

Practica

01

CURSO 2017-2018

<Divide y Vencerás>

FRANCISCO JAVIER MARQUÉS GAONA

JUAN JOSÉ MÉNDEZ CONTRERAS

JOSE ANDRÉS GONZÁLEZ VENTURA

ESTRUCTURA DATOS Y ALGORITMOS II

2º GRADO EN INGENIERÍA INFORMÁTICA



Departamento de Informática
Universidad de Almería

ÍNDICE

- Presentación del problema	Páginas 3 - 5
- Generación de los juegos de prueba	Página 6
- Clase Linea	Página 7
- Clase ComprobarLineas	Páginas 7 - 8
- Diagrama de clases	Página 8
- Análisis teórico del tiempo del algoritmo DyV	Página 9
- Representación en gráficas de los tiempos del algoritmo DyV para todos los casos	Páginas 10 - 13
- Precondiciones y Postcondiciones	Páginas 13 - 14

PRESENTACIÓN DEL PROBLEMA

Hemos conseguido un contrato de soporte informático por parte de la “Metropolitan Transport Authority” (NAMTA) de la expansión urbana de *New Almería* ubicada cerca de Boise, estado de Idaho. NAMTA es responsable de la gestión y mantenimiento del sistema de transporte de la ciudad, incluyendo el metro, metro ligero, ferries y autobuses. El objeto del contrato es dar soporte informático a la gestión del sistema de transporte en dicha expansión urbana, que funciona de forma independiente como una ciudad autónoma (dentro del área metropolitana de Boise).

La expansión urbana de New Almeria tiene una estructura en cuadrícula de calles y avenidas, donde las avenidas van de sur a norte y las calles de oeste a este, numeradas según se muestra en la figura 1; la estructura está ubicada dentro de una isla, en la que existen dos puntos especiales, “Embarkment” (**Em**) y “Central” (**Ce**). Según se muestra en la figura. Como simplificación vamos a considerar que la cuadrícula es totalmente rectangular (en el mundo real, dada la forma de la isla podría no serlo, lo que supondría que algunas “manzanas/bloques” no existirían).

En nuestro caso, como primer trabajo se nos va a pedir el desarrollo de diverso software para el correcto funcionamiento del sistema de metro y metro ligero de la ciudad. La estructura del metro viene dada en la figura 2, donde podemos visualizar una línea principal con tres ramales (líneas 1a, 1b y 1c), una línea circular (línea 2) y cuatro líneas transversales (líneas 3-6), según se ve en la figura.

En nuestro caso se ha utilizado la tecnología (ya un poco obsoleta en la realidad, pero que vamos a considerar se ha usado para simplificar los problemas) de uso de catenaria con cierre de circuito por un “feeder” negativo por tercera vía. Se trata de una extrema simplificación de los sistemas reales en que existen sub-estaciones para aislar tramos de alimentación y estabilizar la corriente, así como evitar riesgos. En la figura 3 se muestra el esquema para una línea tipo (se utiliza el ejemplo de la línea 6).

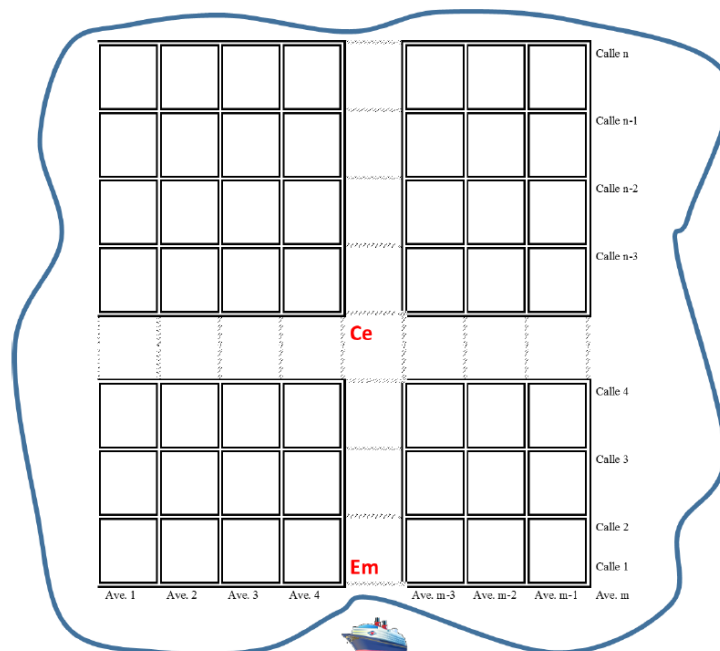


Figura 1. Damero simplificado de New Almería, m avenidas y n calles. Hay dos puntos especiales, **Em**, el embarcadero en que llegan los ferries a la ciudad y **Ce**, el punto central del damero urbano.

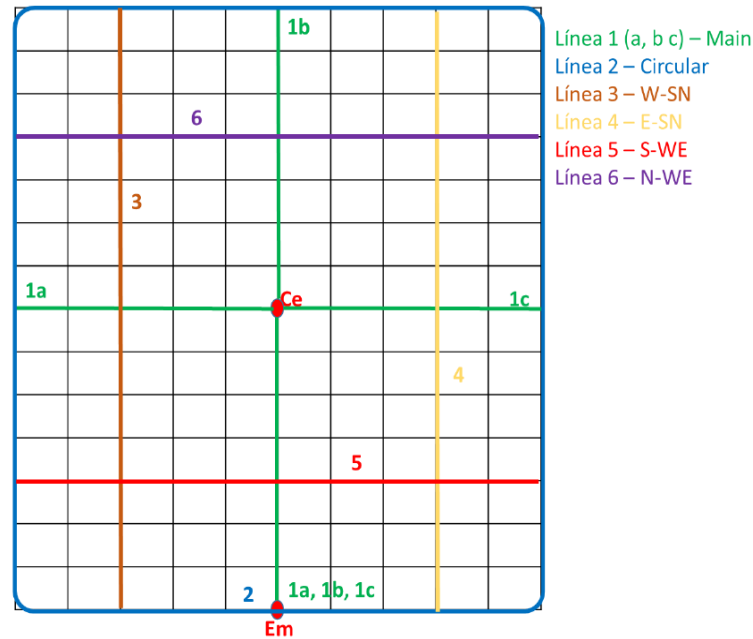


Figura 2. Red de metro y metro ligero de Nueva Almería, con sus distintas líneas.

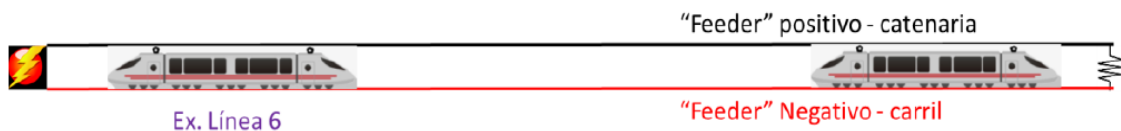


Figura 3. Alimentación por catenaria activa y cierre por carril negativo de una línea de la NAMTA, caso de la línea 6. Se utiliza la tecnología de catenaria con tercer carril de retorno, voltaje constante (600 V DC). Nota. Éste es el sistema utilizado en New York, por ejemplo.

El primer problema que tendremos que resolver será detectar las roturas de la catenaria de la forma más eficiente posible. Para ello disponemos de un sistema de "cierre" de circuito situado cada cierta distancia (cada 100 m, por ejemplo) en la línea que permite comprobar si el circuito se cierra, si la rotura está situada posteriormente a un cierre, el circuito se activa (cierra), en caso contrario no se hay corriente (no se activa). Ver figura 4, en la que se han supuesto n cierres de circuito intermedios.

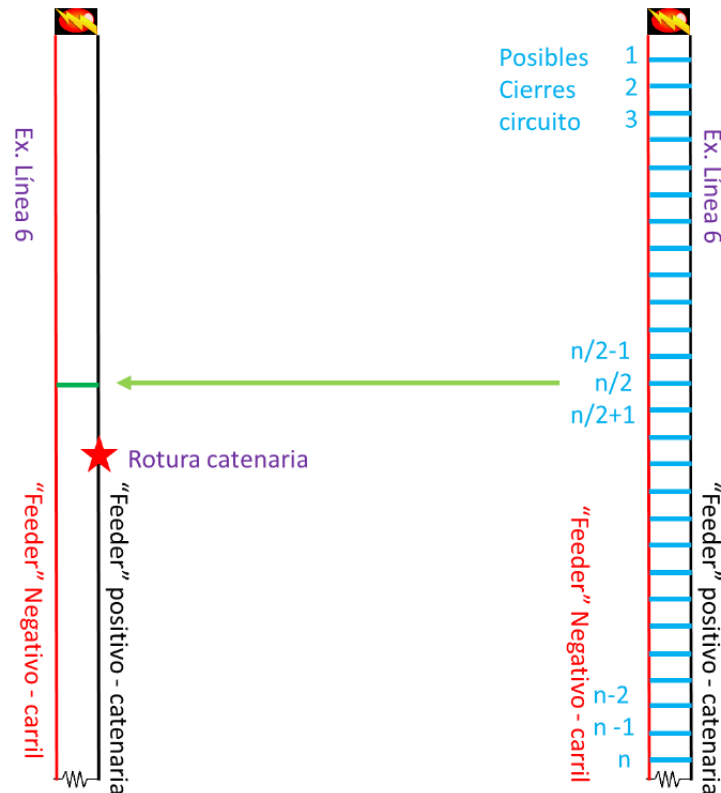


Figura 4. Ejemplo con la línea 6 en la que hay n posibles cierres del circuito (se cortocircuitan en un poste se sujeción de la catenaria los dos "feeders"). En el ejemplo, como al cerrar en la posición $n/2$ el circuito se activa, significa que la rotura es posterior a dicho "cierre $n/2$ ".

En nuestro caso dada una línea con un número de cierres (postes) n , se desea detectar entre qué dos postes está la rotura de la catenaria. Se ha de trabajar desarrollando un software que ordena realizar un cierre, recibe la respuesta de si el circuito se cierra o no y va comprobando hasta encontrar el punto de rotura de la catenaria.

En este caso, se trabajará con unos datos "en memoria" que representarán la situación de la línea (punto de rotura de la catenaria), indicando si los cierres activan el circuito o no; en esta situación suponemos que hay una única rotura de la catenaria (se supone que estadísticamente la probabilidad de rotura simultánea en dos o más puntos es despreciable). El resultado final es una indicación al equipo de reparaciones de ir al tramo (numerado por el poste, y cierre, correspondiente) para solventar el problema.

Para trabajar, se supone que se activa un método, `close(i)`, que activa el cierre i y devuelve si hay corriente (cierra el circuito) o no en dicho cierre; este método activa en la realidad el dispositivo de cierre y analiza el resultado; en nuestro caso se lista la orden y se indica el resultado; se da el resultado en función de la información de roturas del supuesto de trabajo almacenada convenientemente para los distintos casos de prueba.

GENERACIÓN DE LOS JUEGOS DE PRUEBA

En primer lugar dado la presentación del problema era necesario implementar una clase que se encargue de generar juegos de prueba, tal y como se define en el guion, generándose entre 2000 y 10000 postes de forma aleatoria. Cada juego de prueba generado contiene el número total de postes, la distancia de postes y si pasa corriente en cada uno de ellos.

La clase `GeneradorPruebas` contiene 3 métodos de generación de juegos de pruebas:

- ***Pruebas aleatorias (`GenerarAleatorio`)***

Este método genera un total de 20 casos de prueba, cada uno de ellos denominado como línea *i.txt*. A cada fichero se le añade el nombre de la línea, la distancia entre postes que se genera de forma aleatoria entre 25 y 100 metros, además del número total de postes. Finalmente, se le añade cada uno de los postes generados con su índice (0,1,2...) y la corriente asociada al lado, 0 si no pasa corriente o 1 si pasa.

- ***Pruebas de mejor caso (`GenerarMejorCaso`)***

Según nuestro criterio, una línea debe tener al menos 2000 postes para considerarla como tal. Dicho esto, el mejor caso posible sería cuando solo hayan esos 2000 postes, pues se realizaría un menor número de divisiones o llamadas recursivas.

Por otro lado, si siempre hubiera de forma generalista el mismo número de postes, podríamos aseverar que el mejor caso sería siempre cualquier número impar dentro del rango de postes, ya que haría n divisiones, mientras que cualquier número par, haría $n+1$ divisiones.

- ***Pruebas de peor caso (`GenerarPeorCaso`)***

En nuestro caso se genera un archivo en el cual por todos los postes pasa la corriente, de tal forma que el algoritmo no es capaz de dilucidar el error realizando el número máximo de divisiones o llamadas recursivas hasta que se produce una excepción al superar la longitud del `arrayList`.

Adecuándonos al planteamiento del problema, suponiendo que el algoritmo tenga que encontrar el error, éste estaría situado en uno de los dos extremos de la línea ya que efectuaría el mayor número de llamadas recursivas posible.

CLASE LINEA

Nuestra clase Linea tiene un total de 5 atributos, nombreLinea (String), distancia (double), numeroPostes(int), longitud(double), postes (ArrayList-postes).

CONSTRUCTOR:

En cuanto al constructor, recibe el nombre del fichero que generamos con anterioridad. Asignamos a cada atributo los valores de las líneas de dichos ficheros.

METODOS:

- *Close(int i)*: Este método devuelve el estado del poste i, es decir, si hay o no hay corriente.
- *EncontrarError(int ini, int fin)*: Basándonos en el esquema de divide y vencerás de división, realiza una búsqueda binaria hasta encontrar el poste por cual no pasa corriente siempre que su predecesor sí tenga corriente.

```

56 private Poste encontrarError(int ini, int fin) {
57     if (ini >= fin) {
58         //if (postes[ini].close())
59         if (close(ini))
60             return postes.get(ini+1);
61         return postes.get(ini);
62     }
63     int med = (ini + fin) / 2;
64     //if (postes[med].close()) {
65     if (close(med)) {
66         return encontrarError (med + 1, fin);
67     }
68     return encontrarError(ini, med-1);
69 }

```

CLASE COMPROBARLINEAS

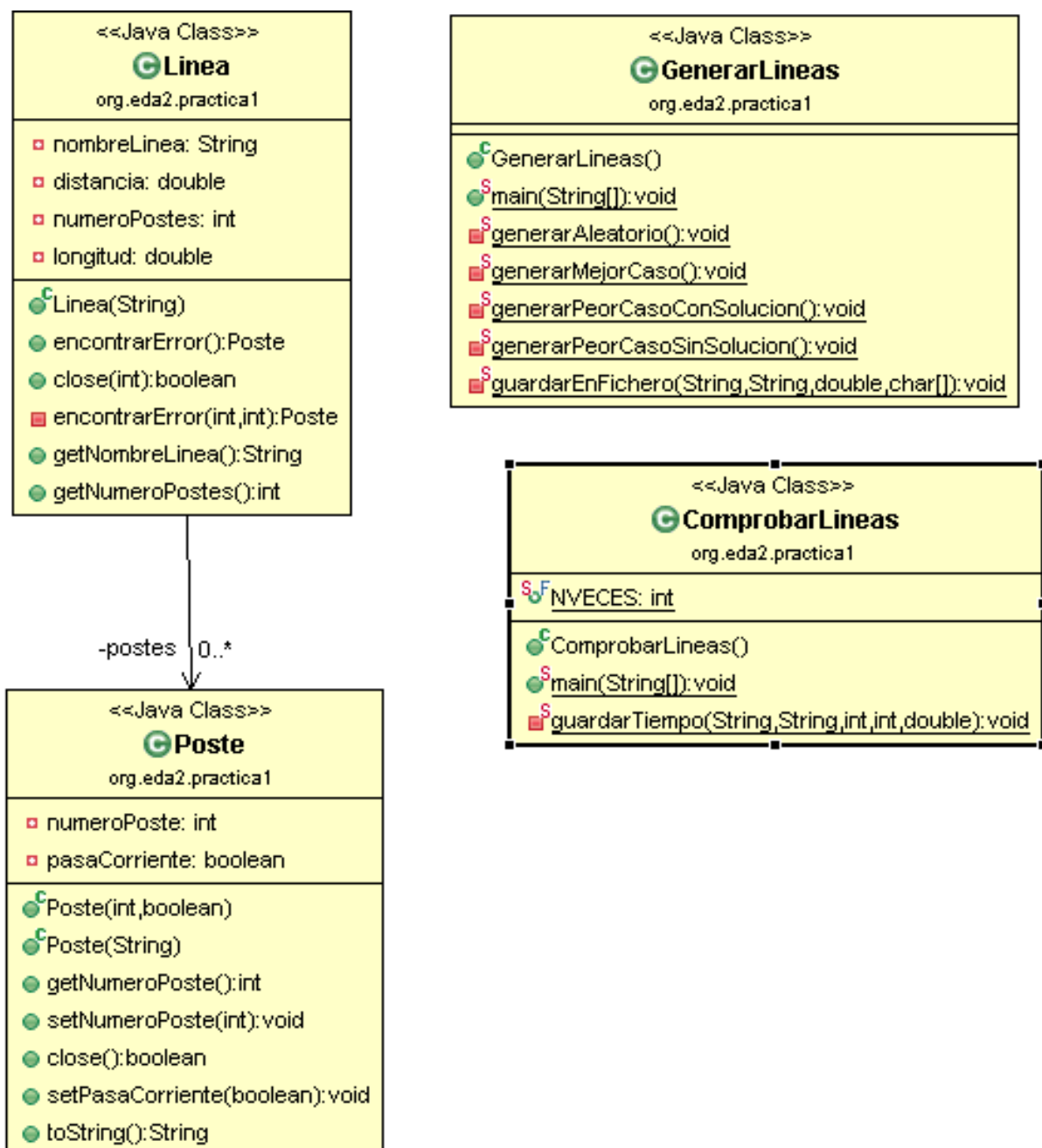
Nuestra clase ComprobarLineas se ha creado con el propósito de calcular los tiempos de nuestro algoritmo donde se realizan 10 repeticiones para obtener el tiempo medio.

Muestra por pantalla cada una de las líneas generadas con sus respectivos números de postes, el poste de error y el tiempo medio de las 10 ejecuciones. Además, tiene también un método denominado *Guardartiempos()* que almacenará cada uno de los tiempos de cada línea en su respectivo fichero, posteriormente utilizado para la realización de las gráficas de cada uno de los casos.

<terminated> ComprobarLineas [Java Application] C:\EclipseNeon2\GIPP2017EclipseNeon2\jdk1.8.0_65\bin\javaw.exe (12 abr. 2018 15:01:01)			
Línea 1	2034	385	10263
Línea 2	8865	1292	9176
Línea 3	3778	2919	10263
Línea 4	3892	101	4407
Línea 5	6370	5044	3924
Línea 6	9021	1801	6339
Línea 7	7157	95	4588
Línea 8	5613	824	4890
Línea 9	8227	4655	6097
Línea 10	3637	114	4225
Línea 11	4828	4426	4950

Problems	Javadoc	Declaration	Console
<terminated> ComprobarLineas [Java Application] C:\EclipseNeon2\GIPP2017EclipseNeon2\jdk1.8.0_65\bin\javaw.exe (12 abr. 2018 15:01:01)			
Línea 13	2374	1603	4950
Línea 14	9666	8655	5131
Línea 15	5329	1392	5071
Línea 16	7323	5069	5675
Línea 17	6775	2003	5192
Línea 18	3874	1273	5493
Línea 19	9872	2436	6459
Línea 20	2430	360	4588
MejorLinea	2000	1000	3682
PeorLineaConSolucion	9556	9555	4105

DIAGRAMA DE CLASES



ANÁLISIS TEÓRICO DEL TIEMPO DEL ALGORITMO DYV

Guiándonos en las diapositivas del tema 1 de Análisis Algorítmico, el tiempo de nuestro algoritmo DyV debe resolverse mediante división, tal y como se muestra en la siguiente imagen.

■ Resolución del problema mediante **división**:

$$t(n) = cn^k$$

Para $0 \leq n < b$

$$t(n) = at(n/b) + cn^k$$

En $n \geq b$

➤ Solución:

$t(n) \in$	$\Theta(n^k)$ $\Theta(n^k \log n)$ $\Theta(n^{\log_b a})$	si $a < b^k$ si $a = b^k$ si $a > b^k$
------------	---	--

Donde:

- a es el n° de llamadas recursivas
- b es el n° por el cual dividimos el tamaño original del problema (n)
- k sería la constante polinómica de las operaciones no recursivas

Por lo tanto tenemos en nuestro caso 1 llamadas recursivas, el tamaño del problema lo dividimos en 2 y la parte no recursiva sería de orden constante.

Tendríamos $a = 1$; $b = 2$ y $k = 0$.

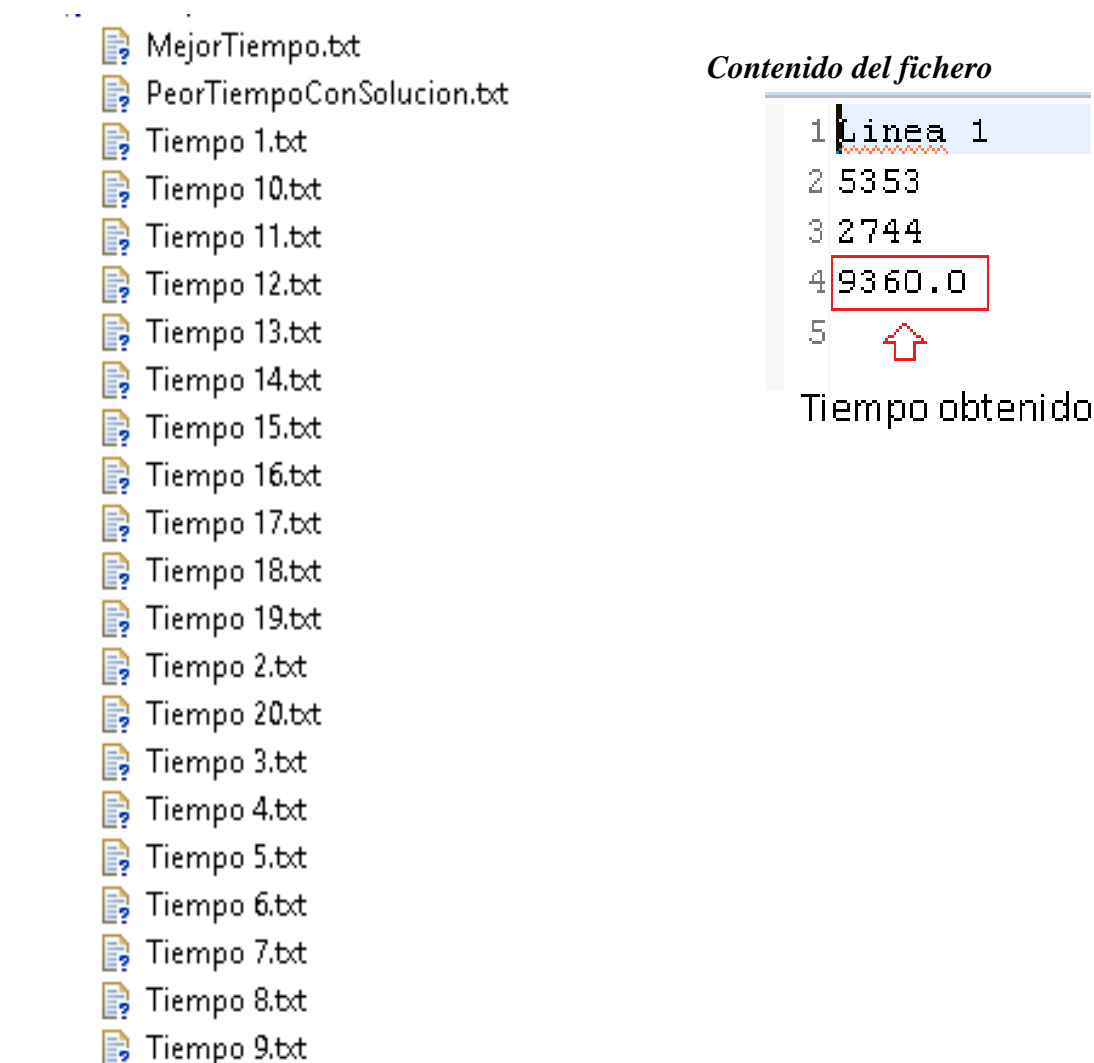
Como $a = 1$ y $b^k = 2^0 = 1$, por lo que $a = b^k$, finalmente:

$$t(n) \in \Theta(n^k \log n) = \Theta(\log n)$$

Tanto en los casos aleatorios como en el mejor y el peor caso $t(n) \in \Theta(\log n)$

REPRESENTACIÓN EN GRÁFICAS DE LOS TIEMPOS DEL ALGORITMO DYV PARA TODOS LOS CASOS

Realizamos un total de diez ejecuciones para todos los posibles casos (mejor caso, peor y casos aleatorios) midiendo el tiempo medio de cada caso. Cada tiempo se recoge en un fichero mostrando el nombre de la línea sobre la que se ha realizado su medición, el nº de postes, el postes de error y el tiempo medio de las diez ejecuciones en ns.



The image shows a file explorer window with a list of files on the left and a preview of the content of 'MejorTiempo.txt' on the right.

Files in the directory:

- MejorTiempo.txt
- PeorTiempoConSolucion.txt
- Tiempo 1.txt
- Tiempo 10.txt
- Tiempo 11.txt
- Tiempo 12.txt
- Tiempo 13.txt
- Tiempo 14.txt
- Tiempo 15.txt
- Tiempo 16.txt
- Tiempo 17.txt
- Tiempo 18.txt
- Tiempo 19.txt
- Tiempo 2.txt
- Tiempo 20.txt
- Tiempo 3.txt
- Tiempo 4.txt
- Tiempo 5.txt
- Tiempo 6.txt
- Tiempo 7.txt
- Tiempo 8.txt
- Tiempo 9.txt

Contenido del fichero (MejorTiempo.txt):

```

1 Linea 1
2 5353
3 2744
4 9360.0
5

```

Below the preview, the text "Tiempo obtenido" is displayed.

A continuación se muestran la gráficas con los tiempos medios de las diez ejecuciones de cada uno de los ficheros para todos los casos y su correspondiente gráfica. Además se añade la línea de tendencia de cada uno de ellos.

Gráfica casos aleatorios

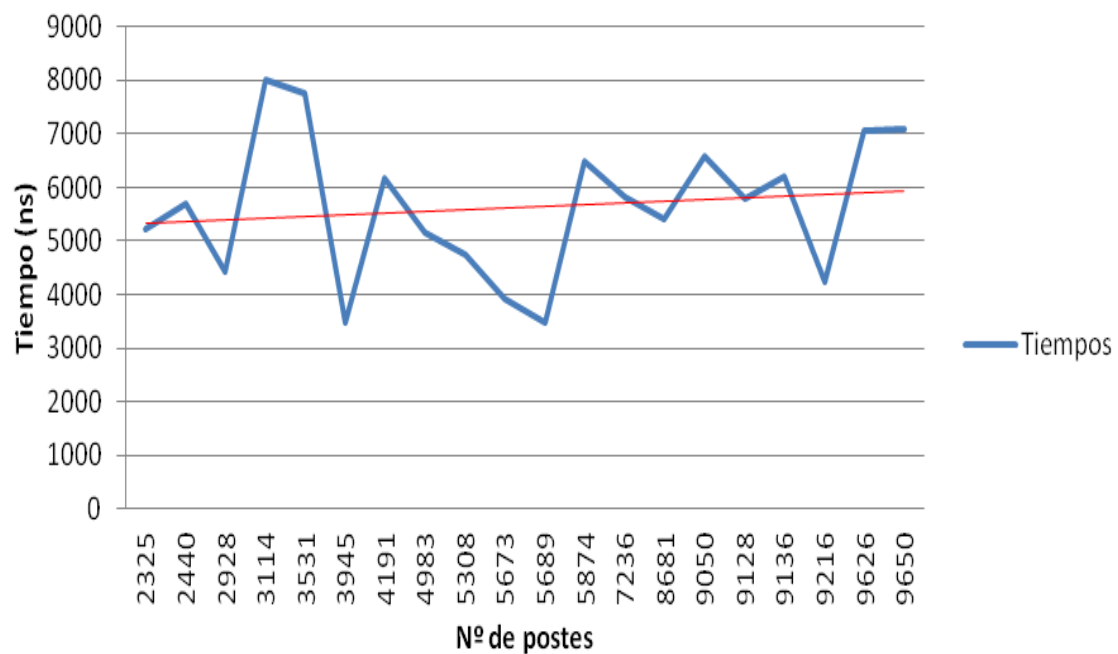
1	Fichero	Tiempo	Nº Postes
2	7	4187	3391
3	6	4146	3597
4	4	3900	3727
5	1	3201	4324
6	5	4146	4376
7	18	7512	4481
8	13	5008	4516
9	11	4639	5231
10	20	9360	5353
11	17	6198	5430
12	3	3900	6205
13	19	8210	6478
14	8	4228	6710
15	2	3653	6764
16	10	4351	7040
17	15	5419	7776
18	12	4802	8350
19	16	5501	8596
20	14	5213	8700
21	9	4329	8959



Gráfica peores casos

1	Fichero	Tiempo	Nº Postes
2	17	5213	2325
3	10	5706	2440
4	5	4433	2928
5	18	8005	3114
6	13	7759	3531
7	2	3489	3945
8	8	6158	4191
9	15	5172	4983
10	1	4762	5308
11	20	3928	5673
12	3	3489	5689
13	6	6486	5874
14	11	5829	7236
15	19	5419	8681
16	14	6568	9050
17	12	5788	9128
18	9	6198	9136
19	4	4228	9216
20	16	7060	9626
21	7	7102	9650

Tiempos peores casos



Gráfica mejores casos

1	Fichero	Tiempo	Nº Postes
2	1	3161	2000

En cuanto al mejor caso como siempre se genera de antemano un fichero con un total de 2000 postes, aunque generemos veintes ficheros distintos, el tiempo sería semejante en cada caso, solo variaría según el uso de la cpu en cada momento.

PRECONDICIONES Y POSTCONDICIONES

➤ Precondiciones:

- *Comprobar Lineas(Main):*
 - Inicializará un objeto de tipo línea que recibirá como parámetro en el constructor la ruta a un archivo.txt
 - El fichero solo puede contener números enteros, decimales o strings.
 - El fichero no debe contener ninguna línea vacía.
 - Solo leerá un número de líneas igual a nº de postes de cada línea además del nombre de la línea, la distancia entre postes, y el total de postes.
 - La primera línea del documento debe ser el nombre de la línea
 - La segunda línea del documento debe ser la distancia entre postes.
 - La tercera línea del documento debe ser el total de postes.
 - El resto de líneas indican el nº del poste y si pasa o no corriente
 - Cada fichero se comprobará un total de 10 veces calculando el tiempo medio de las 10 ejecuciones.
- *Encontrar error(DyV):*
 - Recibe como parámetro de entrada dos enteros que indican la posición del poste inicial y final de la línea.
 - La línea debe contener al menos un poste, aunque por lógica hemos asumido que una línea debe tener un nº de postes mínimo de 2000 si no, no tendría sentido crear una línea que vaya a recorrer una distancia tan corta de tan solo un poste.
 - No se contempla el hecho tal y como se especifica en el guión de que exista un poste sin corriente y a continuación los siguientes vuelvan a tener corriente hasta encontrarse otro poste sin corriente. Es decir el algoritmo busca el primer poste sin corriente y todos los demás no tienen tampoco.
- *Generar Lineas:*
 - El nº de postes varía entre 2000 y 10000.
 - La distancia de postes varía entre 25 y 100.

- La posición de error del poste se calcula como:
 $Math.random()*numeroPostes + 1$
- Se generan un total de 41 casos de prueba, 20 casos de prueba aleatorios(total de postes generado al zar), 20 peores casos de prueba aleatorios(total de postes generado al azar) y un solo mejor caso.
- A cada poste se le asigna 1 si pasa corriente o 0 si no pasa corriente según la posición de error calculada previamente.

➤ Postcondiciones:

- *Comprobar Lineas(Main):*
 - Muestra por pantalla que se han generado los archivos de tiempo con éxito, si no muestra un mensaje de error.
 - Debe generar tantos archivos de tiempo como ficheros leídos.
 - La primera línea del archivo de tiempo corresponderá al nombre de la Línea.
 - La segunda línea del archivo de tiempo corresponderá al total de postes de la línea.
 - La tercera línea del archivo de tiempo corresponderá al poste de error.
 - La cuarta y última línea del archivo de tiempo corresponderá al tiempo que tarda en algoritmo en calcular el poste de error.
- *Encontrar error(DyV):*
 - Encontrar error se ejecutará correctamente siempre y cuando exista un poste por el cual no pasa corriente si no dará una excepción y mostrará el mensaje: “No se ha encontrado error”
 - Debe encontrar solo un poste de error.
 - Devolverá el poste de error indicando su posición.
 - Se puede dar el caso de que el poste de error pueda ser el último o bien el inicial.
- *Generar Lineas:*
 - Se almacenan los ficheros con un nombre distinto específico indicado por nosotros al recorrer un bucle for.
 - Muestra el mensaje creado con éxito o bien error en la creación según el fichero se haya creado con éxito o no.
 - Los ficheros se almacenan en un directorio(package) denominado “datosP1”