DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Algoritmos y Estructuras de Datos III

Segundo Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Bouzón, María Belén	128/13	belenbouzon@hotmail.com
Jiménez, Paula	655/10	puly05@gmail.com
Montepagano, Pablo	205/12	pablo@montepagano.com.ar
Rey, Maximiliano	037/13	rey.maximiliano@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. Problema 1: Telégrafo

1.1. Descripción de la problemática

En el primer ejercicio se nos presenta un contexto en el que se pretende conectar con un cable de longitud arbitraria la mayor cantidad de estaciones férreas consecutivas pertenecientes a un ramal dado. Siéndonos provistas las distancias entre las sucesivas estaciones y la longitud del cable, la propuesta de este problema es hallar el valor de dicha cantidad.

Supongamos, por ejemplo, que nos fuera dado un ramal con cada estación 0i definida en el kilometro $j * 2 + 2^(j - 1)$ de manera tal que en un ramal de cinco estaciones las mismas se encontraran en los kilómetros 0, 3, 6, 10, 16.

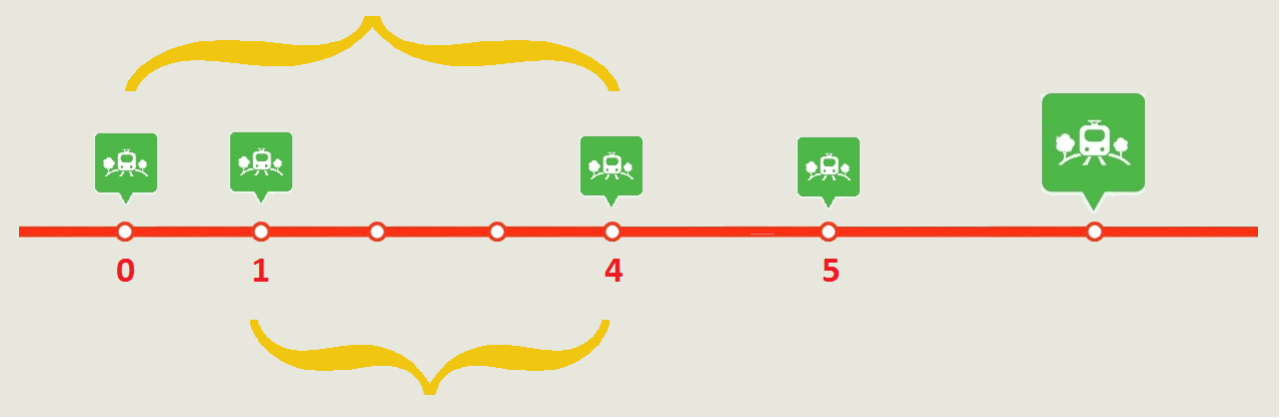
Si contáramos con un cable de 5 kilómetros, la respuesta correcta debería ser ”2”, puesto que con dicha longitud se podrían unir a lo sumo dos estaciones (pudiendo ser las mismas la primera y la segunda, la segunda y la tercera o la tercera y la cuarta).

Si el mismo cable pudiera extenderse hasta los 6 km, la respuesta arrojada debería ser ”3”, representando a la combinación de las ciudades 1, 2 y 3 (cuya distancias suman exactamente 6km).

En cambio, si midiera 1 km, la respuesta debería ser ”1”, puesto que no puede conectarse más de una ciudad - sea cual fuere - con un cable de dicha longitud.

1.2. Resolución propuesta y justificación

Una primera aproximación a la resolución hubiese podido ser calcular, partiendo de cada ciudad, cuántas ciudades se pueden comunicar a partir de ella con el cable ofrecido. El problema de esta solución es que al ser “ ciego“ a los resultados parciales que cada iteración ofrece, potencialmente pueden llegar a realizarse los mismos cálculos en reiteradas oportunidades. Por ejemplo, supongamos la siguiente distribución de las estaciones:



Un algoritmo planteado a partir de esta idea, partiría de la estación Nro. 0 para concluir que se pueden conectar todas desde aquella hasta la Nro 4, para luego continuar revisando una a una las estaciones que pueden unirse partiendo de la Nro 1, cuando resulta evidente que si la distancia entre las estaciones 0 y 4 es menor o igual a la longitud del cable entonces necesariamente la distancia entre la estación siguiente (la Nro. 1) y la Nro. 4 también será menor a la longitud del cable. De no considerar esta premisa se desprende la redundancia en los cálculos que impactan en la complejidad de algoritmo.

Dicho esto, la propuesta de resolución escogida versa sobre la idea de utilizar información relevada

en estaciones anteriores para aminorar la cantidad de cómputos, como se esboza en el siguiente pseudocódigo:

if Existe alguna estación en el ramal y la longitud del cable es positiva **then**
  while La estación a partir de la cuál estoy midiendo no es la última y tapoco lo es la más lejana alcanzada por dicha estación **do**
    Calcular la cantidad de ciudades para las cuales se sabe que alcanza el cable
    while Alcanza el cable para unir otra estación **do**
      Incrementar la cantidad de estaciones que pudieron unirse
      Guardar cuál fue la estación más lejana alcanzada hasta el momento a partir de la que se mide
    end while
    Avanzar ciudad a partir de la cuál se mide
    Actualizar la máxima cantidad de ciudades unidas hasta el momento.
  end while
  if Se lograron conectar dos o más ciudades **then**
    Devolver la maxima cantidad de estaciones que se consiguió unir.
  else
    Devolver 0
  end if
end if

Este procedimiento resuelve adecuadamente el problema propuesto porque calcula para cada ciudad inicial la máxima cantidad de ciudades que pueden ser recorridas y lo realiza con una cota de complejidad de O(n), como se detallará en el próximo apartado.

Además, como se afirma que un programa es correcto si resuelve lo pedido en una cantidad finita de pasos y sabiendo que para cualquier entrada de tamaño N el algoritmo termina a lo sumo en $m * N$ interaciones del loop principal (con m un entero positivo) [1], podemos afirmar que nuestra solución es correcta respecto del problema presentado.

[1] Nótese que el bucle principal llega a su fin cuando se alcanza como ciudad base a la última estación de un ramal de N estaciones que se recorre linealmente

1.3. Análisis de la complejidad

Sea N la cantidad total de estaciones

Entonces se afirma que en el peor caso cada estación va a ser estación basal en alguna instanciación del ciclo. En otras palabras, el ciclo se repetirá a lo sumo N veces.

Dada una instanciación del ciclo en la estación i-ésima ($i! = 0$), existen dos posibilidades:

- La estación $i - 1$ llegó a conectar a la i-ésima y a k estaciones posteriores a ella.
- El cable no fue lo suficientemente largo para lograrlo.

Si nos encontráramos en el segundo caso, el algoritmo partiría de la ciudad i-ésima recorriendo a lo sumo $N - i - 1$ estaciones posteriores para analizar hasta cuál es la más lejana alcanzable.

Si el caso fuese el primero, entonces la ciudad posterior (i) contrastará la longitud del cable y las distancias entre kilometros para - a lo sumo - N-i-1-k estaciones (en lugar de hacerlo para N-i-1, como hubiese debido hacerlo bajo otra implementación).

Esto implica que para cada estación i, podrían llegar a realizarse a lo sumo N-i-1 comparaciones pero todas aquellas que se efectúen serán descontadas de la cantidad de cálculos que potencialmente haría la ciudad siguiente, forzando a que cada estación sea consultada acerca de su proximidad una única vez y permitiendo de esta manera que la ejecución del algoritmo se complete en tiempo lineal respecto del tamaño del input.

1.4. Código fuente

A continuación se presenta la parte más relevante del código fuente, seleccionada en función de la pertinencia con la resolución del problema, dejando de lado aspectos necesarios pero secundarios, como ser el manejo de la lectura del input, la escritura del output, la medición de los tiempos, etc.

```
public class Main
{
  public static void main(String[] args) throws Exception
  {

if (longitudCable != 0 && ciudades.size() > 1)
{
    Ramal ramal = new Ramal(ciudades);
    int indiceCiudadMasLejanaAlcanzada    = 0;
    int cantMaximaCiudadesUnidas          = 1;
    int ciudadesUnidas;

    while(ramal.HayCiudadesMasLejanas() &&
 !ramal.EsLaUltimaCiudad(indiceCiudadMasLejanaAlcanzada))
    {
        if (indiceCiudadMasLejanaAlcanzada > ramal.indiceCiudadActual)
            ciudadesUnidas =  indiceCiudadMasLejanaAlcanzada -
 ramal.indiceCiudadActual + 1;
        else
            ciudadesUnidas = 1;

        while(ramal.AlcanzaParaUnirUnaCiudadMas(ciudadesUnidas, longitudCable))
        {
            ciudadesUnidas ++;
            indiceCiudadMasLejanaAlcanzada ++;
        }

        ramal.AvanzarCiudadBase();

        if (ciudadesUnidas > cantMaximaCiudadesUnidas)
            cantMaximaCiudadesUnidas = ciudadesUnidas;

    }

    res = cantMaximaCiudadesUnidas == 1? 0: cantMaximaCiudadesUnidas;

}
else
{
    res = 0;
}
  }
}
```

```
public class Ramal {

    public Ramal(){}

public ArrayList<Ciudad> ciudades;

public Ramal (ArrayList<Ciudad> ciudades)
{
this.ciudades = ciudades;
this.indiceCiudadActual = 0;
}

public int indiceCiudadActual;

public boolean AlcanzaParaUnirUnaCiudadMas(int ciudadesUnidas, int longitudDeCable)
{
return !EsLaUltimaCiudad(indiceCiudadActual) &&
!EsLaUltimaCiudad(indiceCiudadActual + ciudadesUnidas - 1) &&
DistanciaEntreCiudades(indiceCiudadActual, indiceCiudadActual + ciudadesUnidas)
<= longitudDeCable;
}

public void AvanzarCiudadBase()
{
if (!EsLaUltimaCiudad(indiceCiudadActual))
indiceCiudadActual++;
}

public boolean HayCiudadesMasLejanas()
{
return (indiceCiudadActual < ciudades.size() - 1);
}

public int DistanciaEntreCiudades(int indiceCiudadA, int indiceCiudadB)
{
return this.ciudades.get(indiceCiudadB).GetKilometro() -
 this.ciudades.get(indiceCiudadA).GetKilometro();
}

public boolean EsLaUltimaCiudad(int indiceCiudad)
{
return indiceCiudad == this.ciudades.size() - 1;
}
}
```

1.5. Experimentación

Para contrastar nuestra hipótesis acerca de la complejidad temporal (presuntamente lineal) del algoritmo propuesto, diseñamos una clase denominada ClassGenerator1 encargada de generar ramales pseudoaleatorios con una cantidad de estaciones secuencialmente creciente.
La misma clase ejecuta sobre cada instancia una cantidad paramétrica de veces [2] el algoritmo que resuelve el problema propuesto. Por cada iteración sobre una misma instancia se guardan los tiempos medidos y al finalizar todas las ejecuciones sobre cada una de las instancias, se almacena en un archivo denominado $resultados.out$ el procesamiento de los datos relevados, de manera tal que persista para cada caso el promedio de los tiempos calculados sin considerar los outliers.
De esta forma procuramos estandarizar en la medida de lo posible los resultados, de manera que no resulten considerablemente afectados por limitaciones del hardware y del software empleado durante el análisis.

[2] 350 veces, para el presente trabajo

1.5.1. Constrastación Empírica de la complejidad

Habiendo ejecutado nuestro algoritmo para 200 casos distintos donde vale que para todo i, caso$_i$ tiene un input de tamaño 10*i, obtuvimos resultados que expresamos visualmente en los gráficos 1 y 2.

Mientras que el primero muestra los tiempos absolutos conseguidos para cada entrada (luego del promediado y la eliminación de outliers), el segundo presenta las razones entre los tiempos medidos para cada instancia y el tamaño de sus correspondientes parámetros de entrada.

En la figura 2 puede verse cómo al dividir cada tiempo computado por el tamaño del input los resultados tienden a una constante.

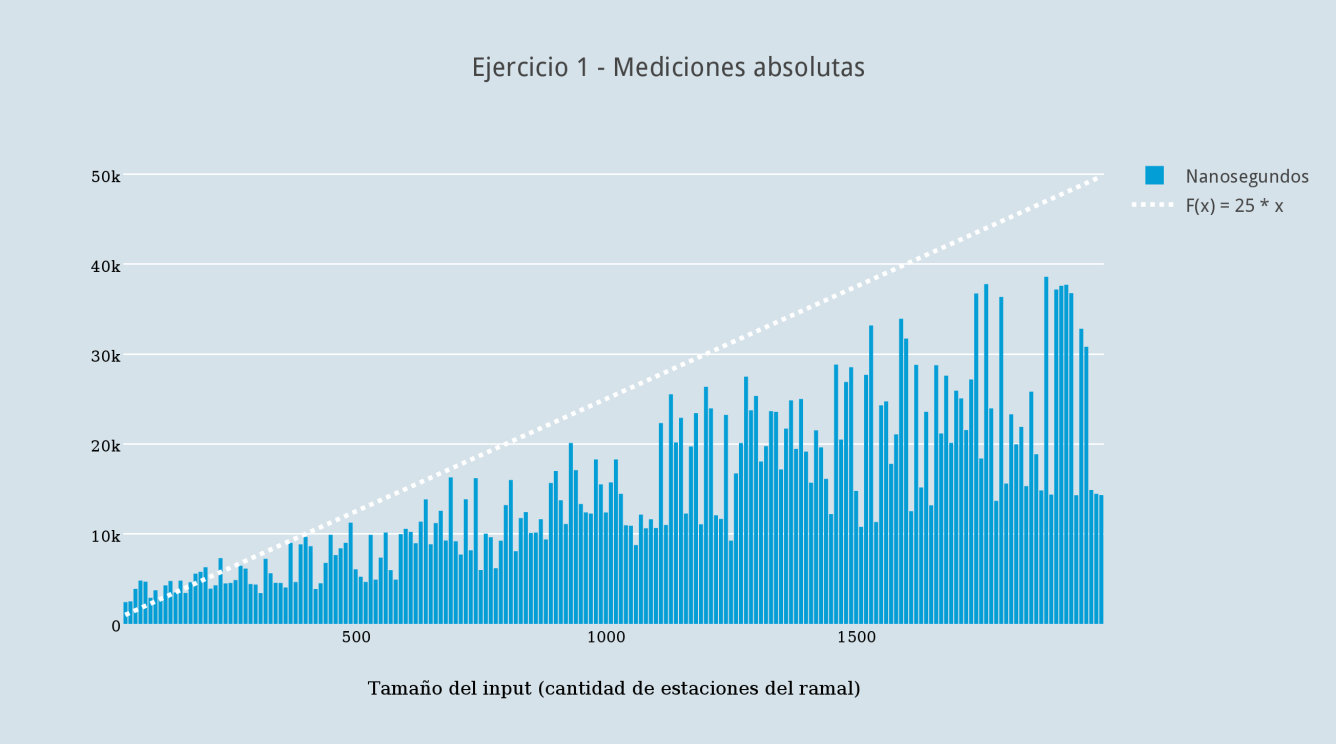


Figura 1: Mediciones absolutas sobre el tamaño del input

En la figura 1 graficamos además - a modo de referencia - la función F(x) = 25*x, con el fin de dejar en evidencia la cota superior lineal que de forma teórica dedujimos para el algoritmo que se analiza en esta sección.

Es conveniente aclarar que, si bien se puede vislumbrar que existe una correlación directa entre el tamaño del parámetro de entrada y los tiempos asociados a los peores casos, la gráfica presentada no es en sí la base que nos hace afirmar que la complejidad asintótica del algoritmo es lineal, si no que esta aseveración se apoya fuertemente en el estudio formal del algoritmo, esbozado en la sección *Análisis de la complejidad*. Es decir que consideramos a los resultados obtenidos tan sólo como un sustento empírico más en favor de nuestros supuestos. En cambio, si los mismos hubieran sido inconsistentes con el análisis teórico entonces hubiesen resultado de gran relevancia a la hora de poner en tela de juicio nuestras hipótesis iniciales.

Ahora bien, no todas las instancias aumentan su tiempo de cómputo a medida que el tamaño de la entrada se acrecienta. Esto puede deberse a varios factores, de entre los cuales consideramos necesario mencionar al menos dos:

1. Diferencia entre instancias de casos

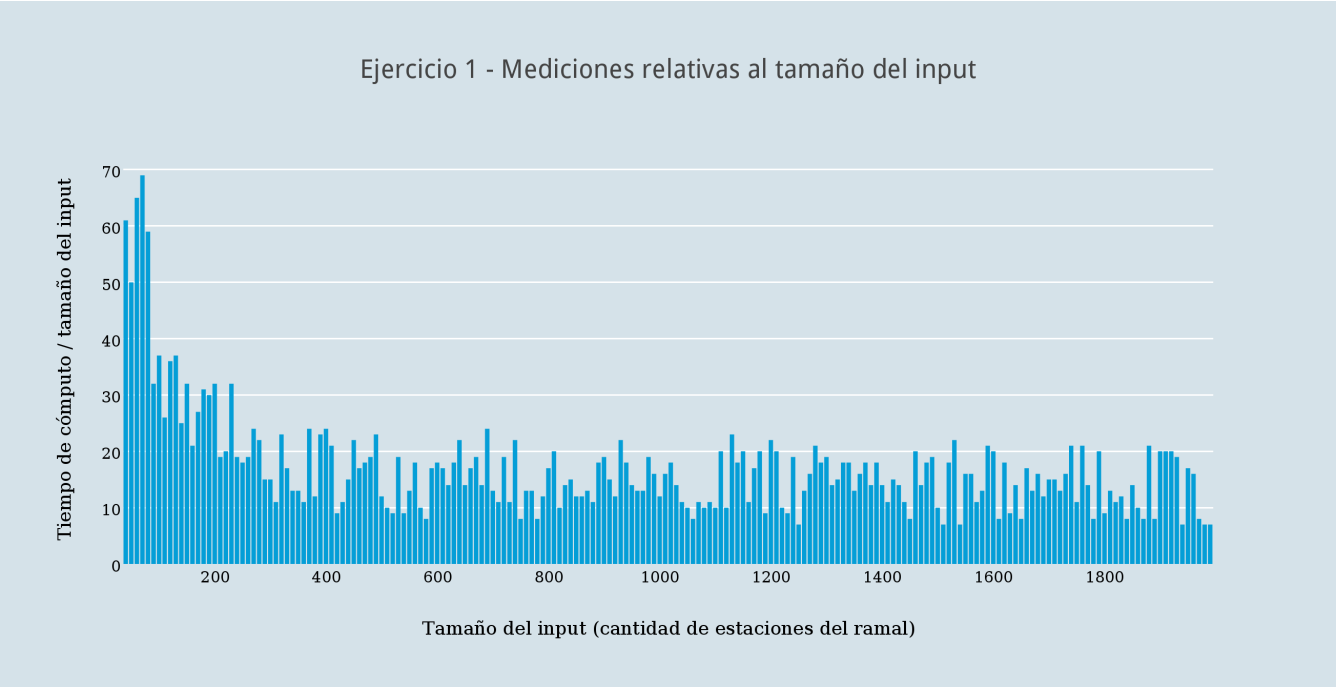


Figura 2: Mediciones relativas al tamaño del input

2. Cuestiones de implementación y entorno de testing

En relación a las cuestiones de implementación y entorno de testing aparecen involucrados una enorme cantidad de factores que influyen al momento de realizar las pruebas, como ser:

- prestaciones de hardware
- ejecución de programas y servicios del sistema operativo, de manera de manera sincrónica y paralera a la toma de mediciones
- manejo de la memoria caché
- ejecución del Garbaje Collector [3], etc.

Si bien varios de ellos escapan a nuestro control, decidimos seleccionar las mediciones con diversos criterios previamente mencionados, de manera que los resultados fueran tan representativos como fuese posible.

En cuanto a los casos de prueba de performance, hemos de recordar que fueron generados de manera pseudo-aleatoria, siendo el tamaño del input el único parámetro ofrecido para la generación de las diversas instancias.

Es por esto que podemos ver casos donde el tamaño del input es similar y los tiempos medidos son aparentemente contradictorios.
Tomemos, por ejemplo, los casos de 1960 y 1970 estaciones. Si se los compara, se puede ver que para la primera instancia I_1 el valor de Tiempo(I_1) / CantEstaciones(I_1) resultó ser exactamente el doble que el obtenido mediante el cálculo de Tiempo(I_2) / CantEstaciones(I_2).
Observando este y otros casos llamativos, se puede comprender lo planteado observando de qué manera

[3] El Garbage Collector es un proceso de baja prioridad que se ejecuta en la JVM y es el encargado de liberar la memoria que no se emplea (elimina del heap los objetos que no posean una referencia apuntándolos en el Stack). El hecho de que sea de baja prioridad implica que no puede ejecutarse en cualquier momento, si no que su comienzo se verá supeditado al manejo que haga el procesador con los trabajos de mayor prioridad.

particular fueron instanciados los problemas [4].
De esta manera y haciendo un seguimiento de la implementación para estos contextos, es posible atribuir diferencias temporales tan significativas entre parámetros cuantitativamente cercanos pero cualitativamente disímiles a la longitud del cable obtenida.
Esto cobra sentido al observar que en el caso de menor cantidad de estaciones la longitud con la que se contaba era de aproximadamente la mitad del kilometraje que separaba a la primera ciudad de la última, mientras que en el problema resuelto para mayor cantidad de estaciones el cable llegaba a unir 1866 de 1970 estaciones desde la primera iteración principal (es decir, comparando cuántas podían unirse desde la ciudad ubicada en el Km 0).

Esto es consistente con lo que se puede observar si se busca que instancia tuvo la mínima razón Tiempo/TamañoInput. En nuestras pruebas ocurrió para un ramal generado de manera que la longitud del cable era tal que permitía unir 1227 de 1250 estaciones partiendo de la inicial. [5]

Todo esto tiene sentido si se observa a la luz del código, donde una simple guarda impidió que el ciclo se ejecutara una cantidad de veces mayor a la necesaria: Las líneas a las que nos referimos son

```
while(ramal.HayCiudadesMasLejanas() && !ramal.EsLaUltimaCiudad(indiceCiudadMasLejanaAlcanzada))
```

Gracias a la segunda condición, podemos asegurar que el ciclo deja de ejecutarse cuando habiéndonos parado en una determinada ciudad inicial logramos alcanzar la última de las estaciones, lo cual sucede con mayor velocidad cuando no se ha de cambiar varias veces el índice de la ciudad inicial, situación que se cumple cuando el cable provisto es lo suficientemente extenso como para cubrir una cantidad considerable [6] de estaciones consecutivas.

[4] Todos los casos sobre los que fueron corridos las pruebas junto con las mediciones de tiempos correspondientes se encuentran en la carpeta /bin junto a otros datos útiles
[5] Ver instancia con 1250 estaciones
[6] Proporcional a la cantidad total de estaciones

2. Problema 2: A medias

2.1. Descripción de la problemática

En este problema se nos pide calcular todas las medianas parciales, que se obtienen al tomar subconjuntos del original de la siguiente manera: Suponiendo que tengo el siguiente conjunto $\{2, 8, 5, 3, 10\}$, tomamos los subconjuntos

$$\{2\}\ \{2, 8\}\ \{2, 8, 5\}\ \{2, 8, 5, 3\}\ \{2, 8, 5, 3, 10\}$$

Luego- entendiendo la mediana como el valor en la posición central o el promedio entre los dos valores en las posiciones centrales (en el caso que la cantidad de elementos sea par) de un conjunto de datos ordenados - la calculamos para cada uno de esos subconjuntos. De esta manera, con el algoritmo que analizaremos a continuación, podemos obtener el siguiente conjunto de medianas $\{2, 5, 5, 4, 5\}$ que es solución al problema y que cumple que el i-ésimo, con $i = \{0..4\}$, representa la parte entera de la mediana de los primeros i números de la entrada.

2.2. Resolución propuesta y justificación

La solución que proponemos utiliza dos heaps (uno que ordena los elementos de menor a mayor -min heap- y el otro de mayor a menor -max heap-) para obtener de forma rápida los elementos que se encuentran en la mitad del conjunto ordenado. Para lograr esto, el algoritmo almacena los elementos mas grandes en el min heap y los mas chicos en el max heap, decidiendo si es mas chico o mas grande al compararlo con la mediana del subconjunto anterior.

El pseudocódigo que se muestra a continuación representa nuestro algoritmo y en el mismo se utilizan las variables $heap1$ y $heap2$ solamente para mostrar que estamos comparando los dos heaps y que no importa diferenciar cual es el max y cual es el min heap.

Pseudocódigo:

$mediana =$ mediana actual (inicializada en cero)
for Cantidad de elementos en el conjunto **do**
  if Valor del elemento actual $\geq mediana$ **then**
    Insertar el valor en el min heap
  else
    Insertar el valor en el max heap
  end if
  if La diferencia absoluta entre los tamaños de los dos heaps > 1 **then**
    Pasar la cabeza del heap mas grande al otro heap
  end if
  if Tamaño del $heap1 =$ tamaño del $heap2$ **then**
    $mediana =$ el promedio entre las cabezas de los heaps
  else
    $mediana =$ la cabeza del heap mas grande
  end if
  Devolver $mediana$
end for

Los heaps se encargan de mantener ordenadas las dos mitades del conjunto, pero como no se sabe de antemano contra que valor comparar para ubicar los elementos en el heap correcto, los mismos se pueden debalancear, generando que los elementos del medio no se encuentren en las cabezas. Por este motivo si llega a ocurrir ese caso, balanceamos los heaps pasando la cabeza del heap mas grande al otro.

Luego calcular la mediana simplemente implica decidir si el subconjunto actual tiene cantidad par o impar de elementos y en función de eso, calcular el promedio de las dos cabezas o tomar la cabeza del heap mas grande.

Por el invariante del ciclo *for*: $\{i >= 0 \land i <$ `cantidad de elementos del conjunto` $\land$ $mediana =$ `mediana de los primero` i `números`$\}$, podemos asegurar que el ciclo termina, ya que se cumple antes y después de cada iteración del ciclo y al darse la condición de terminación ($i >=$ `cantidad de elementos del conjunto`), efectivamente termina.

Luego por la documentación de *Priority Queue* sabemos que los heaps van a mantener los elementos ordenado en todo momento, dándonos fácil acceso al mínimo y el máximo. Al rebalancear los heaps nos aseguramos de que siempre se cumpla que la diferencia absoluta entre los tamaños de los heaps sea menor igual a uno, lo que causa que los heaps tengan la mitad o la mitad mas uno de los elementos del subconjunto actual.

Teniendo en cuenta que vale lo anterior y por estar ordenados de forma ascendente el que contiene los elementos más pequeños y de forma descendente el otro, se cumple que las cabezas siempre representan a los elementos centrales del subconjunto actual con j elementos, o sea, los valores en las posiciones $j/2 \land (j/2) + 1$. Por lo que para calcular la mediana podemos obtener los tamaños de los heaps y decidir si hay que calcular el promedio o devolver la cabeza del mas grande. De esta manera el algoritmo cumple con la definición de mediana para cada subconjunto incremental.

2.3. Análisis de la complejidad

Nuestra solución tiene una complejidad temporal de $\mathcal{O}(n \log n)$ siendo n la cantidad de elementos del conjunto.

El algoritmo está contenido en un *for* que recorre una sola vez cada elemento del conjunto, con lo cual la ejecución de ese ciclo tiene una complejidad de $\mathcal{O}(n)$. Luego se utilizan los métodos de la clase *Heap* que utiliza como base a la clase *Priority Queue* que está incluida en *java.util*. Las complejidades de todos los métodos de *Priority Queue* que se mencionan a continuación, están especificadas en la documentación de java[7]. El algoritmo ejecuta los métodos de *Heap* de la siguiente manera:

- Insertar elemento en heap: Es $\mathcal{O}(1)$ elegir en que heap se va a insertar el elemento mas $\mathcal{O}(\log n)$ insertar en la cola de prioridad con *PriorityQueue.add()*.
  Notar que el elemento nunca se inserta en los dos heaps, por lo tanto la complejidad queda $\mathcal{O}(1) + \mathcal{O}(\log n) = \mathcal{O}(\log n)$.
- Balancear heaps: Hace comparaciones y asignaciones en $\mathcal{O}(1)$, luego llama a *PriorityQueue.poll()* que obtiene y remueve la cabeza del heap en $\mathcal{O}(\log n)$ y por último inserta la cabeza en el otro heap con *PriorityQueue.add()* también en $\mathcal{O}(\log n)$.
  Quedando una complejidad de $k * \mathcal{O}(1) + 2 * \mathcal{O}(\log n) = \mathcal{O}(\log n)$ con k una constante.
- Calcular mediana: Hace operaciones aritméticas, asignaciones y comparaciones en $\mathcal{O}(1)$, obtiene los tamaños de los heaps con *PriorityQueue.size()* también en $\mathcal{O}(1)$ y obtiene la cabeza de uno o de los dos heaps, lo cual no hace diferencia ya que la complejidad de *PriorityQueue.peek()* es $\mathcal{O}(1)$.
  En este paso nos queda una complejidad de $k * \mathcal{O}(1) = \mathcal{O}(1)$ con k una constante.

Por lo tanto, siguiendo el análisis de complejidad, nos termina quedando:

$$\mathcal{O}(n) * (\mathcal{O}(\log n) + \mathcal{O}(\log n) + \mathcal{O}(1)) = \mathcal{O}(n \log n)$$

Lo cual condice la afirmación que hicimos al principio de la subsección.

[7] http://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html

2.4. Código fuente

A continuación se incluyen las partes más relevantes del código.
El loop principal que se encuentra en la clase *Main.java*:

```
        mediana = 0;
        tam = conj.length;
        time0 = System.nanoTime();

        for(int i = 0; i < tam; i++){
           val = conj[i];
           if(val >= mediana){
              heap.insertMinHeap(val);
           }else{
              heap.insertMaxHeap(val);
           }

           heap.balancearHeaps();

           mediana = heap.calcularMediana();
           //imprimir solucion
           esc.EscribirInt(mediana);
        }
```

Los métodos que manipulan los heaps que se encuentran en la clase *Heap.java*

```
public Heap(){
   this.minHeap = new PriorityQueue<Integer>();
   this.maxHeap = new PriorityQueue<Integer>(11, new MyComparator());
}

public void insertMinHeap(int val){
   this.minHeap.add(val);
}

public void insertMaxHeap(int val){
   this.maxHeap.add(val);
}
```

```
public void balancearHeaps(){
   /*si la diferencia absoluta entre la cantidad de elementos
   es mayor a uno, tengo que rebalancear.
   si es 0, 1 o -1 esta bien.
   */
   int head;
   int balance = this.minHeap.size() - this.maxHeap.size();
   if(balance < -1){
       //maxHeap es mas grande
      head = this.maxHeap.poll();
      this.minHeap.add(head);
   }

   if(balance > 1){
      //minHeap es mas grande
      head = this.minHeap.poll();
      this.maxHeap.add(head);
   }
}
```

```
public int calcularMediana(){
   int balance = this.minHeap.size() - this.maxHeap.size();
   int res = 0;
   if(balance == 0){
      res = this.minHeap.peek() + this.maxHeap.peek();
      res = res/2;
   }else if(balance > 0){
      res = this.minHeap.peek();
   }else{
      res = this.maxHeap.peek();
   }
   return res;
}
```

2.5. Experimentación

Para analizar la performance del algoritmo creamos dos scripts: uno que se encarga de generar casos de test pseudoaletorios[8], el cual genera archivos con instancias del problema incrementando el n con un cierto espaciado; El otro script[9] se encarga de correr el programa una cierta cantidad de veces para cada instancia generada por el primer script, luego descarta los outliers y calcula el tiempo promedio entre todas las corridas. Estos promedios se guardan en el archivo *resultados.out*.

2.5.1. Constrastación Empírica de la complejidad

A continuación presentamos dos gráficos que muestran el comportamiento en la práctica del algoritmo. Para obtener estos resultados utilizamos 301 tests cada uno conteniendo una instancia del problema en las que el n varía entre 1 y 3001, saltando de a 10 números y los valores de cada instancia se generaron pseudoaleatoriamente con un rango entre 1 y 10000. Luego, cada una de estas instancias fue ejecutada 100 veces.

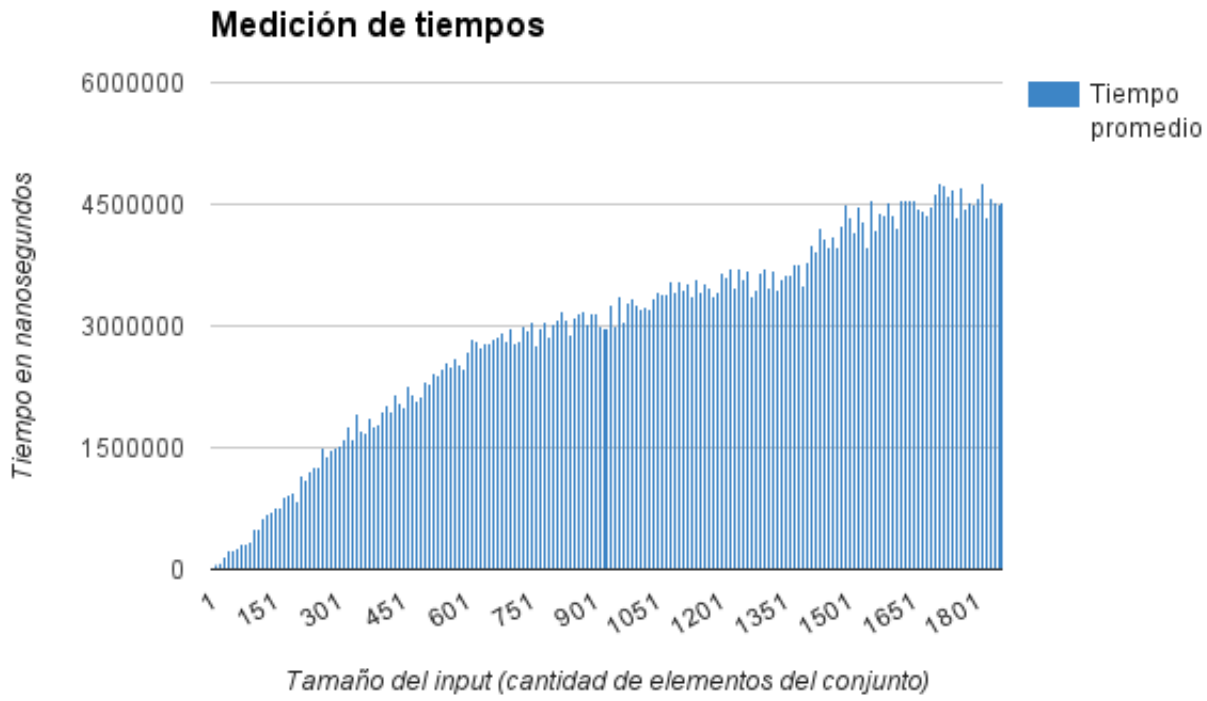


Figura 3: Medición de tiempo promedio entre 100 corridas

Al mirar el gráfico de la figura 3 no queda muy claro que la cota de complejidad propuesta sea correcta. Por lo que decidimos linealizar los resultados, dividiéndolos por $k * \log n$ con n tamaño del input y k una constante. Como se esperaba, el gráfico de la figura 4 se asemeja bastante a una recta. Con lo cual parecería que el algoritmo cumple con la cota de complejidad teórica que analizamos anteriormente.

Por último, tomamos los tiempos promedio y los dividimos por la cota teórica para poder observar si los resultados tienden a una constante.

La figura 5 muestra este análisis en el que efectivamente los resultados parecen tender a una constante. De esta forma concluimos que el algoritmo cumple con la cota de complejidad propuesta.

[8] genTest.sh
[9] testPerformance.sh

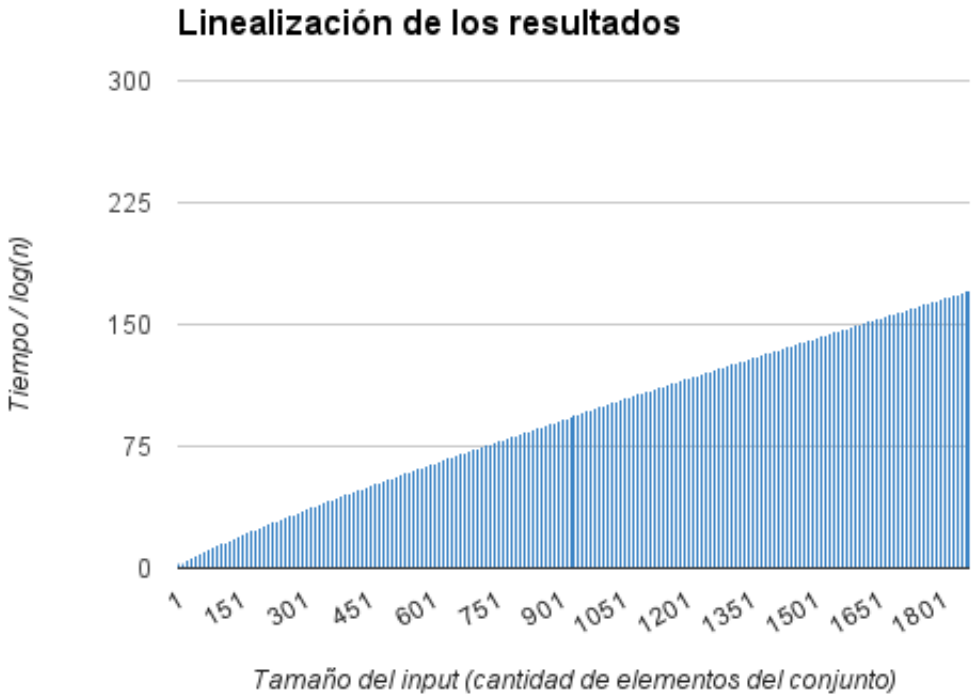


Figura 4: Tiempo promedio dividido por el logaritmo del tamaño del input

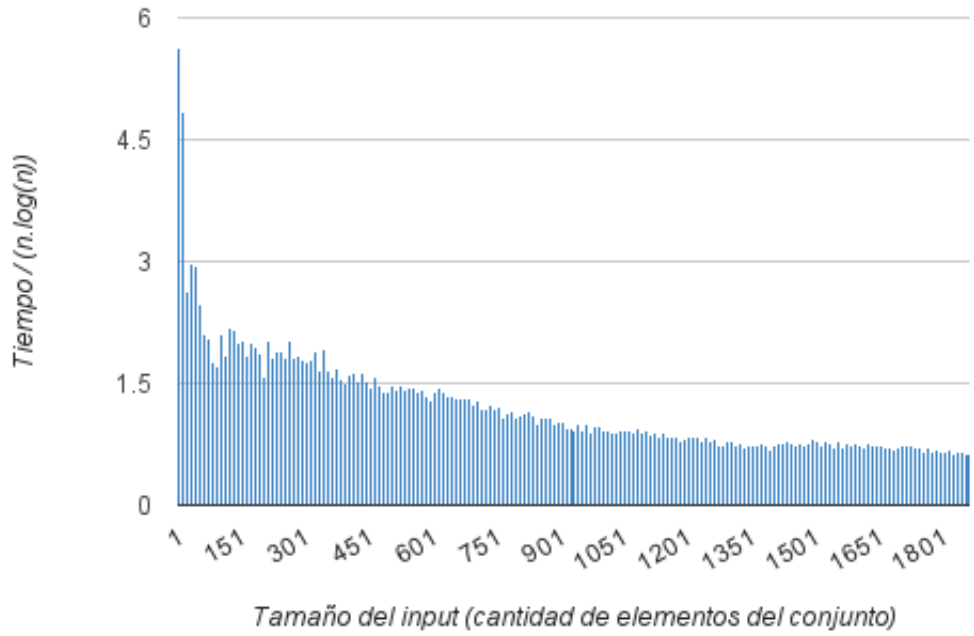


Figura 5: Tiempo promedio dividido por la cota de complejidad

3. Problema 3: Girl Scouts

3.1. Descripción de la problemática

En este ejercicio existe un grupo de niñas exploradoras, algunas de ellas amigas entre sí, que deben formarse en una ronda. Siendo e la cantidad de exploradoras, existen $(e-1)!$ permutaciones posibles en las cuales pueden sentarse. Podemos medir la distancia entre dos amigas como la mínima cantidad de chicas que hay entre una y otra, más 1. Por ejemplo, si una chica está al lado de la otra, la distancia entre ellas es 1. Si tenemos una ronda de tres chicas, la distancia entre cualquier par de chicas va a ser 1.

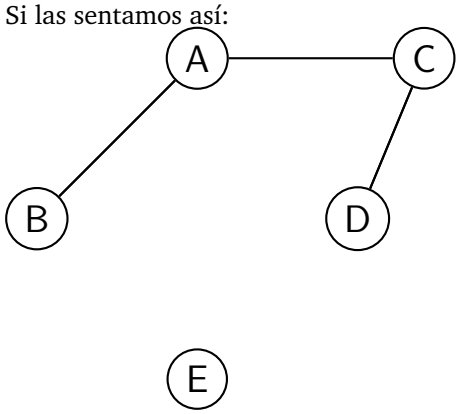
El problema consiste en hallar la forma de ubicar a las exploradoras de forma tal que exista la menor distancia posible entre cada amistad, y logrando que la suma total de la distancia entre todos los pares de amigas sea mínima.

El formato de entrada es un archivo de texto que 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.

Veamos un ejemplo. Tenemos un grupo de 5 chicas: A, B, C, D y E. A es amiga de B y C, y D es amiga de C. Esto se puede escribir de muchas maneras. Si escribimos a cada amistad dos veces, podemos hacerlo así:

```
A BC;B A;C AD;D C;E
```

Si las sentamos así:



vamos a estar obteniendo una solución óptima, ya que la suma total de distancias es 3 (contando a cada amistad una sola vez) y la distancia máxima es 1.

El formato de salida para cada grupo de exploradoras es una línea de texto donde se debe indicar la distancia máxima obtenida, seguida de un espacio y una cadena de letras que indica la forma en que fueron sentadas las exploradoras. En caso de haber obtenido más de una solución, se debe devolver la primera en órden alfabético. Para el último ejemplo, la forma de codificar sería así:

```
1 ACDEB
```

Nótese que es equivalente a esta otra: `1 CDEBA`, ya que la única diferencia es por cuál exploradora empezamos a describir la ronda.

3.2. Algoritmo desarrollado

Desarrollamos un algoritmo que utiliza la técnica de *backtracking*. Esta técnica consiste en generar de forma incremental candidatos parciales para formar soluciones. De forma implícita, los candidatos parciales pueden ser representados como los nodos de un árbol; las soluciones completas son las hojas del árbol. El algoritmo va generando el árbol de forma recursiva al hacer un recorrido DFS. En cada paso, se hace una evaluación que permite decidir si las hojas descendientes del candidato parcial actual pueden ser solución o no del problema. De esta manera, se evalúa todo el árbol de posibilidades sin necesariamente recorrerlo entero, ya que se pueden descartar ramas enteras antes de entrar en ellas.

Aplicando esta técnica a este problema en particular, desarrollamos un algoritmo de *backtracking* que busca las mejores permutaciones posibles para las niñas exploradoras. El algoritmo comienza con una ronda vacía con tantos lugares como exploradoras hay en el grupo. Los lugares son fijos, y medimos la

distancia entre dos chicas como la cantidad de lugares que me tengo que “mover” para llegar de una chica a la otra.

Separamos a las exploradoras en dos grupos: aquellas que no tienen ninguna amiga, y aquellas que sí (tienen al menos una amistad). A las que no tienen amigas las ubicaremos en la ronda solamente al final, ya que, una vez ubicadas todas las exploradoras que sí tienen amigas, aquellas que no tienen amistades no influirán en las distancias, y cualquier permuitación de las chicas sin amigas será equivalente en el resultado final (podemos simplemente generar la que es alfabéticamente primera, descartando las otras).

En cada paso del algoritmo recursivo vamos agregando una chica a la ronda en un lugar determinado (que queda fijo), sin modificar la ubicación de las que ya están en la ronda. Luego, evaluamos si la distancia máxima entre pares de amigas es mayor a la mejor obtenida (para alguna solución óptima hallada anteriormente). Si lo es, entonces descartamos todas las ramas que descienden de este caso y retornamos, de manera tal de continuar la ejecución por otra rama del árbol. Si no lo es, continuamos la exploración del árbol, a menos que ya hayamos llegado a una hoja. En tal caso, si la solución obtenida es mejor a la que hasta ahora era la mejor, la nueva solución pasa a ser la mejor. Al finalizar el recorrido completo, se devuelve la solución que fue guardada como la mejor.

Para medir la sumatoria de distancias entre pares de amigas en cada iteración no es necesario recalcular todas las distancias en la ronda. Simplemente se debe calcular las las distancias de la exploradora que se agrega o se quita de una determinada posición con sus amigas que ya esten colocadas.

3.3. Justificación de correctitud

El hecho de estar generando todo el árbol de posibilidades nos asegura no estar dejando fuera soluciones posibles. El cuidado debe estar, entonces, en realizar las podas de forma correcta y de medir distancias de forma correcta. Analicemos más en detalle las podas realizadas:

- Una vez ubicadas todas las exploradoras que tienen amigas, no probamos las distintas combinaciones con aquellas exploradoras que no tienen amigas: simplemente elegimos la permutación que está primera alfabéticamente. Esto es correcto ya que las exploradoras que no tienen amigas no forman parte de ninguna amistad, con lo cual, no cambian en nada a las distancias.
- A medida que se van formando los distintos fogones, se chequea en cada instancia si el caso que se esta construyendo produce una mayor separación promedio que el mejor caso registrado. Es decir, si en un fogón, aún estando incompleto, se registra mayor separación entre amigas, se concluye que todas las posibles permutaciones que partan desde ese punto no seran óptimas.

3.4. Análisis de la complejidad temporal

La cantidad de posibles permutaciones para una ronda es de ’(e-1)!’; en el peor escenario posible, los datos de entradas tienen un orden tal que el algoritmo encuentre una mejor situación en cada iteración, por lo cual tendrá que recorrer las ’(e-1)!’ permutaciones. Cada permutación del fogon cuesta no mas de ’2*e’ pasos. En cada paso, al colocar o quitar una exploradora, se debe calcular la alteración que se produce en la distancia de amistades. Para esto se debe chequear para cada amiga de la exploradora si esta en el fogon y, en caso afirmativo, cual es su posición. Logicamente, la cantidad de amistades de una exploradora no excede ’a’ y, como se utiliza un TreeMap para identificar las posiciones de las exploradoras con amigas, determinar la posicion de cada amiga en el fogon puede resolverse en ’ln a’ En un peor caso, entonces, la complejidad del algoritmo pertenece a $O((e-1)!ealna)$ lo cual puede ser acotado por $O(e^e alna)$, estrictamente menor a la complejidad solicitada en el enunciado.

3.5. Tests de correctitud

3.6. Experimentación para observar la performance real

3.6.1. Peor caso

3.6.2. Caso promedio