Estrategias de Programación y Estructuras de Datos

Grado en Tecnologías de la Información

Curso 2014-2015

Memoria de la práctica

Francisco Javier Crespo Jiménez

fcrespo14@alumno.uned.es

Móvil: 687 768 000

Índice de contenido

1 Introducción.

La práctica de la asignatura "Estrategias de Programación y Estructuras de Datos" correspondiente al curso académico 2014-2015. ha consistido en el desarrollo de un sistema de sugerencias del estilo de los utilizados para el autocompletado de formularios utilizados, entre otros, por motores de búsqueda tales como Google, Yahoo o Bing.

Para ello se ha utilizado el lenguaje de programación Java, y como base para los TAD o estructuras de datos se han utilizado las proporcionadas por el Equipo Docente de la asignatura.

En el enunciado de la práctica se facilitó un interfaz y se propusieron dos aproximaciones:

- La primera basada en una lista de consultas ordenadas alfabéticamente.
- La segunda basada en un árbol de caracteres.

El trabajo ha consistido en llevar a cabo los siguientes pasos:

- Contestar a las preguntas propuestas en el enunciado de la práctica reflexionando sobre las consecuencias de cada una de las cuestiones planteadas.
- Desarrollo en Java de ambas aproximaciones.
- Realización de baterías de pruebas empíricas para contrastar los resultados esperados teóricamente con los obtenidos en dichas pruebas.

Durante el desarrollo en Java de ambas implementaciones se ha tratado de buscar tanto la claridad de código (separando las clases en paquetes, creando métodos privados y documentando todos los métodos), tratando de conseguir la mayor eficiencia posible y comprobando la salida generada por la aplicación con el juego de pruebas y el comprobador de la salida proporcionados por el Equipo Docente de la asignatura.

El material generado a entregar se compone de los siguientes ítems:

- Memoria de la práctica (este documento).
- Archivos de los juegos de prueba comprimidos en un archivo .zip
- Archivo eped.jar para poder ejecutar las pruebas sin necesidad de compilar.
- Códigos fuente de la aplicación junto con las librerías con los TAD proporcionados por el equipo docente y archivo de proyecto en BlueJ. El archivo README incluido en el proyecto, contiene instrucciones para facilitar la ejecución de pruebas.

El entorno en el que ha sido probada la aplicación con el juego de pruebas facilitado por el Equipo Docente ha sido:

- Sistema Operativo Windows 10 – 64 bits.
- Java versión "1.8.0_60".
- BlueJ versión 3.1.4.

Se ha detectado que para la implementación basada en listas, resulta necesario ampliar la pila con el parámetro -Xss64m.

2 Primera aproximación: lista de consultas ordenadas alfabéticamente.

Pregunta 1.1

- *¿Cómo depende el tamaño de almacenamiento del número de consultas diferentes en el registro de consultas?*

  Dado que cada nueva consulta no existente previamente debe crear una nueva query que debe ser introducida en el depósito de consultas, el tamaño de almacenamiento en la lista ordenada alfabéticamente resulta linealmente proporcional del número de consultas diferentes introducidas en el depósito de consultas.

- *¿Cómo depende el número de repeticiones?*

  Respecto al número de repeticiones, este no altera el tamaño de almacenamiento, ya que en el caso de cambio de frecuencia de cada query, únicamente se modifica su atributo de frecuencia permaneciendo el tamaño del almacenamiento invariable.

Pregunta 1.2

- *¿Cómo depende el tiempo de localizar todas las posibles sugerencias de búsqueda del número de consultas diferentes?*

  El tiempo de localizar las posibles sugerencias para un prefijo dado, tiene una dependencia linealmente proporcional del número de Queries almacenadas en el depósito dado que se deben recorrer todas las queries comparando el prefijo con cada una de las queries.

- *¿Y del número de repeticiones?*

  El número de repeticiones no tiene ninguna repercusión en el tiempo necesario para localizar las sugerencias, aunque si que lo tiene a la hora de ser ordenadas por frecuencia y devueltas en ese orden.

- *¿Y de la longitud máxima de las consultas medida como el número de caracteres de su texto?*

  La longitud máxima de las consultas no tiene influencia en el tiempo necesario para ser localizadas. Sin embargo, si que existiría una dependencia con respecto al tamaño en número de caracteres del prefijo introducido para localizar las sugerencias.

- *¿Y del tamaño del conjunto de caracteres permitido?*

  En esta implementación, no se considera que el tamaño del conjunto de caracteres tenga influencia alguna sobre el tiempo necesario para localizar las sugerencias con la presunción de que realizar una comparación entre distintos caracteres tiene el mismo coste.

- *Razona en términos del coste asintótico temporal en el caso peor que se podría conseguir para el método* ***listOfQueries*** *con este diseño.*

Considerando que el caso peor consistiría en que todas las consultas en el depósito tengan una raíz común y se nos introduzcan esa raíz como prefijo de búsqueda. Esto haría que tuviésemos que devolver todo el depósito completo.

Por lo tanto en primer lugar se debe recorrer la lista completa con un coste O(N) buscando todas las coincidencias. Una vez obtenida la lista de queries coincidentes, se debe considerar también el coste de la operación de ordenación teniendo en cuenta que se ha seleccionado sortMerge como algoritmo de ordenación. El coste del proceso de ordenación resulta de orden O(N log (N)).

A la vista de lo anterior, podemos concluir que el coste del método listOfQueries es O(max(N, N log (N))) o, simplificando la expresión, O(N log (N)).

Pregunta 1.3

- *Consideremos una estructura de datos alternativa en la que disponemos de una lista de consultas similar a la considerada por cada carácter inicial (es decir, tendríamos una lista con todas las consultas que empiezan por el carácter 'a', otra con las que empiezan por 'b', etc.) ¿Cuánto se podría reducir el tiempo de búsqueda en caso peor?*

  Para esta pregunta consideraremos el siguiente caso como caso peor: supondremos que todas las listas de los caracteres permitidos están vacías excepto una, pongamos por caso la lista de las queries que comienzan por la letra 'z' y a su vez la búsqueda se pretende realizar sobre la última query de la lista de la letra 'z'. Para este caso, el coste sería idéntico tanto en esta alternativa como en la original por lo que, respondiendo a la pregunta realizada, no se reduciría el tiempo de búsqueda en el caso peor.

- *¿Cambiaría el coste asintótico temporal?*

  Como se ha explicado el la respuesta anterior, considero que el coste asintótico temporal no cambiaría para el caso peor. Teniendo en cuenta el ejemplo anterior como caso peor y considerando el número de caracteres permitidos, por tanto el número de "sublistas" existentes tan sólo se trataría de una constante divisora que resulta despreciada a la hora de realizar el coste asintótico temporal.

Pregunta 1.4

- *¿Se ajusta este diseño a los requisitos del problema? Explica por qué.*

  Este nuevo diseño no cumple con el enunciado de la primera aproximación si lo entendemos en sentido estricto de "lista de consultas ordenadas alfabéticamente". Se trataría mas bien de una "lista índice de listas de consultas ordenadas alfabéticamente".

  No obstante, a pesar de lo indicado anteriormente si antendemos solamente a lo indicado en el apartado 1 del enunciado de la práctica, se podría decir que a nivel funcional este diseño continúa cumpliendo el objetivo final de proporcionar una lista ordenada por frecuencia descendente, de las consultas compatibles con una consulta incompleta.

3 Segunda aproximación: árbol de caracteres.

Pregunta 2.1

- *¿Cómo depende el tamaño de almacenamiento del número de consultas diferentes en el registro de consultas?*

  No es posible proporcionar una respuesta exacta a esta pregunta ya que obviamente, el tamaño del almacenamiento crecerá si crece el número de consultas diferentes sin embargo el crecimiento en esta aproximación depende de a partir de cuando empieza a ser una consulta diferente a otra. Utilizando alguno de los ejemplos del enunciado de la práctica, si en nuestro árbol existe la query "casa" e introducimos "caso" , sólo se generarán 2 subárboles nuevos: el subárbol correspondiente al primer carácter distinto entre ellas (o) y el subárbol cuya raíz mantiene la frecuencia.

  Abundando en el ejemplo, si ahora añadimos una nueva query "cosa", se generarán como hijos del árbol con raíz en "c" 4 subárboles: 3 con los caracteres a partir del primer carácter distinto (o, s, a), más el subárbol cuya raíz mantiene su frecuencia. Por lo tanto, a diferencia de la primera aproximación donde se creaba una nueva Query, en esta aproximación no se puede determinar con exactitud cual será el crecimiento del almacenamiento.

- *¿Y del tamaño máximo de las consultas?*

  Por el mismo motivo expuesto en la respuesta anterior, no se puede determinar el crecimiento exacto por el tamaño máximo de las consultas. La única aproximación intuitiva que podremos hacer es que, independientemente del tamaño de las mismas, se creará un subárbol por cada carácter a partir del primero no existente previamente más un subárbol hoja que contendrá la frecuencia de la consulta. Por lo tanto, el tamaño será mayor cuantas más consultas distintas en sus primeros caracteres pretendamos almacenar. Tomando un ejemplo concreto, podemos saber que en el caso de consultas que tuvieran un tamaño máximo de 10 caracteres, por ejemplo, la profundidad máxima del árbol sería de 12 niveles teniendo en cuenta la raíz y la hoja con la frecuencia.

- *¿Y del tamaño del conjunto de caracteres permitido?*

  Consideremos un conjunto de 50 caracteres y un tamaño máximo de consulta de 10 caracteres. Compara el tamaño máximo del árbol en la segunda aproximación con el tamaño máximo de la lista en la primera aproximación.

  Para el cálculo del tamaño del conjunto de caracteres permitido en el caso de las listas, se ha determinado que se trata de variaciones con repetición de 50 elementos tomados de 10 en 10 por lo tanto: VR_{50}^{10} de modo que el tamaño máximo para listas será 50^{10} consultas diferentes que se almacenarán en la lista de consultas. En el caso de los árboles el tamaño máximo sería $1+\sum_{i=1}^{10} 50^{i}$ donde se suma 1 (la raíz del árbol) a 50 que es el número de caracteres elevado al sumatorio de i que representará los niveles máximos de profundidad (en este caso 10). Para este segundo caso, la cifra representa en número de caracteres máximo que podrán contener el árbol teniendo en cuenta el conjunto proporcionado.

  Resulta conveniente recalcar que aunque se han comprado los tamaños máximos de ambas aproximaciones, en el primer caso se ha calculado el número máximo de consultas

distintas y en segundo caso, se trata de nodos o subárboles.

¿La diferencia se agrandará o se reducirá para conjuntos de caracteres mayores y consultas más largas?

El crecimiento del árbol de caracteres de la segunda aproximación será sensiblemente inferior al de las listas. De manera informal podemos pensar que en la implementación basada en árbol de caracteres las nuevas consultas compartirán, en mayor o menor medida, parte de la consulta y solamente se crearán los subárboles a partir del momento en que sean diferentes a las consultas ya introducidas en el depósito, en la implementación basada en lista, cualquier nueva consulta creará una nueva query aún cuando la diferencia sea mínima con respecto a otra ya existente en el depósito previamente. Por tanto podemos concluir que la diferencia de agrandará para conjuntos de caracteres mayores y consultas más largas.

Pregunta 2.2

- ¿Cómo depende el tiempo de localizar **una** posible sugerencia de búsqueda del número de consultas diferentes?

  Si bien parece claro que cuanto mayor es el número de consultas existentes en el árbol de caracteres mayor será el tiempo necesario para localizar una posible sugerencia, incluso tratándose del mismo árbol, el tiempo será superior en caso de buscar la sugerencia en unas ramas del árbol con mayor densidad de consultas que en otra zona del árbol donde la búsqueda tenga menos caracteres que recorrer en cada nivel.

- ¿Y del número de repeticiones?

  En este caso, el número de repeticiones no tiene ninguna influencia sobre el tiempo de localizar una posible sugerencia.

- ¿Y de la longitud máxima de las consultas medida como el número de caracteres de su texto?

  Este factor si que hace que haya una dependencia directamente del número de caracteres de su texto ya que por cada carácter debe hacer un recorrido entre la lista de hijos buscando la coincidencia y, en su caso, descendiendo un nivel en el árbol de caracteres.

- ¿Y del tamaño del conjunto de caracteres permitido?

  También el conjunto de caracteres permitido influye directamente en el tiempo necesario para buscar una coincidencia entre un conjunto mayor de caracteres. El tamaño del conjunto de caracteres determinará el número máximo de "hijos" que puede tener cada árbol.

Pregunta 2.3

- ¿Cómo depende el tiempo de localizar **todas** las posibles sugerencias de búsqueda del número de consultas diferentes?

  Del mismo modo que se indicó para el caso de localizar una posible sugerencia, a mayor número de consultas mayor será el tiempo en localizarlas dependiendo de la densidad en la zona del árbol donde se realice la búsqueda a este tiempo habrá que añadirle el tiempo de crear la lista con todas las posibles sugerencias y el tiempo de ordenar la lista de

resultados en función de la frecuencia.

- ¿Y del número de repeticiones?

  De nuevo nos encontramos conque el número de repeticiones o frecuencia de las consultas en el depósito no tiene influencia si hablamos estrictamente de localizarlas. Es decir, componer la lista de sugerencias no tiene en cuenta sus frecuencias de modo que no tiene ninguna repercusión. Sin embargo, el método que devuelve esas posibles sugerencias, las ordena por frecuencia de modo que habrá que estimar el tiempo que invierta el algoritmo de ordenación.

- ¿Y de la longitud máxima de las consultas medida como el número de caracteres de su texto?

  Dado que por cada carácter debe hacer un recorrido entre la lista de hijos buscando las consultas coincidentes, el tiempo necesario se vé directamente influido por la longitud máxima de las consultas que determinará el número de niveles a descender en el árbol.

- ¿Y del tamaño del conjunto de caracteres permitido?

  También en este caso, el tiempo necesario tiene una dependencia del conjunto de caracteres permitido ya que conocemos que el tiempo de buscar una coincidencia entre un conjunto mayor de caracteres no pude ser el mismo que entre un conjunto de caracteres más reducido. Este tamaño determinará el número de hijos sobre los que habrá que realizar la búsqueda.

- Razona en términos del coste asintótico temporal en el caso peor que se podría conseguir para el método **listOfQueries** con este diseño.

  Vamos a considerar el siguiente caso como el peor para esta implementación: Supongamos que nuestro árbol tiene un carácter inicial común a todas las consultas. Utilizando el ejemplo del enunciado de la práctica, supongamos que en el árbol tenemos las consultas casa, caso y casos. El prefijo que nos introducen para buscar la lista de queries coincidentes es el carácter 'c'. De este modo, tenemos que recorrer todo el árbol y crear la lista de coincidencias con un coste O(N). Posteriormente esta lista debe ser ordenada por frecuencias, y tal y como se ha indicado para la implementación basada en lista, el método sort con la variante sortMerge imprime un coste O(N log (N)) finalmente tendremos que devolver el máximo de ambos costes siendo O(max(N, N log (N))) y de nuevo simplificando la expresión el coste obtenido resulta ser de orden O(N log (N)).

Pregunta 2.4

- A la vista de las respuestas a las preguntas anteriores y de los requisitos de nuestro problema, ¿consideras éste un diseño más adecuado para **QueryDepot**?

  Aunque en términos de coste asintótico temporal los resultados puedan parecer idénticos y dada la improbabilidad del caso peor, informalmente pensando en los casos medios este diseño parece mucho más acertado teniendo en mente el tiempo de ejecución. Máxime cuando el problema exceda de un determinado tamaño en el que la implementación basada en lista ordenada dejará de ser más eficiente que la implementación basada en árbol de caracteres.

Pregunta 2.5

- Compara el coste de encontrar todas las sugerencias posibles con el coste de ordenarlas por frecuencia (consideremos que el coste de ordenación en el caso peor sea del orden O(n·log(n)).

  El tiempo de encontrar las posibles sugerencias ha resultado ser menor – O(N) -, que el coste de ordenación – O(N log (N)) -, siendo el coste de ordenación el que ha determinado el coste total del método listOfQueries.

  Pensando de modo práctico, cuanto menor sea la lista de sugerencias encontradas, menor será la influencia del coste de ordenación. De modo contrario, en tanto la lista de sugerencias sea de un tamaño más parecido al depósito completo, mas cerca nos encontramos del caso peor y mayor será la influencia del coste de ordenación, que como ya hemos comentado previamente es O(N log (N)).

- ¿Sería aconsejable comenzar a realizar sugerencias antes incluso de que el usuario comience a teclear, con este diseño? ¿Por qué?

  Precisamente, abundando en la respuesta anterior, si tuviésemos que comenzar a elaborar una lista de sugerencias antes de que el usuario comenzase a teclear deberíamos elaborar una lista de sugerencias con todo el contenido del depósito ya que a priori no podemos descartar ninguna posibilidad lo que nos haría estar en el caso peor.

4 Estudio empírico del coste temporal.

Para la realización del estudio empírico del coste temporal, se han utilizado unos archivos de prueba generados con palabras del diccionario castellano sin repeticiones.

Tratando de conseguir una localización sencilla, los archivos siguen la siguiente nomenclatura para su nombrado:

- La primera letra es '**c**' u '**o**' dependiendo de si el contenido del archivo son las **consultas** en si mismas o las **operaciones** a realizar en el depósito.
- Los siguientes caracteres, es el tamaño del depósito. Si la cantidad tiene el sufijo 'k' se refiere a miles de consultas. Éste varia desde una única consulta hasta las 20.000 en incrementos que se describirán posteriormente.
- Existe un archivo de operaciones llamado opeor.txt que busca el caso peor y obtener con el método listOfQueries todo el depósito de consultas.
- Los archivos tienen extensión .txt

De este modo el archivo de consultas más pequeño será c1.txt y el mayor c20k.txt; los respectivos archivos de operaciones son nombrados como: o1.txt y o20k.txt respectivamente.

Para una mejor visualización de los resultados se han efectuado baterías de pruebas en 4 rangos concretos de resultados. Al tratarse de 4 escalas completamente distintas, las mediciones se han efectuado por separado para poder graficarlos de manera independiente y poder apreciar mejor la trazada que cada uno de ellos genera.

Independientemente del tamaño del depósito de consultas y para una mejor medición de tiempos, cada operación se realiza un número de veces determinado con la constante "*REPETICIONES*" que por defecto está fijada en un valor de 10.000. Siguiendo el procedimiento descrito en el punto 3.4 del enunciado de la práctica se realiza la medición de tiempos con una precisión inferior a la milésima de segundo.

Para tratar de homogeneizar los resultados se ha llevado a cabo también la siguiente estrategia: Siempre se ha buscado la frecuencia de la última consulta introducida en el depósito y la lista de sugerencias también se ha obtenido con el mínimo número de caracteres para obtener la última consulta introducida como sugerencia. La intención de mostrar solamente una sugerencia ha sido la de tratar de obviar el tiempo de ordenación de las sugerencias obtenidas y evidenciar la diferencia entre ambas implementaciones.

Finalmente se ha realizado una quinta y última batería de pruebas, en la que se ha buscado el efecto contrario: con un tamaño moderado de entre 1.000 y 20.000 consultas almacenadas en el depósito, se ha buscado el caso peor: que el método listOfQueries tuviese que devolver todo el depósito como posibles sugerencias y el coste de ordenación sea el que determine el tiempo del algoritmo "igualando" ambas implementaciones.

En algunos casos de tamaños de depósito superiores a las 15.000 consultas se ha detectado la necesidad de ampliar la pila de ejecución pasando el parámetro -Xss16m a la máquina virtual Java para evitar el desbordamiento de la pila.

4.1 Rango 1 a 10 consultas.

Se pretende comenzar el estudio, con los tamaños mínimos del depósito de consultas de modo que la primera batería de pruebas contendrá un depósito de entre 1 a 10 consultas. Se hace notar que a pesar de contar con una precisión de microsegundos (10^{-6} segundos), en los primeros tamaños del depósito no se obtienen medidas apreciables por lo que se dejan a 0.

Consultas (unidades)	1	2	3	4	5	6	7	8	9	10
getFreqQuery (Lista)	0,0000	0,0000	0,0000	0,0012	0,0012	0,0011	0,0011	0,0013	0,0015	0,0017
getFreqQuery (Árbol)	0,0035	0,0083	0,0089	0,0053	0,0071	0,0130	0,0086	0,0083	0,0091	0,0104
	1	2	3	4	5	6	7	8	9	10
listOfQueries (Lista)	0,0000	0,0010	0,0013	0,0015	0,0013	0,0018	0,0020	0,0021	0,0022	0,0016
listOfQueries (Árbol)	0,0055	0,0082	0,0077	0,0061	0,0077	0,0096	0,0077	0,0062	0,0065	0,0081

Tabla 1: Tiempos obtenidos en milisegundos para el rango de 1 a 10 consultas

Como se puede comprobar en los siguientes gráficos, el aspecto más significativo que se puede destacar en este rango de tamaños es que el tiempo que necesita la implementación basada en el árbol de caracteres, es siempre superior a la que necesita la lista ordenada para unas entradas de muy pequeño tamaño.

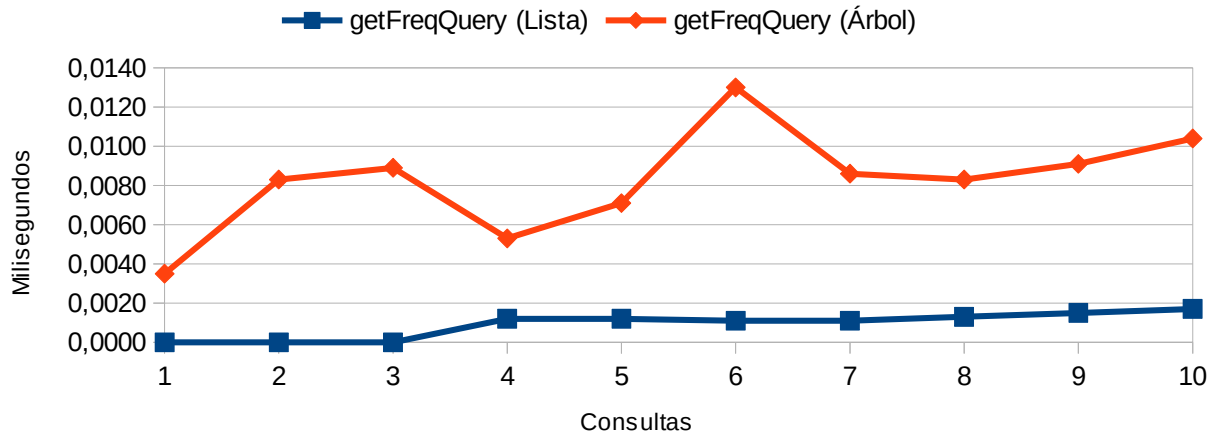


Figura 1: Gráfico de tiempos de ejecución del algoritmo getFreqQuery para rango 1 a 10 consultas.

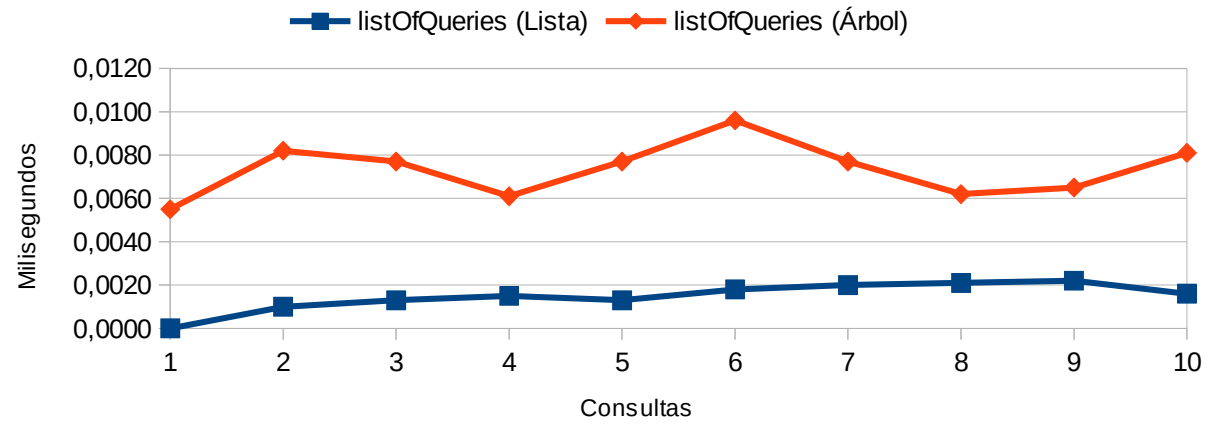


Figura 2: Gráfico de tiempos de ejecución del algoritmo listOfQueries para rango 1 a 10 consultas.

4.2 Rango 10 a 100 consultas.

Se procede a continuación con una segunda batería de pruebas con un rango de tamaño de almacén de consultas 10 veces superior a la primera batería de pruebas.

Consultas (unidades)	10	20	30	40	50	60	70	80	90	100
getFreqQuery (Lista)	0,0017	0,0022	0,0025	0,0027	0,0029	0,0044	0,0038	0,0043	0,0047	0,0043
getFreqQuery (Árbol)	0,0104	0,0098	0,0117	0,0118	0,0126	0,0102	0,0135	0,0151	0,0137	0,0133
	10	20	30	40	50	60	70	80	90	100
listOfQueries (Lista)	0,0016	0,0033	0,0039	0,0039	0,0045	0,0054	0,0061	0,0067	0,0078	0,0075
listOfQueries (Árbol)	0,0081	0,0077	0,0098	0,0101	0,0082	0,0094	0,0084	0,0087	0,0079	0,0105

Tabla 2: Tiempos obtenidos en milisegundos para el rango de 10 a 100 consultas

De los datos obtenidos se pueden sacar la clara conclusión de que para este tamaño de problema, todavía resulta más eficiente la implementación basada en listas ordenadas de queries.

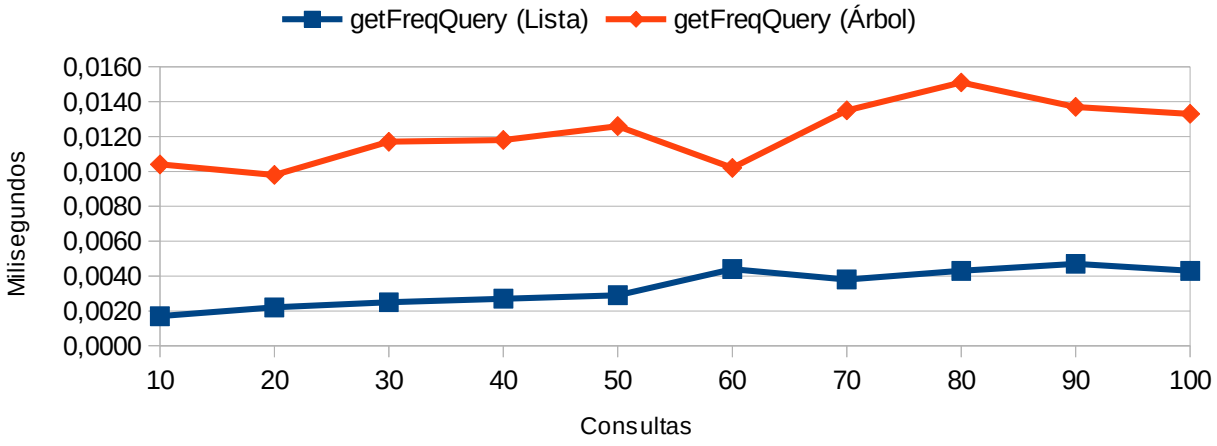


Figura 3: Gráfico de tiempos de ejecución del algoritmo getFreqQuery para rango 10 a 100 consultas.

Destacar que la implementación basada en listas del algoritmo listOfQueries al no tener que realizar ningún trabajo de ordenado, comienza a mostrar más claramente un comportamiento lineal, llegando incluso a converger cuando el tamaño del depósito alcanzó el tamaño de 90 consultas.

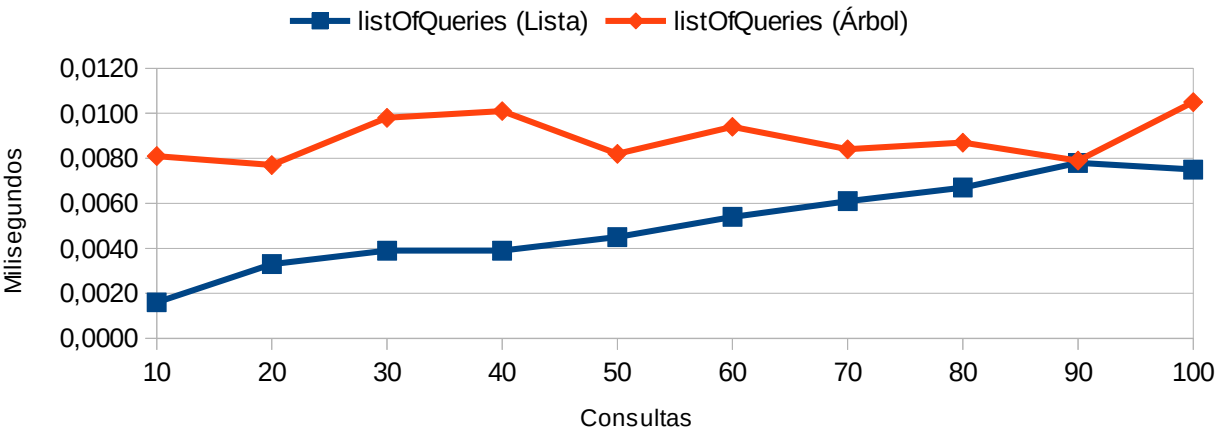


Figura 4: Gráfico de tiempos de ejecución del algoritmo listOfQueries para rango 10 a 100 consultas.

4.3 Rango 100 a 1.000 consultas.

En esta tercera batería de pruebas de un rango 10 veces superior a la anterior, se espera obtener el punto de convergencia que nos permita determinar a partir de que tamaño del depósito de consultas empieza a ser más eficiente la implementación basada en el árbol de caracteres.

Consultas (unidades)	100	200	300	400	500	600	700	800	900	1000
getFreqQuery (Lista)	0,0043	0,0070	0,0085	0,0113	0,0125	0,0151	0,0156	0,0187	0,0210	0,0232
getFreqQuery (Árbol)	0,0133	0,0114	0,0139	0,0125	0,0111	0,0138	0,0150	0,0157	0,0107	0,0140
	100	200	300	400	500	600	700	800	900	1000
listOfQueries (Lista)	0,0075	0,0101	0,0124	0,0166	0,0176	0,0209	0,0230	0,0270	0,0303	0,0330
listOfQueries (Árbol)	0,0105	0,0078	0,0117	0,0098	0,0093	0,0133	0,0154	0,0172	0,0129	0,0150

Tabla 3: Tiempos obtenidos en milisegundos para el rango de 100 a 1.000 consultas.

El algoritmo getFreqQuery comienza a converger a partir de las 400 consultas en el depósito manteniendo un comportamiento parecido a la implementación basada en lista entre las 400 y las 800 consultas comenzando a distanciarse a partir de ese punto.

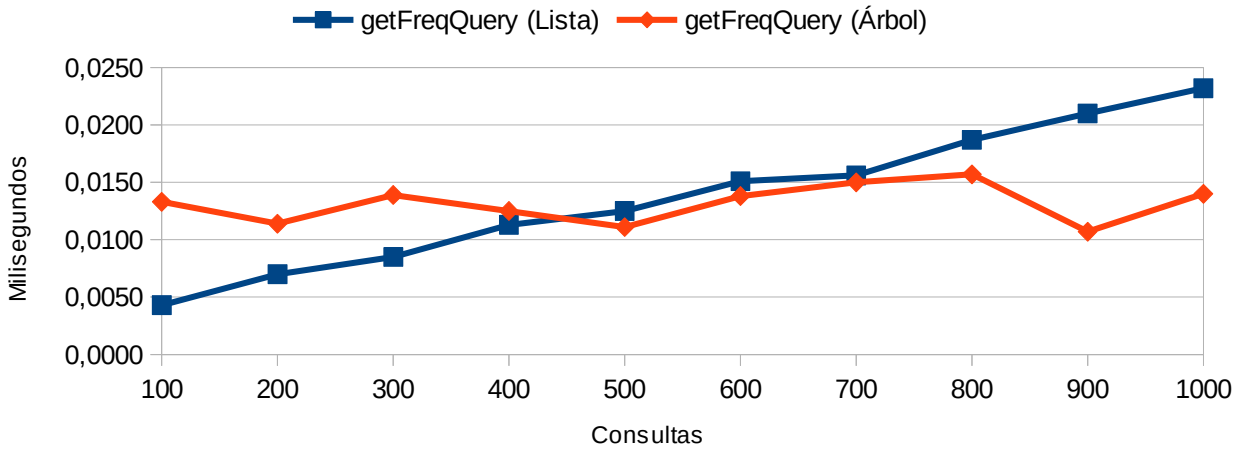


Figura 5: Gráfico de tiempos de ejecución del algoritmo getFreqQuery para rango 100 a 1.000 consultas.

Algo parecido ocurre con el algoritmo listOfQueries, aunque este converge antes de las 200 consultas permaneciendo todo el resto de la prueba con unos tiempos inferiores la implementación basada en árbol de caracteres que su homónima basada en lista ordenada.

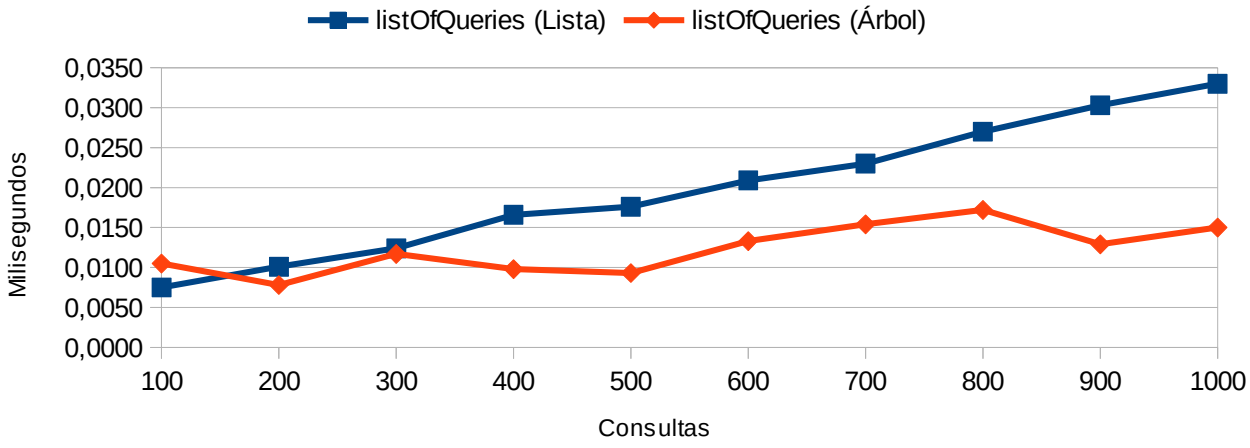


Figura 6: Gráfico de tiempos de ejecución del algoritmo listOfQueries para rango 100 a 1.000 consultas.

4.4 Rango de 1.000 a 20.000 consultas.

En esta cuarta batería de pruebas se busca confirmar que el crecimiento lineal de la aproximación basada en lista ordenada que comenzó a apreciarse en las anteriores baterías de pruebas se hace aún más patente en rangos mayores de consultas.

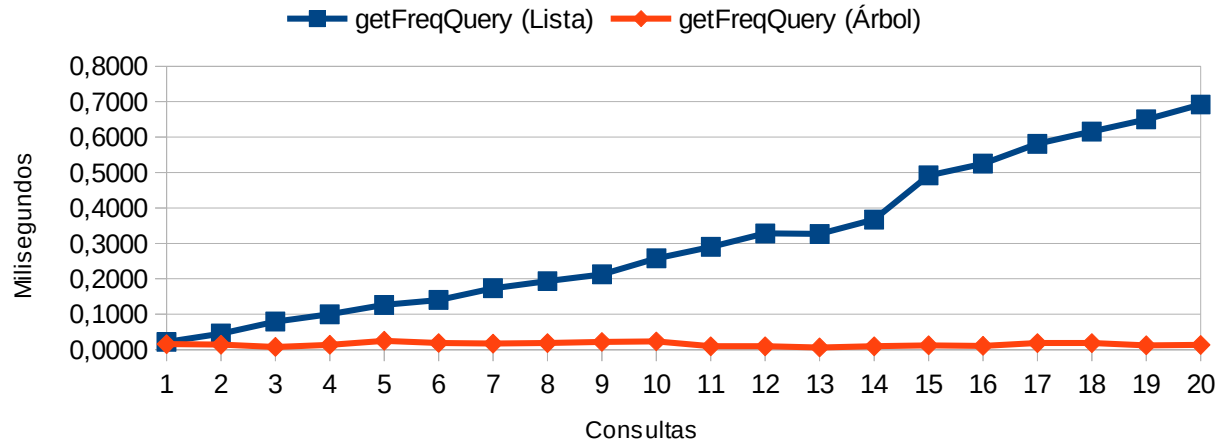
Esta batería consta de 20 mediciones ya que se ha buscado evidenciar, y exagerar si se prefiere, la distancia que van obteniendo ambas funciones comparando las implementaciones basadas en listas con las correspondientes basadas en árboles de caracteres.

Consultas (miles)	1	2	3	4	5	6	7	8	9	10
getFreqQuery (Lista)	0,0219	0,0453	0,0793	0,1000	0,1265	0,1403	0,1735	0,1937	0,2126	0,2578
getFreqQuery (Árbol)	0,0156	0,0141	0,0078	0,0140	0,0253	0,0188	0,0172	0,0187	0,0219	0,0234
	1	2	3	4	5	6	7	8	9	10
listOfQueries (Lista)	0,0718	0,0635	0,0937	0,1157	0,1483	0,1672	0,2062	0,2282	0,2625	0,2832
listOfQueries (Árbol)	0,0282	0,0203	0,0312	0,0312	0,0661	0,0468	0,0594	0,0500	0,0559	0,0532

Tabla 4: Tiempos obtenidos en milisegundos para el rango de 1.000 a 10.000 consultas

11	12	13	14	15	16	17	18	19	20 Consultas (miles)
0,2901	0,3282	0,3265	0,3672	0,4921	0,5249	0,5812	0,6157	0,6500	0,6920 getFreqQuery (Lista)
0,0094	0,0094	0,0062	0,0094	0,0125	0,0110	0,0187	0,0187	0,0124	0,0134 getFreqQuery (Árbol)
11	12	13	14	15	16	17	18	19	20
0,3078	0,3453	0,3782	0,4297	0,4613	0,4947	0,5609	0,5813	0,6465	0,6813 listOfQueries (Lista)
0,0406	0,0219	0,0297	0,0297	0,0312	0,0609	0,0453	0,0625	0,0422	0,0565 listOfQueries (Árbol)

Tabla 5: Tiempos obtenidos en milisegundos para el rango de 11.000 a 20.000 consultas



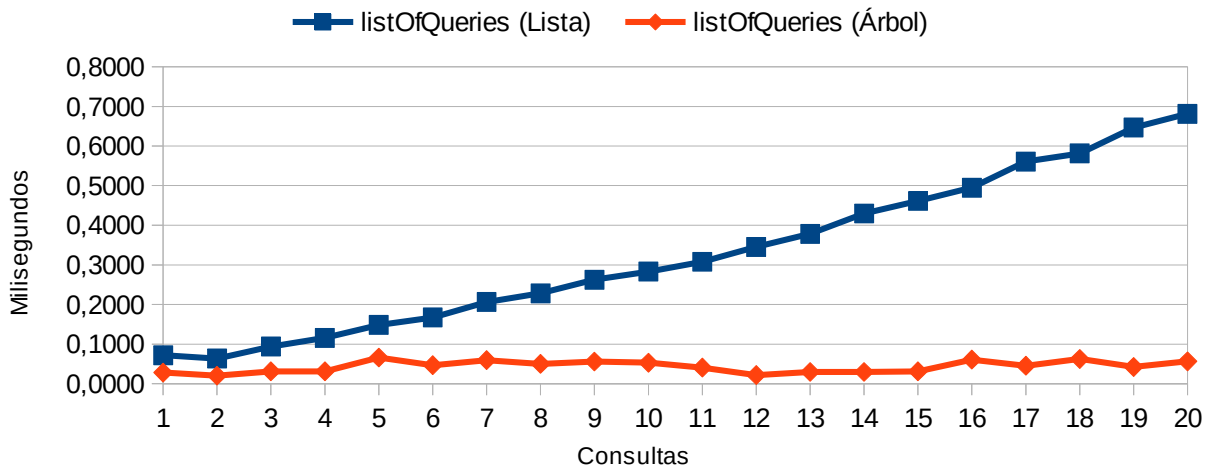


Figura 8: Gráfico de tiempos de ejecución del algoritmo listOfQueries para rango 1.000 a 20.000 consultas.

Con esta batería de pruebas ha quedado demostrado que a partir de determinado tamaño de problema y sin la influencia del algoritmo seleccionado para la ordenación de resultados el crecimiento de ambas funciones getFreqQuery y listOfQueries es similar. En el caso de la implementación basada en árbol de caracteres resulta patente que es una implementación muy eficiente para tamaños de problema moderados a grandes. Por el contrario, la implementación basada en lista ordenada, continúa teniendo un crecimiento cuasilineal haciendo que a partir de terminado tamaño de problema, resulte ineficiente y por consiguiente inaceptable desde el punto de vista del rendimiento.

4.5 Rango de 1.000 a 20.000 consultas, caso peor para listOfQueries.

Como se explicaba en la introducción al estudio empírico, se ha llevado a cabo una última batería de pruebas con los mismos tamaños de depósito que la batería de pruebas 4.4. Si bien en la prueba anterior los datos obtenidos muestran la diferencia entre las implementaciones basadas en lista ordenada y árbol de caracteres, en este caso se busca el efecto contrario: se provoca el caso peor que consiste en que el método listOfQueries devuelva todo el depósito como posibles sugerencias y sea el coste de ordenación sea el que determine el tiempo total del algoritmo de modo que se igualen los tiempos de ambas implementaciones. Para el desarrollo de esta prueba se ha seleccionado el archivo "opeor.txt" como archivo de operaciones.

CASO PEOR	1	2	3	4	5	6	7	8	9	10
listOfQueries (Lista)	18,1	57,9	121,1	218,2	342,0	502,6	696,8	923,0	1188,7	1584,4
listOfQueries (Árbol)	32,4	76,8	147,4	235,9	370,2	532,6	739,0	956,5	1221,2	1617,3

Tabla 6: Tiempos obtenidos en milisegundos para el rango de 1.000 a 10.000 consultas.

11	12	13	14	15	16	17	18	19	20	CASO PEOR
1941,0	2327,4	2753,8	3195,3	3762,1	4354,6	4924,0	5604,0	5988,0	6690,0	listOfQueries (Lista)
1860,1	2221,2	2688,4	3078,1	3605,4	4173,0	4720,6	5296,3	6009,6	6685,3	listOfQueries (Árbol)

Tabla 7: Tiempos obtenidos en milisegundos para el rango de 11.000 a 20.000 consultas.

Los tiempos obtenidos reflejados aún más claramente en la figura 9 evidencian el objetivo buscado: El coste de ordenar los resultados por frecuencia ha conseguido que la diferencia entre ambas implementaciones se haga inapreciable comparado con el tiempo necesario para su ordenación que en ambos casos se trata del algoritmo sortMerge.

Hablando estrictamente de tiempos, se puede comprobar además que los tiempos necesarios para un tamaño relativamente pequeño de datos (en torno a las 8.000 consultas) ya alcanzan los 1.000 milisegundos lo cual ya es fácilmente apreciable por el usuario del sistema de sugerencias.

Esto no hace sino ratificar, tal y como se ha explicado en la última subpregunta de la pregunta 2.5, no es posible comenzar a generar sugerencias antes de que el usuario comienza a teclear un prefijo. En el caso de un depósito de simplemente 20.000 consultas tardaríamos prácticamente del orden de 7 segundos en poder ofrecer dichas sugerencias haciéndolo desde el punto de vista de la eficiencia, inaceptable.

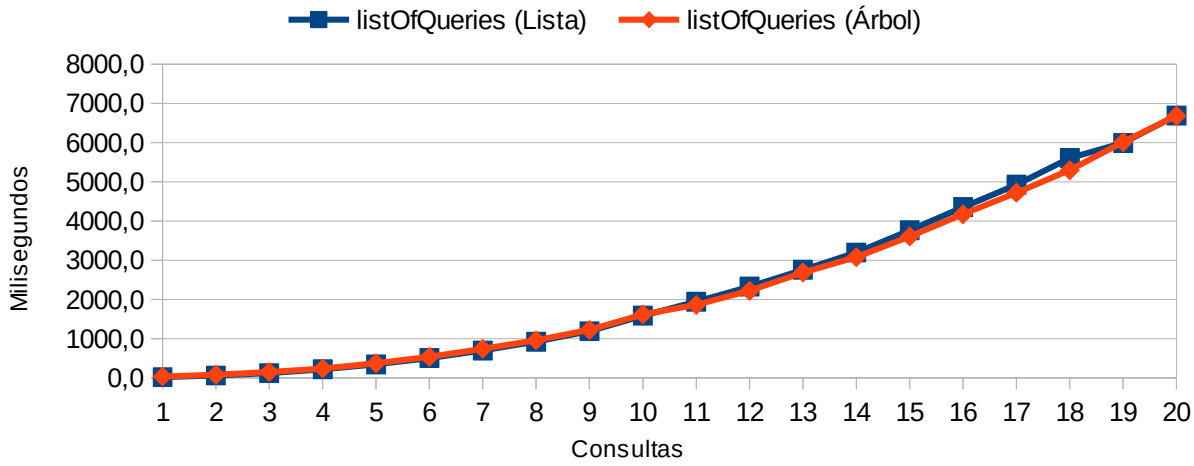


Figura 9: Gráfico de tiempos de ejecución del algoritmo listOfQueries para el caso peor y rango 1.000 a 20.000 consultas.

Índice de tablas

Índice de gráficos