Este problema se soluciona dejando fija a la niña de la primera posición. Ahora bien, como el problema nos pide devolver la configuración con el menor orden lexicográfico, fijamos a la niña de menor orden lexicográfico en la primera posición del array. De esta manera todas las soluciones generadas serán la de menor orden lexicográfico entre sus equivalentes.
- Poda de distancia parcial: Si en vez de calcular las distancias al final las calculamos a medida que agregamos a alguien nuevo, podemos calcular esta suma parcial con la mejor suma obtenida hasta ahora. Es decir, si en un "nodo interno" ya nos pasamos de la mejor suma obtenida hasta ese momento, no tiene sentido seguir investigando esta rama del árbol (ya que esta suma sólo puede aumentar, y por lo tanto nunca será la mejor). ¿Suma complejidad calcular la suma de distancias parcial cada vez que sentamos a una niña? Si bien esto parece agregar complejidad, lo cierto es que cada vez que se sienta una niña sólo se actualiza la suma actual con las distancias de las amistades de la niña agregada, es decir, no se vuelven a mirar amistadas ya miradas previamente. Por lo tanto, para cada hoja del arbol del *backtracking*, solo habremos mirado una vez cada pareja de niñas, que no es más veces que en el caso sin la poda. De hecho, sin esta poda hay distancias que se calculan repetidas veces. Si calculamos las distancias al final, (A y B amigos) para las permutaciones $ABCD$ y $ABDC$ estaremos calculando varias veces la distancia entre A y B, mientras que con la poda esa distancia ya la tenemos calculada.
- Poda de amistades restantes: Resulta evidente que si ninguna de las niñas que resta sentar tiene amigas, no tiene sentido probar todas las permutaciones, ya que el orden en que se sienten no modificará la suma total. Por lo tanto, esta poda consiste en llevar la cuenta de cuántas amistades ya se calculó la distancia (i.e. ambas niñas de la amistad están sentadas) y si ya no quedan amistades para calcular ordena a las siguientes niñas. Esto último es porque de todas las permutaciones queremos la de menor orden lexicográfico.

3.3 Algoritmos

A continuación se detalla el pseudo-código de la parte principal del algoritmo incluyendo las podas:

Algoritmo 4 Calculate

global: remainingFriendships, girls, currentSum, currentMin, bestRound

```
1:  function CALCULATE(in currentIdx: Integer)
2:      if remainingFriendships = 0 then                                          //O(1)
3:          sittingGirls ← copy(girls)                                            //O(n)
4:          subList ← sittingGirls[currentIdx, size(girls)]                       //O(1)
5:          sort(subList)                  //O((k * log(k)) where k = size(girls) - currentIdx
6:
7:          if currentSum < currentMin then                                       //O(1)
8:              bestRound ← sittingGirls                                          //O(1)
9:          else
10:             bestRound ← firstLexicographically(bestRound, sittingGirls)       //O(n)
11:         end if
12:         currentMin ← currentSum                                               //O(1)
13:     else
14:         for swapIdx from currentIdx to size(girls) do
15:             swap(girls, currentIdx, swapIdx)                                  //O(1)
16:             partialDistance ← getPartialDistance(currentIdx)                  //O(n log(n))
17:             currentSum ← currentSum + partialDistance                         //O(1)
18:             if currentSum ≤ currentMin then                                   //O(1)
19:                 calculate(currentIdx + 1)
20:             end if
21:             swap(girls, currentIdx, swapIdx)                                  //O(1)
22:             currentSum ← currentSum - partialDistance                         //O(1)
23:         end for
24:     end if
25: end function
```

Complejidad: $O(n!.n^2.log(n))$

Algoritmo 5 getMaxDistance

```
1:  function GETPARTIALDISTANCE(in currentIdx: Integer, out res: Integer)
2:      sum ← 0                                                                   //O(1)
3:      count ← 0                                                                 //O(1)
4:      for girlIdx from 0 to currentIdx do                                       //O(n)
5:          friendship ← Friendship(girls[girlIdx], girls[currentIdx])            //O(1)
6:          if contains(friendships, friendship) then                             //O(log(n))
7:              idxDiff ← rightIdx - leftIdx                                      //O(1)
8:              distance ← min(idxDiff, size(girls) - idxDiff)                    //O(1)
9:              sum ← sum + distance                                              //O(1)
10:             count ← count + 1                                                 //O(1)
11:         end if
12:     end for
13: end function
```

Complejidad: $O(nlog(n))$

3.4 Análisis de complejidades

El algoritmo recorre todas las permutaciones (que en total son $n!$), y para cad auna de ellas calcula la suma de las distancias, lo que nos agrega un factor $n^2 * log(n)$ en nuestra complejidad.

Concluimos así que la complejidad total es $O(n!.n^2.log(n))$.

Se puede ver que la complejidad de nuestro algoritmo es estrictamente menor que $O(n^n a^2)$, ya que

$$O(n!.n^2.log(n)) \subset O(n^n) \subset O(n^n a^2)$$

Por lo tanto, nuestro algoritmo cumple con lo pedido por el enunciado.

Nuestra implementación final incluye las podas anteriormente explicadas, y como las mismas no empeoran la complejidad, sigue respetando la complejidad requerida por la cátedra.

El peor caso de nuestro algoritmo es cuando todas las niñas son amigas entre sí. En este, no es posible aprovechar las podas aplicadas y por lo tanto se debe evaluar cada permutación.

El mejor caso en cambio, se da cuando ninguna niña es amiga de otra. En este se da que en la primer llamada se ingresa en el caso de la poda por cantidad de amistades. Luego, la complejidad de este caso es $O(n^2)$, que es el costo de busar el par de amigas con mayor distancia.