

1 Problema 1: Telégrafo

1.1 Descripción del problema

El problema del telégrafo consiste en conectar mediante un cable de telégrafo a las estaciones de los ramales del sistema férreo que se encuentra en el país, sabiendo que cada ramal dispone de una cierta cantidad de cable para conectar sus estaciones, cuya longitud se mide en kilómetros.

Lo que buscamos conseguir es que dado un ramal con sus estaciones y los kilómetros de cable de los que dispone, saber cual es el mayor número de estaciones seguidas que se pueden conectar con dicho cable. Para esto contamos con los ramales y sus estaciones, la distancia en kilómetros de cada una de las estaciones a la capital, que se encuentra en el kilómetro 0, y la cantidad de cable de la que dispone cada ramal.

En los datos de entrada, cada ramal ocupa dos líneas. La primera representa los kilómetros de cable de los que dispone dicho ramal, y la segunda las distancias de las estaciones del ramal a la capital.

Como datos de salida, el algoritmo encargado de resolver el problema debe devolver una línea por cada ramal, donde cada línea indica la cantidad máxima de estaciones de dicho ramal que se pueden conectar con el cable de telégrafo.

Ejemplos:

1. **Kilómetros de cable:** 6. **Distancia de las estaciones:** 6 8 12 15.

En este ejemplo la máxima cantidad de estaciones que puedo conectar con 6 kilómetros de cable es 3, las cuales son las que se encuentran en el kilómetro 6, 8 y 12, ya que desde las estaciones que se encuentran en el kilómetro 0 y el 8 solo puedo conectar 2, las que se encuentran en el kilómetro 0 y 6 para la primera, y las que se encuentran en el kilómetro 8 y 12 para la segunda.

2. **Kilómetros de cable:** 6. **Distancia de las estaciones:** 7 14 21 28.

En este caso, la distancia entre cada una de las estaciones es mayor que la cantidad de kilómetros de cable del que se dispone, por lo que la solución de este caso es que hay 0 estaciones que se pueden conectar mediante el cable.

3. **Kilómetros de cable:** 6. **Distancia de las estaciones:** 3 6 9 12 15 18.

En este último caso las distancias entre todas las estaciones es la misma, y con los kilómetros de cable de los que disponemos podemos conectar como máximo hasta 3 estaciones.

1.2 Resolución

Lo que nuestro algoritmo hace primero es buscar, en un vector con las distancias de las estaciones del ramal a la capital llamado *distanciaDeLasEstaciones*, la máxima cantidad de estaciones que puedo conectar con la capital, lo cual me va a servir como cota mínima de la cantidad máxima de estaciones que puedo conectar.

Para esto recorro el vector linealmente hasta encontrar la última estación alcanzable desde la capital con el cable. El valor resultante lo guardo en *maximaCantidadDeEstaciones*, que va a servir para almacenar la máxima cantidad de estaciones conectables con el cable de telégrafo.

Luego, utilizando dos índices, *inicio* y *fin*, vuelvo a recorrer *distanciaDeLasEstaciones* para encontrar la máxima cantidad de estaciones que voy a poder conectar con el cable. Los índices *inicio* y *fin* van a representar la posición de las estaciones dentro del vector *distanciaDeLasEstaciones*, y van a comenzar inicializados con valores 0 y 1 respectivamente.

La manera en la que voy ir recorriendo dicho vector es ir aumentando el valor de *fin* hasta que la cantidad de estaciones entre *inicio* y *fin* supere el valor de *maximaCantidadDeEstaciones* o me quede sin estaciones para recorrer.

En caso de que suceda lo primero reemplazo el contenido de *maximaCantidadDeEstaciones* por la cantidad de estaciones entre *inicio* y *fin* (sin contar la última estación a la que el cable no llega), aumento en uno el valor de *inicio*, y vuelvo a comenzar a aumentar el valor de *fin*.

Por otro lado, como *inicio* no puede ser mayor que *fin*, si cuando tengo que aumentar el valor de *inicio* este iguala al valor de *fin*, en vez de aumentar únicamente *inicio*, aumento el valor de ambos

índices en uno.

Cuando me quede sin estaciones para recorrer devuelvo el valor de *maximaCantidadDeEstaciones* como resultado del problema.

Para finalizar, a continuación mostramos un pseudocódigo de lo explicado anteriormente.

Algorithm 1: Telégrafo

Input: int *kilometrosDeCable*, vector< int> *distanciaDeLasEstaciones*

```
1 int inicio ← 0
2 int fin ← 1
3 int cantidadDeEstaciones ← 0
4 int maximaCantidadDeEstaciones ← última estación alcanzable desde la capital con el cable
5 mientras fin < total de estaciones del ramal
6     Si puedo unir con cable las estaciones entre inicio y fin
7         cantidadDeEstaciones ← La cantidad de estaciones entre inicio y fin
8         Si cantidadDeEstaciones > maximaCantidadDeEstaciones
9             maximaCantidadDeEstaciones ← cantidadDeEstaciones
10        Aumento el valor de fin en 1
11    Si no puedo unir las estaciones entre inicio y fin con el cable
12        Si fin es igual a inicio+1 aumento ambos índices en 1
13        Si no, aumento únicamente inicio en 1
14 Devuelvo el valor almacenado en maximaCantidadDeEstaciones
```

1.3 Justificación

Como se menciona en el algoritmo explicado en el punto anterior, se van a recorrer las estaciones del ramal, las cuales se encuentran en el vector *distanciaDeLasEstaciones*, utilizando los índices *inicio* y *fin*.

Se va a ir aumentando el valor del índice *fin* mientras pueda seguir conectando las estaciones con el cable, y cuando no pueda conectarlas se va a ir aumentando el índice *inicio* hasta poder volver a hacerlo. Eso quiere decir que por cada valor que toma el índice *inicio* se va a buscar la última estación que se pueda conectar con el cable a partir de dicho índice, actualizando el valor de *maximaCantidadDeEstaciones* si la cantidad de estaciones supera su valor.

Por ende, como el valor de *maximaCantidadDeEstaciones* solo almacena la máxima cantidad de estaciones conectables por el cable de telégrafo, y desde cada estación marcada con el índice *inicio* calculo la máxima cantidad de estaciones que puedo conectar con el cable desde dicha estación, *maximaCantidadDeEstaciones* va a terminar almacenando el valor de la máxima cantidad de estaciones desde alguna estación, es decir, la respuesta al problema.

1.4 Complejidad

Llamemos n a la cantidad de estaciones. Como lo que hace el algoritmo es recorrer con dos índices el vector de estaciones de tamaño n , y ninguno de los dos índices decrece, al final lo que termina haciendo es recorrer el vector de estaciones hasta el final con cada índice. La complejidad de recorrer un vector dos veces es $O(2n)$, que es lo mismo que $O(n)$.

Por otro lado, a parte de recorrer el vector de estaciones, lo que hace el algoritmo es hacer acceder a dicho vector mientras lo recorre y hacer cuentas entre las estaciones ubicadas en los índices *inicio* y *fin* del vector para saber si se pueden conectar mediante el cable de telégrafo, actualizar el valor de *maximaCantidadDeEstaciones*, y aumentar el valor de los índices, todo lo cual tiene una complejidad $O(1)$.

Por todo lo anterior, la complejidad final del algoritmo es $O(n)$.

1.5 Casos de Test

Para corroborar el correcto funcionamiento del algoritmo, este mismo se probó con el siguiente conjunto de tests.

1. **Kilómetros de cable: 5. Distancia de las estaciones: 6 12 18 24**

Este primer test se utilizó para corroborar que el algoritmo devolviera 0 en caso de que no se pudiera conectar ninguna estación mediante el cable, dado que la distancia entre cada una de las estaciones es mayor que la cantidad de kilómetros de cable disponible.

2. **Kilómetros de cable: 0. Distancia de las estaciones: 3 8 12 17**

Este test se utilizó para corroborar que si la cantidad de cable disponible para conectar las estaciones es 0, el algoritmo devolviera 0, ya que sin cable las estaciones no se pueden conectar.

3. **Kilómetros de cable: 6. Distancia de las estaciones: 6 8 12 15**

Este test, dado por la cátedra, se utilizó para corroborar el correcto funcionamiento del algoritmo dado una instancia normal, es decir, una instancia sin ningún caso especial.

4. **Kilómetros de cable: 34. Distancia de las estaciones: 6 10 22 34**

Este test sirvió para corroborar que si el cable alcanzara para conectar todas las estaciones, el algoritmo devolviera la cantidad total de estaciones, incluida la de la capital, como resultado.

5. **Kilómetros de cable: 5. Distancia de las estaciones: 5 10 15 20 25**

Por último, este test se utilizó para corroborar que si la distancia entre todas las estaciones fuera igual a la cantidad de cable, el algoritmo devolviera 2 como resultado, ya que solo dos estaciones se pueden conectarse mediante el cable.

1.6 Tiempos de ejecución y gráficos

Para poder observar la performance en términos de tiempo del problema, se corrieron los siguientes casos de tests, variando en cada uno la cantidad de estaciones del ramal, la cantidad de cable de telégrafo disponible para conectar las estaciones, y la distancia entre dichas estaciones.

1. En el primer test utilizado para observar el tiempo de ejecución del problema, la cantidad de cable que se utilizó para poder conectar las estaciones fue de 200.

El mismo test se corrió 1000 veces, en donde en cada iteración la cantidad total de estaciones iba aumentando en 100, es decir, comienza con 100 estaciones en la primera iteración del test, y para la última iteración termina con 100000 estaciones. Luego, para cada una de las iteraciones del test, la distancia entre cada una de las estaciones se mantuvo en 10.

El figura 1 muestra el resultado de dicho test.

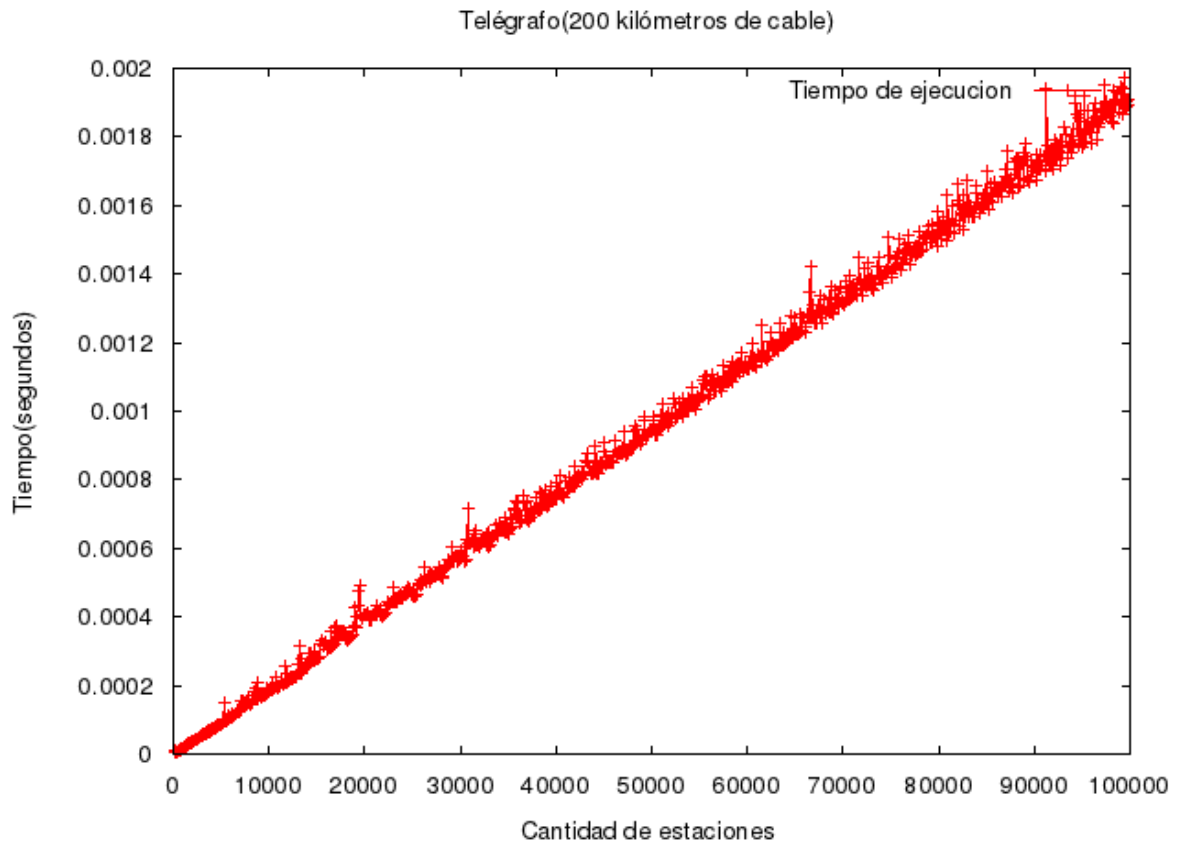


Figure 1: Primer test

2. Para el segundo test se mantuvieron los mismos datos que para el primero, a excepción de la cantidad de cable, que se disminuyó a 50.

Como puede verse en la figura 2, el tiempo que tardó en resolver el mismo problema pero con una cantidad de cable menor es el mismo, con lo que podemos afirmar que la cantidad de cable utilizada para conectar las estaciones no influye en el tiempo, teniendo en cuenta que la distancia entre estaciones es menor que la cantidad de cable disponible.

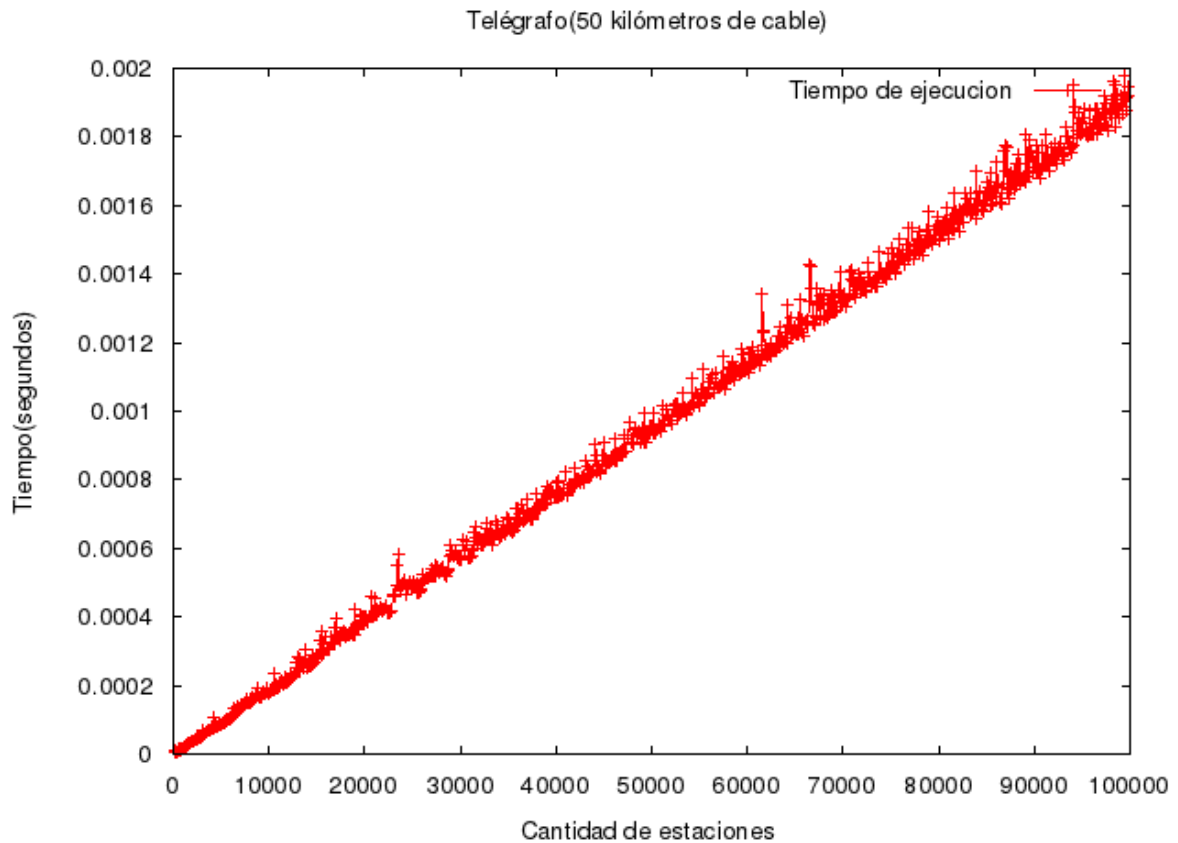


Figure 2: Segundo test

3. Ahora, en el tercer test, para poder observar si la distancia entre las estaciones influye en el tiempo de ejecución del problema, se corrieron las mismas instancias que en el primer test, salvo que ahora la distancia entre cada una de las estaciones se hizo de 220, de manera que la cantidad de cable no alcanzara para conectar ninguna estación.

En la figura 3 puede observarse que el tiempo de ejecución de dicha instancia decreció respecto de lo que tardó en el primer test. Esto sucede porque cada vez que el algoritmo no puede conectar una estación con la siguiente mediante el cable pasa a la siguiente estación para corroborar lo mismo. Esto resulta en que el arreglo de estaciones se recorra una única vez, en vez de un total de 2 veces si siempre encuentra una estación para poder conectar mediante el cable.

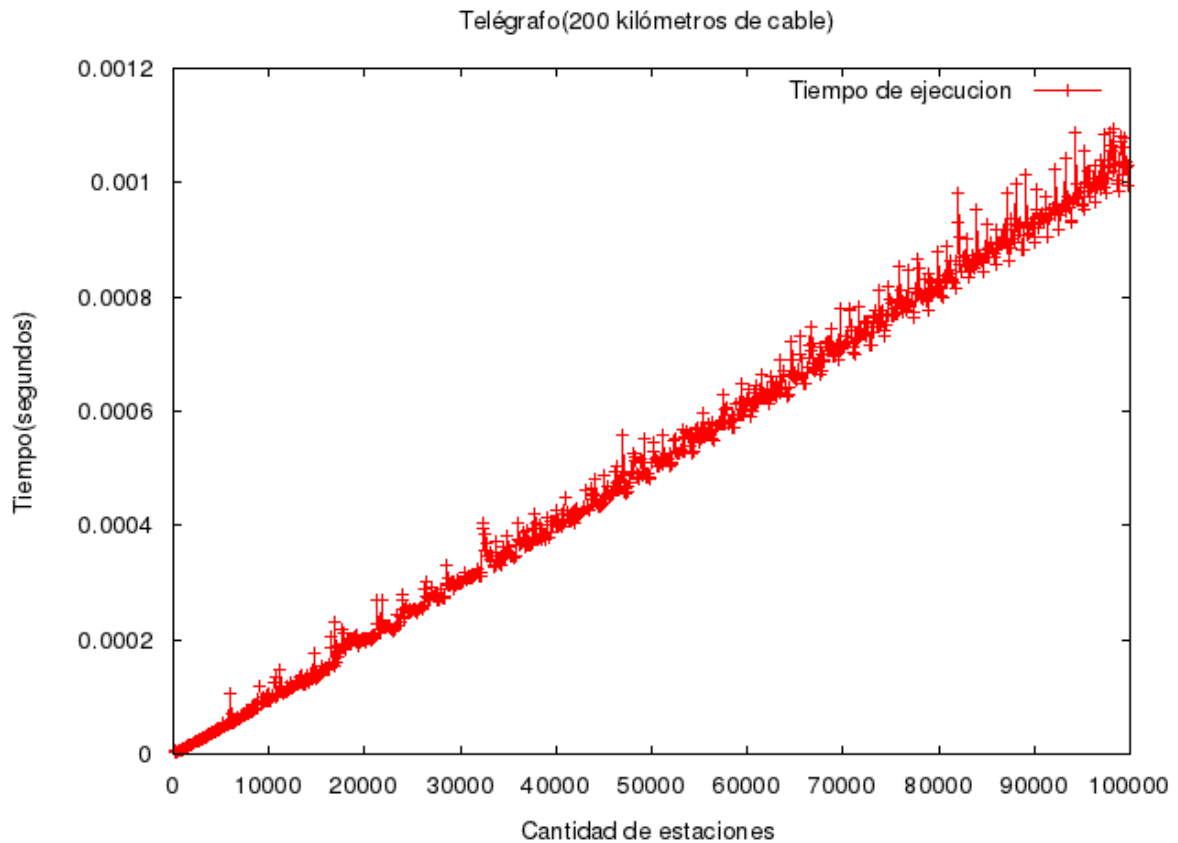


Figure 3: Tercer test

4. El siguiente test se utilizó para corroborar si la cantidad de estaciones efectivamente influye en el tiempo de ejecución. En este caso, se tomaron los mismos valores que en el primer test, pero se aumento la cantidad de iteraciones del test a 30000, es decir, que para la última iteración del test la cantidad de estaciones se hace de 300000.

En la figura 4 se aprecia claramente que, como era de esperarse, mientras mas estaciones haya que revisar, mas tiempo le toma al algoritmo encontrar la solución al problema.

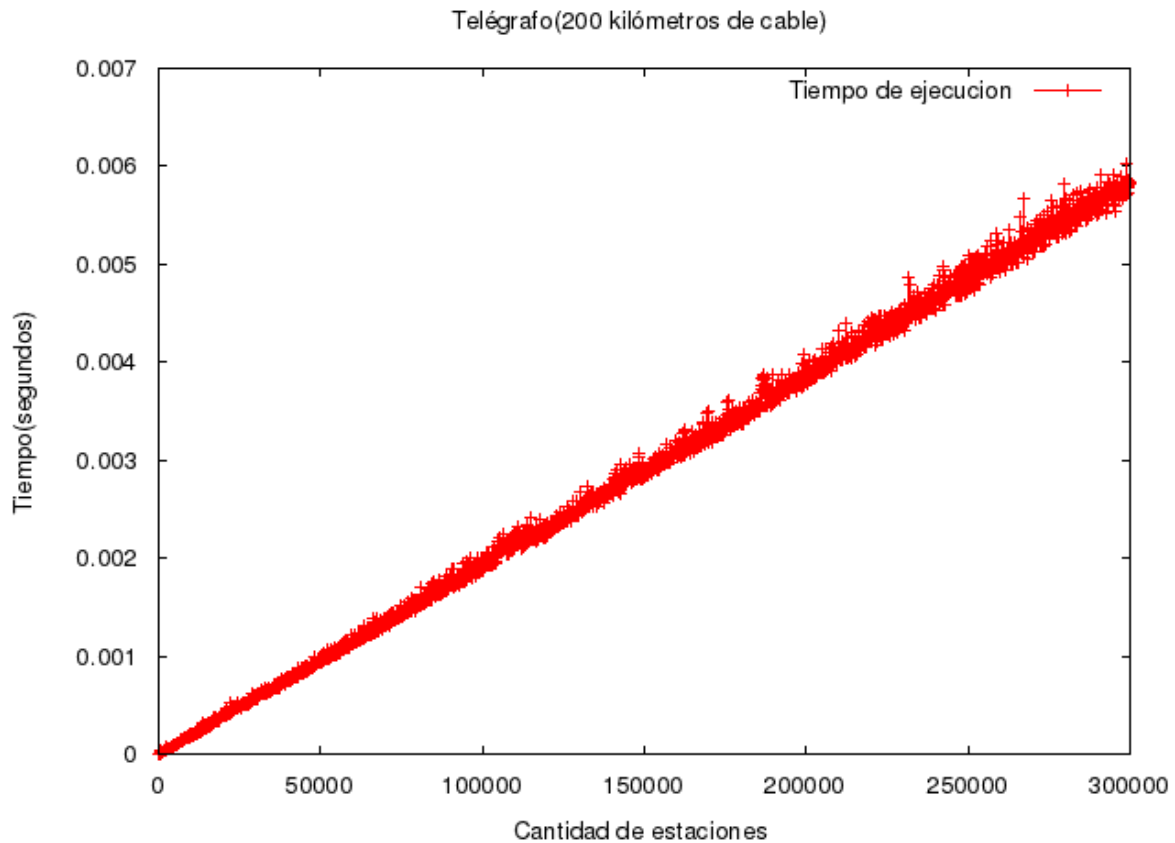


Figure 4: Cuarto test

5. Para terminar, el siguiente test fue utilizado para corroborar que la complejidad del algoritmo fuera la esperada. Para eso, se corrieron la misma cantidad de instancias que para el test 1 y con los mismos parámetros a excepción de la distancia entre cada una de las estaciones. En este caso cada estación está separada aleatoriamente entre 100 y 200 kilómetros de la siguiente estación. Luego, a los tiempos resultantes de dicha experimentación se los dividió por el tamaño de la entrada, es decir, la cantidad de estaciones en cada iteración del test. Las figuras 5 y 6 muestran el tiempo de ejecución de correr dicho test, y el resultado de dividir los tiempos obtenidos por el tamaño de la entrada, y como puede observarse en la figura 6, la función de tiempo sobre tamaño de entrada tiende a una constante, por lo que podemos asumir que la complejidad del algoritmo es, según lo esperado, $O(n)$.

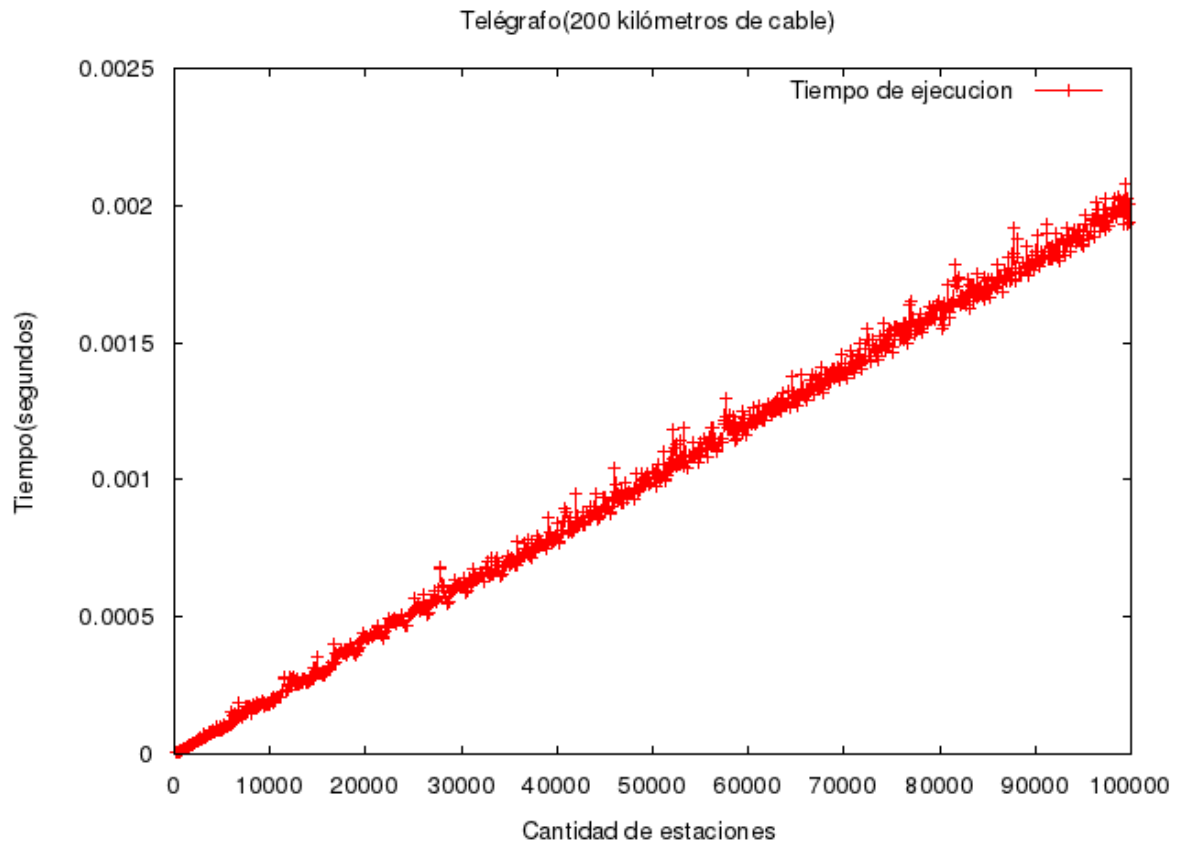


Figure 5: Quinto test

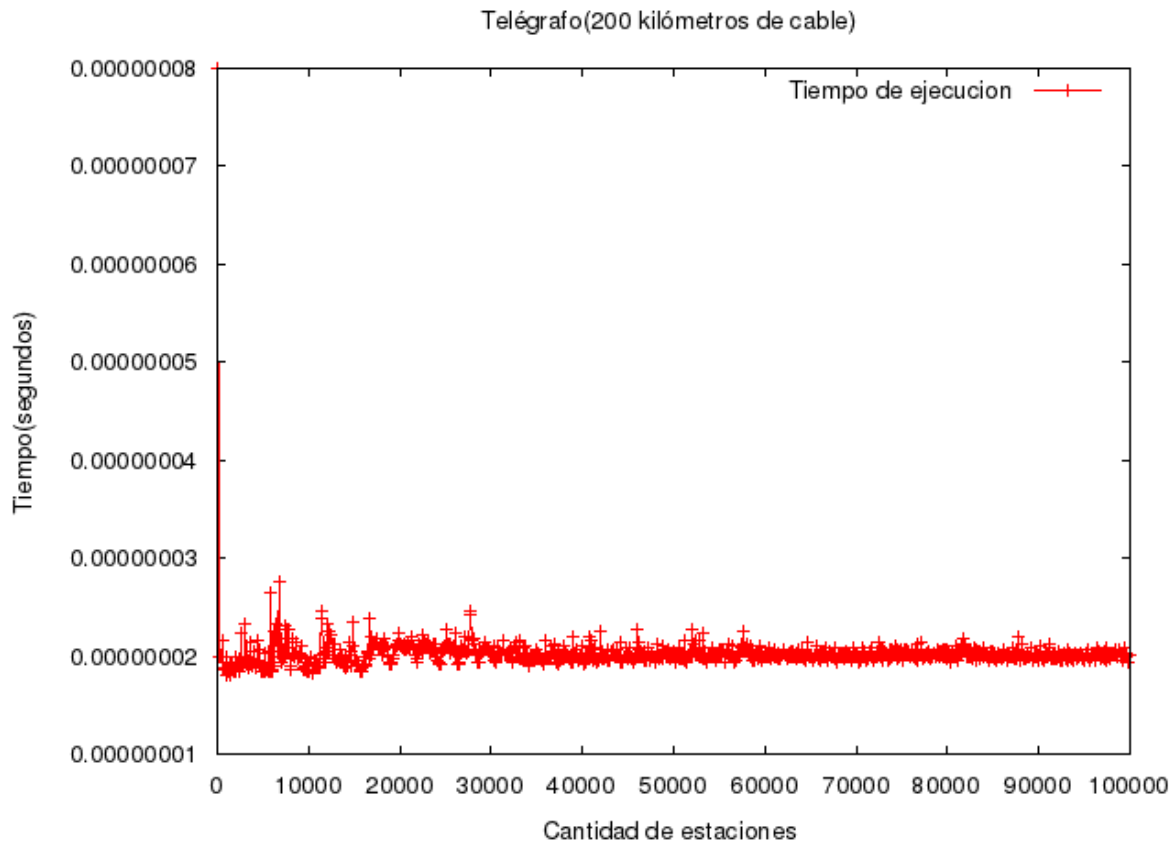


Figure 6: Quinto test comparando con tamaño de la entrada

1.7 Partes relevantes del Código

El siguiente algoritmo es el encargado de encontrar la máxima cantidad de estaciones que se pueden conectar mediante el cable de telégrafo proporcionado al ramal.

```
void calcularMaximaCantidadDeEstaciones(int kilometrosDeCable,vector<int> distanciaDeLasEstaciones

    int inicio=0;
    int fin=1;
    int cantidadDeEstaciones=0;
    int maximaCantidadDeEstaciones=0;
    int j=0;

    while(distanciaDeLasEstaciones[j]<=kilometrosDeCable)
    {
        maximaCantidadDeEstaciones=j+2;
        j++;
    }
    while(fin<distanciaDeLasEstaciones.size()){

        if((distanciaDeLasEstaciones[fin]-distanciaDeLasEstaciones[inicio])<=kilometrosDeCable){
            cantidadDeEstaciones=fin-inicio+1;

            if(cantidadDeEstaciones>maximaCantidadDeEstaciones)
                maximaCantidadDeEstaciones=cantidadDeEstaciones;
```

```

        fin++;
    }
    else{
        if(fin==inicio+1){
            fin++;
            inicio++;
        }
        else{
            inicio++;
        }
    }
}
cout<<maximaCantidadDeEstaciones<<endl;
}

```

2 Problema 3: Girl Scouts

2.1 Descripción del problema:

El problema consiste en obtener a partir de una lista de exploradoras, con forma de ronda y un peso entre cada par de elementos determinado por la distancia entre estos, obtener una permutación de ella tal que se haya minimizado la suma de los pesos de todos los elementos. Se debe devolver dicha lista en el orden que cumpla con la condición y un número correspondiente a el peso máximo entre dos elementos de la ronda obtenida. En caso de existir más de una solución, se debe devolver la de menor orden lexicográfico. Notar que si dos elementos no están relacionados, no son amigas, entonces el peso es nulo.

Ejemplos:

1. Se tiene la siguiente lista, vista como una ronda: $l = [a, b, c, d]$ y la siguiente relación de amistad $a = (a, c)(c, d)$. La solución debe ser: $l' = [a, b, d, c]$ ya que $d(a, c)$ y $d(d, c)$ son uno, que es lo mínimo que se puede conseguir. Es importante ver que la distancia, al ser una ronda, es siempre el $MIN_{posicion(y) - posicion(x), longitudRonda - posicion(y) + posicion(x)}$ con x e y dos elementos cualesquiera. Luego, esta solución es la más chica ordenada de forma lexicográfica. Notar que, entonces, la suma de las distancias totales, que fue minimizada, es: $d(a, c) + d(d, c) + 0 = 2$. Finalmente, el output debe ser: $1[a, b, d, c]$ pues 1 es la distancia máxima entre cualquier par de elementos.
2. Se tiene la siguiente lista, vista como una ronda: $l = [a, b, c]$ y la siguiente relación de amistad $a = (a, c)$. La solución debe ser: $l' = [a, b, c]$ ya que $d(a, c) = 1$. Finalmente, el output debe ser: $1[a, b, c]$ pues 1 es la distancia máxima entre cualquier par de elementos y la lista es la menor de todas las que cumplen.
3. Se tiene la siguiente lista, vista como una ronda: $l = [a, b, c, d, e]$ y la siguiente relación de amistad $a = (a, b)(a, c)(a, d)(a, e)(c, e)$. La solución debe ser: $l' = 2[a, b, c, e, d]$. Notemos que si bien la lista $[a, b, d, c, e]$ cumple con ser una minimización, la misma es mayor lexicográficamente que l' . Podemos ver fácilmente que como a esta relacionada con todas, no importa el orden en el que se pongan, la suma de las distancias no cambia. Sin embargo, al pedir que (c, e) también sean amigas, obliga a tener que elegir un orden apropiado para estas dos últimas letras. Básicamente, deben ir "pegadas". Luego, el mejor lugar para ponerlas es aquella que acerque la "e" hacia la "c", para obtener la menor de las posibles soluciones.

2.2 Resolución:

Estructura:

Para resolver el problema, se decidió dividirlo en un conjunto de modulos,clases, para poder abstraer mejor los detalles que no fuesen inherentes al problema principal en si, el de la minimizacion de las distancias. En este sentido, se implementaron tres clases: Amistad,Ronda y Fogon. Este es el orden de abstraccion, es decir la ultima mencionada utiliza los servicios de las de abajo, y asi sucesivamente. A continuacion se muestra la estructura y jerarquia de las mismas. (HACER ESQUEMAS CON MIEMBROS DE LA CLASE)

Ideas centrales:

Basicamente, implementaremos un algoritmo que haga backtracking sobre las posibles formas de ordenar una ronda de exploradoras y nos quedaremos con aquella que minimice la suma de las distancias. Esta misma sera la que quedara referenciada por el campo rondaOptima de la clase. Para esto, cuando el algoritmo de backtracking haye una primera permutacion de la ronda, esta sera nuestra candidata a optima. Luego,el algoritmo ira explorando nuevas soluciones posibles. Cuando llegue a una hoja del arbol de posibilidades, es decir, cuando quede definida una ronda que es permutacion de la original se preguntara si la nueva suma de distancias es mejor. Si es asi, habremos obtenido una nueva rondaOptima. De esta forma continuaremos hasta haber inspeccionado todas las posibles soluciones que sean relevantes(que podrian no ser todas, por las posibles podas).

Mas en detalle, diremos que la clase fogon cuenta con dos funciones: **Solve** y **SentarExploradoras**, que hace el trabajo mencionado arriba. La primera recibirá una *lista(ArrayList)* de exploradoras(Character) y un conjunto de amistades(*HashSet;Amistad*). A partir de esto:

1. Ordenara la lista alfabeticamente
2. Verificara si hay amistades definidas en el conjunto.
3. Si esto no fuese cierto, la lista de exploradoras es la solucion optima. Generamos una ronda igual a dicha lista y la instanciamos en rondaOptima.
4. Si fuese cierto, y la lista de exploradoras es no vacia, definimos una rondaAuxiliar a la cual le agregamos el primer elemento de la lista de exploradoras. Tambien se remueve dicho elemento de la lista. Finalmente, se llama a la funcion SentarExploradoras(lista exploradoras, rondaAuxiliar,Tam(exploradoras)) que instanciara el miembro rondaOptima con la solucion
5. Devolvera una referencia a el miembro rondaOptima

De esta funcion podemos remarcar dos aspectos:

La funcion de **SentarExploradoras** usa, como precondition, que la lista este ordenada. La razon se vera en detalle luego, pero ,por ahora, diremos que el algoritmo de backtracking mira, en cada nivel de la recursion, las posibles elecciones de letras(exploradoras) en orden. Ademas, como la solucion optima debe estar ordenada lexicograficamente, en principio, una buena decision podria ser fijar como primer elemento de la ronda,vista como lista, a la primer exploradora(donde la primera es la menor alfabeticamente, pues ya esta ordenada). Esto evitara, en el backtracking, tener que mirar el resto de las posibles elecciones para la primer posicion y comenzar desde la siguiente.

Veamos con detenimiento el algoritmo de SentarExploradoras:

Algorithm 2: SentarExploradoras

```

Input: Lista exploradoras , Ronda rondaAux, int tam
// Verificamos si rondaAux esta completa,i.e si ya se formo una rama
1 if Tamaño(rondaAux) = tam + 1 then
    // El caso "base": Si la solucion es mejor que la optima actual o es la
    // primera ronda que se arma, pasa a ser la nueva optima:
2 if (Peso(rondaAux) < Peso(rondaOptima) or Vacia?(rondaOptima)) then
3     rondaOptima  $\leftarrow$  rondaAux
4     retornar
5 end
6 end
// Si no esta completa, es decir todavia quedan decisiones por tomar, estamos
// en un nuevo paso de la recursion y tenemos que explorar todas las
// elecciones disponibles en exploradoras, para este nivel del arbol.
7 for i < Tamaño(exploradoras) do
    // Elegimos el primero de la lista, que debe ser el elemento mas chico
    // disponible en el nivel
8 eleccion  $\leftarrow$  exploradoras[i]
9 Agregar(eleccion,rondaAux)
    // PODA1: Se aplica ya sobre el final. Si vamos a comenzar a mirar las
    // soluciones del tipo [a,ultima,...()] entonces,podamos esa rama(ver exp
    // mas adelante)
10 if Tamano(rondaAux) = 2 AND i es el ultimo indice then
11     salir del bucle
12 end
    // PODA2: Si la distancia de la ronda actual ya es peor que la otima,
    // podar rama
13 if Peso(rondaAux) >= Peso(rondaOptima) AND !Vacia?(rondaOptima) then
14     Remover(rondaAux,eleccion)
15 end
16 else
    // Si no se pudo podar, hay que explorar las posibles elecciones del
    // nivel.
17 Remover(exploradoras,eleccion)
    // Sacamos al elemento eleccion de la lista de exploradoras, para
    // reducir el espacio de elecciones del proximo nivel y garantizar que
    // no se pueda volver a agarrar mas abajo en la recursion
    // Hacemos la llamada recursiva para continuar expandiendo la rondaAux y
    // hallar una nueva solucion
18 sentarExploradoras(exploradoras, rondaAux,tam);
    // Al regresar, habremos explorado el subarbol generado a partir de
    // nuestra ultima decision
19 ultima  $\leftarrow$  RemoverUltima(rondaAux) AgregarAdelante(exploradoras,eleccion)
    // /Sacamos la ultima eleccion de la rondaAux y mantenemos el invariante
    // ,colocando el elemento al comienzo de la misma para mantenerla
    // ordenada y para que quede disponible para las subsiguientes
    // elecciones que se haran en el proximo ciclo(otra forma de verlo es
    // pensar que tiene que estar lista para las exploraciones que van a
    // hacer sus hermanos, i.e las demas posibles decisiones del mismo nivel
20 end
21 end
22 retornar

```

Es importante remarcar , ahora, algunos detalles sobre las clases de Ronda y Amistad. Sobre esta ultima, solamente basta aclarar que es una tupla que implementa un par no ordenado. Se hizo,entonces, un "override" del metodo equals para definir la igualdad como $(a,b) = (b,a)$.

En cuanto a la ronda, la misma es una clase que extiende a *ArrayList < Character >*. Solamente se le agregaron dos metodos relevantes: *distancia(int a, int b)* y *sumaDistancias(HashSet<Amistad>, amigas)*

La primera, simplemente calcula las distancias dados dos indices en la ronda. Para eso, tiene en cuenta que si la diferencia entre b y a es mayor que la mitad, la distancia debe ser: tamaño - b + a. Si no, es simplemente la resta b -a.

En cuanto a *sumaDistancias*, basicamente calcula las distancias todos contra todos de la siguiente forma:

1. Define un acumulador inicializado en cero

2. Itera la lista(ronda)

Por cada elemento que recorre, itera el resto de la lista(i.e hay dos bucles anidados).

Calcula la distancia entre el par de elementos y luego crea una instancia de amistad

Se fija si dicha amistad esta contenida en el conjunto de amigas, que recibe por parametro.

Si es asi, acumula la distancia

Si no, no hace nada

Continua el curso de los bucles

2.3 Justificación:

Vamos a asumir que las funciones de las demas clases son correctas. Razonaremos , luego, por induccion en la cantidad de exploradoras (e) y dejaremos fijo la cantidad de amistades ($a \neq 0$).

Si $n = 1$, entonces la funcion solve agregara el primer, y unico, elemento de la lista a la *rondaAuxiliar*. Luego, *SentarExploradoras(vacia, rondaAux, 0)* caera en el primer if(caso base del backtrack) pues $Tamano(vacia) = tam + 1 = 0 + 1$ Entonces, entrara al siguiente if, ya que la *rondaOptima* se encuentra vacía. De esta forma se instancia *rondaOptima* con *rondaAux* y se retornara a la funcion solve, la cual devolvera *rondaOptima*. Es facil ver que, entonces, la solucion es correcta.

Supongamos que $n > 1$. Es claro que no entrara en el primer if, pues *rondaAux* tiene un solo elemento. Luego, supongamos que estamos en la i esima iteracion del ciclo que explora las posibles elecciones. Siguiendo el algoritmo: se agrega el i esimo elemento de la lista de exploradoras a la *rondaAux*, que tiene $n-1$ elementos(pues inicialmente, el backtrack se llama sin el primer elemento de la lista). El invariante del ciclo garantiza que la lista este en orden y que , dentro del mismo nivel del backtrack, el tamaño va a ser $n - 1$. Luego, asumamos que no hay ninguna poda para hacer. Entonces, se remueve la i esima exploradora de la lista(ahora con $n-2$ elementos). Luego, por hipotesis inductiva, *SentarExploradoras(exploradoras, rondaAux, tam)* es correcta y por lo tanto explora todo el arbol de decision posible con la eleccion que acabamos de hacer y se tiene en *rondaOptima* la mejor ronda hasta el momento. Luego, hay que reestablecer el invariante. Para eso, quitamos de la *rondaAux* a la ultima eleccion, y la volvemos a agregar en la lista adelante de todo, para mantener el orden. Luego, la lista vuelve a tener $n - 1$ elementos. Cuando lleguemos a la iteracion final($n - 1$) habremos inspeccionado correctamente todo el arbol de decision para cada eleccion de la lista de $n - 1$ elementos. Luego, como miramos todas las posibles combinaciones, habremos encontrado la *rondaOptima*.

¿Que hubiese pasado en caso de haberse efectuado una poda? Si en alguna iteracion hubiesemos caido en la primer poda, se habrian perdido posibles soluciones. Sin embargo, se puede ver que aquellas que se descartan ,en este caso, son rondas que se obtienen de mirar "al reves" las que ya se vieron. Por lo tanto no aportan un nuevo espacio de soluciones. Si en alguna iteracion hubiesemos caido en la segunda poda, significaria que aquella ronda no era mejor que la optima. Luego, como agregando nuevas elecciones no podria mejorar, es seguro podar.

2.4 Complejidad:

Como hay mas de una clase, podemos comenzar por analizar la complejidad de las mas pequeñas. En el caso de la clase Ronda, la complejidad de los metodos es la misma que los de ArrayList, pues es una extension. Para los definidos por nosotros, se puede observar que la complejidad de la funcion `distancia(int a, int b)` es $O(1)$, pues solo realiza comparaciones de enteros, operaciones aritmeticas basicas y asignaciones. El metodo `int sumaDistancias(HashSet<Amistad> amistad)` recorre la lista subyacente por cada elemento en ella. Es decir:

$$\sum_{i=0}^n O(i)O(a) = \sum_{i=0}^n O(i * a) = O(\sum_{i=0}^n i * a) = O(n^2 * a)$$

Notar que en cada iteraciones del ciclo interior se llama a la funcion `contains()` del HashSet de Java. Como la misma esta implementada sobre una tabla de Hash, aunque , en promedio, la operacion sea $O(1)$, el peor caso es $O(a)$. La funcion `contains()` utiliza el metodo `hashCode` y `equals` de la clase `Amistad` para poder determinar si el elemento en cuestion esta en el conjunto. Ambas, se puede ver, son $O(1)$.

Determinemos ahora, la complejidad de `SentarExploradoras`. Primero, notemos que se utilizan algunos metodos propios de la clase ArrayList(y Ronda). A saber, los mismos son: `size()`, `add()`, `remove()` y `get()`. Segun la documentacion de Java: "The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires $O(n)$ time. All of the other operations run in linear time (roughly speaking)." Para el analisis, se tendra en cuenta el peor caso de `add()`, que es $O(n)$.

Centrandonos en el algoritmo, veamos cuantas "pasos" toma realizarlo. Dada una lista de exploradoras e y un conjunto de amistades a , el algoritmo trabaja con $e - 1$ elementos. A partir de esto, podemos pensar el algoritmo de backtracking como un arbol de decision, donde luego de colocar el primer nodo raiz, elegimos entre $e - 1$ posibilidades(hijos), por cada uno de ellos, restan $(e - 2)$ elecciones, y asi siguiendo. De esta forma, la cantidad de hojas del arbol es $(e - 1)!$. Cada hoja representa la cantidad de posibles rondas que se armaron. Sin embargo, esta no es la cantidad de pasos que realiza el algoritmo. Todavia falta agregarle los nodos internos, que fueron pasos previos para obtener las soluciones. Luego, la cantidad de nodos en un arbol completo exactamente e -ario es $N = \sum_{i=0}^h e^i = \frac{e^{(h+1)} - 1}{e - 1}$. Si t es la cantidad de hojas en el arbol, $e^h = t$, por lo tanto $N = \frac{e \cdot t - 1}{(e - 1)}$.

Como nuestro arbol esta "acotado" por un arbol exactamente e -ario, diremos que $N =$ cantidad de pasos $= O(\frac{e \cdot t - 1}{e - 1}) = O(\frac{e \cdot (e - 1)! - 1}{e - 1}) = O(\frac{e! - 1}{e - 1}) = O(e!)$. Luego, falta ver cual es el "peso" de estos pasos. En cada iteracion del ciclo se hacen: `add()` y `remove()` sobre la lista, que es $O(2e) = O(e)$. `add()` y `remove()` sobre la ronda, que es $O(2e) = O(e)$. Se pregunta si hay que hacer la `PODA1`, $O(1)$ pues `size()` lo es. Se pregunta si hay que hacer la `PODA2`, llamando a `sumaDistancia` de la `RondaAux` y `sumaDistancia` de la `RondaOptima`, ambas con el conjunto a . Luego, esto es, por lo visto, $O(2e^2a) = O(e^2a)$

Finalmente la cota de complejidad, teorica, resulta $O(e!e^2a)$. Veamos que esto es $O(e^e a^2)$ Probamos por induccion que $e!e^2a < c(e^e)a \quad \forall e \geq 1, c = 4$ y un a fijo.

$$Sie = 1, \quad e!e^2a = a < 4(e^e) = 4a$$

$$Sie > 1, \quad (e+1)!(e+1)^2a = e!(e+1)^2(e+1)a = (e!(e^2) + 2e(e!) + e!)(e+1)a < 3e!(e^2)(e+1)(a) < (porhi) \quad 3e^e(e+1)(a) = (3e^{(e+1)} + 3 * e^e)a < 4(e+1)^{(e+1)}a.$$

2.5 Casos de test:

TO DO

2.6 Tiempo de ejecucion y gráficos:

TO DO

2.7 Partes relevantes del código