INFORME DE PRÁCTICA 1

1.1. ¿Qué problemas introduce en una ordenación comparar distancias a un punto determinado en vez de directamente valores enteros?

Se ha tenido que crear una clase punto para guardar las coordenadas x e y de un punto y que calcule la distancia a otro punto que se introduzca. Para comparar la distancia a un punto hay que calcular en cada iteración la distancia al punto respecto al cual se quiere ordenar las distancias, el cual se ha tenido que introducir como entrada extra a las funciones. Esto hace que se necesite más memoria que la que utilizarías solo para ordenar enteros.

1.2. ¿Qué ventajas e inconvenientes tienen las alternativas que usan menos o más memoria y por qué?

Las alternativas que utilizan más memoria son alternativas más rápidas ya que al utilizar más memoria no son necesarios tantos cálculos y esto hace que el tiempo de ejecución sea menor.

Las alternativas que utilizan más memoria son alternativas que consumen menos recursos.

1.3. ¿Qué coste/complejidad tiene cada alternativa en tiempo y memoria?

El algoritmo de inserción directa tiene un coste de tiempo de O(n^2) y un coste de memoria de O(1).

El algoritmo de Quicksort tiene un coste de tiempo de O(n log n) y un coste de memoria de O(n).

2.1. ¿Cómo has eliminado la recursión final en el algoritmo de quicksort?

He utilizado una pila para guardar los partes del array que queremos ordenar. Cuando lo que queremos ordenar con la recursión final es más de un elemento guardamos el inicio y el fin en la pila creada. Tras hacer esto entramos en un bucle del que se sale cuando la pila esté vacía, es decir, cuando el array esté completamente ordenado. En este bucle utilizamos la pila para hacer el algoritmo Quicksort pero de manera iterativa en vez de recursivamente. Para hacer esto, como se hace de la manera recursiva, llamamos a la función partición, quitando el inicio y final del array de la pila, que lo que hace es que a través del último elemento, que es el elemento que utiliza como pívot, del array deja al inicio del array los elementos menores que el pívot y al final los mayores que el pívot y coloca el pívot en la posición donde se juntan. Tras hacer esto, si los elementos menores del pívot son mínimo dos guarda el inicio y fin de este nuevo array en la pila. Hace lo mismo con los elementos mayores del pívot.

2.2. ¿Cómo has introducido la inserción directa en quicksort?

He añadido al Quicksort un if que, en el caso de que haya elementos del array que ordenar, mire si el número de elementos que quedan por ordenar del array, que corresponde con el resultado de fin-inicio, es menor o igual que el valor de la entrada c. Si esto es así, lo ordena mediante inserción directa. Si no es así lo ordena por quicksort.

3.1. ¿Cómo has diseñado los experimentos para comprobar la corrección de los programas y para medir qué versión es mejor o peor?

Para medir qué versión es mejor o peor he cogido varios tamaños aleatorios del vector de puntos y he generado aleatoriamente puntos aleatorios para este tamaño de vector. He calculado el tiempo que tarda para estos casos y con estos datos he generado un gráfico para ver de manera más visual los datos.

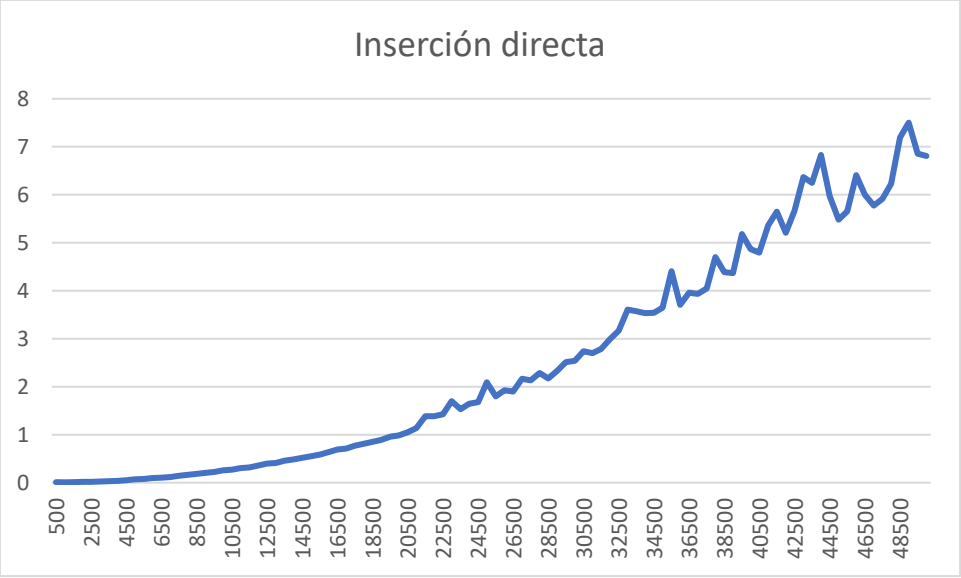
Para comparar Quicksort con inserción directa he utilizado arrays de un tamaño cuya ordenación sea prácticamente 0 a arrays que tarden varios segundos en los dos casos. Tras esto cojo tamaños adecuados para comparar ambos algoritmos.

Para ver cual es el valor óptimo de c hemos generado muestras aleatorias de un vector de un determinado tamaño y hemos visto para qué valor de c el tiempo de ejecución es menor.

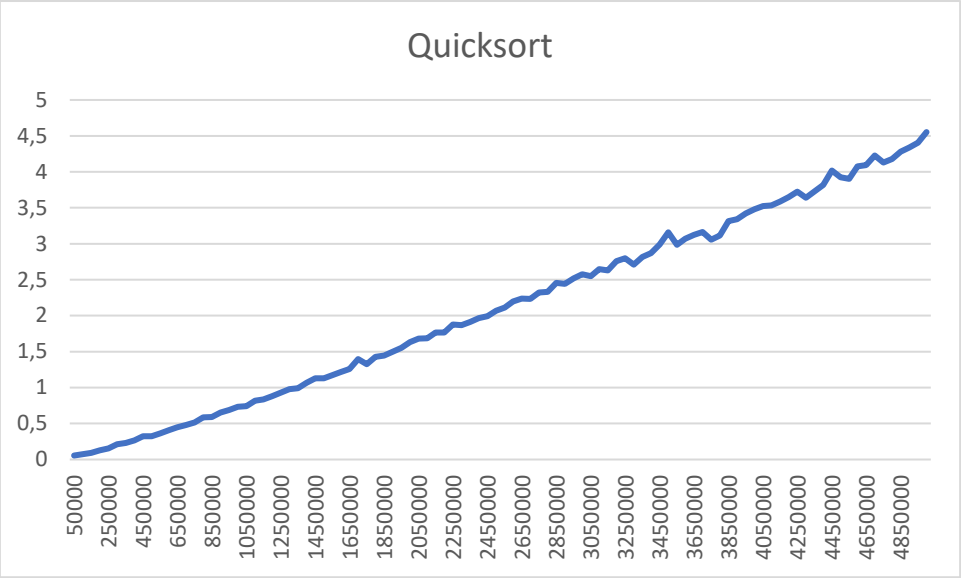
Para ver si hay mejoras de rendimiento de la versión en la que se ha cambiado la segunda llamada recursiva del algoritmo de Quicksort por un método iterativo con el Quicksort original calculamos el tiempo de ejecución con esta modificación con los tamaños de arrays que habíamos utilizado con el Quicksort normal y he comparado sus tiempos de ejecución.

3.2. ¿Qué tamaños de problema has usado y que tiempos aproximados has obtenido?

Para la inserción directa he tomado tamaños empezando por 500 hasta 50000. Para 500 tarda 0.01 segundos y para 50000 tarda 6.812 segundos. Creo una gráfica para el tiempo obtenido para los diferentes tamaños del vector de puntos:

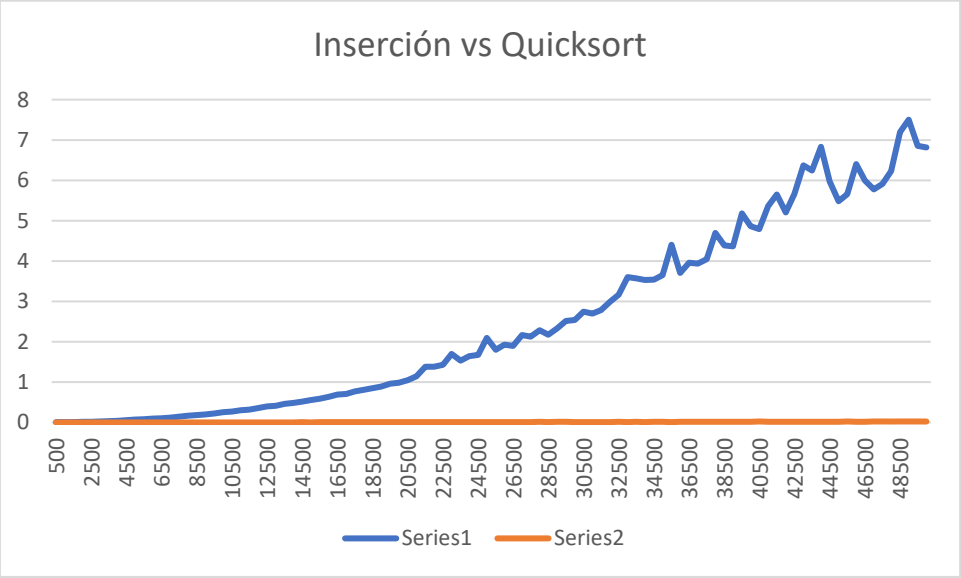


Para Quicksort hemos utilizado datos de 50000 a 5000000. Con tamaño de 50000 tarda 0.055 segundos y con tamaño 5000000 tarda 4.553 segundos. Creo una gráfica para los tiempos de Quicksort:



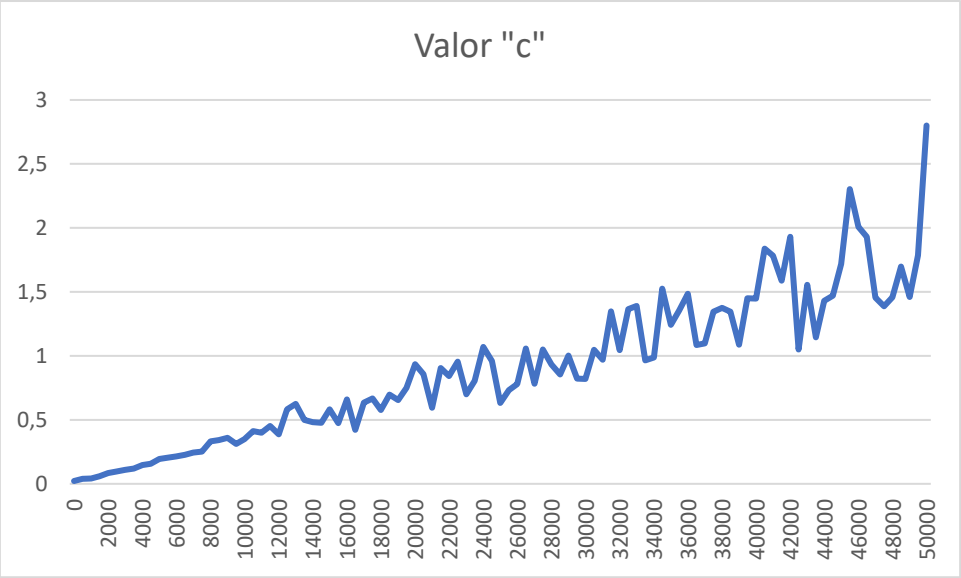
Por lo que vemos con los datos obtenidos parece que el algoritmo de Quicksort parece que el algoritmo Quicksort es más rápido que el algoritmo de inserción directa.

Para comparar ambos gráficos necesitamos compararlos para los mismo tamaños de array, por lo que hacemos Quicksort para los tamaños de arrays que hemos utilizado para inserción directa. Para tamaño 500 tarda 0.002 segundos y para tamaño 50000 tarda 0.024 segundos, en ambos casos menos que con la inserción directa. Hacemos un gráfico que compare los tiempos de la inserción directa y los tiempos de Quicksort.

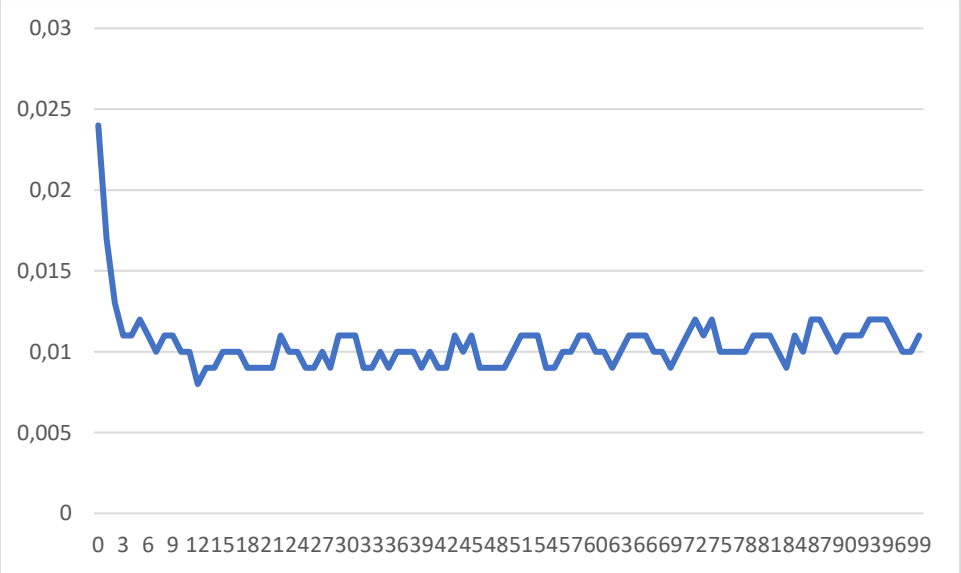


Como se puede ver, en este caso, los valores del tiempo en Quicksort el tiempo nos da prácticamente 0.

Para ver cual es el menor valor de c he creado un bucle que vaya desde c=0 a c=50000 y en cada iteración genere un vector de puntos aleatorios de tamaño 50000 y ordene el vector Quicksort con el valor que corresponda de c y medimos el tiempo que tarda. Usamos los datos que nos de para hacer un gráfico con los puntos y ver cual es menor. La gráfica resultante es la siguiente:

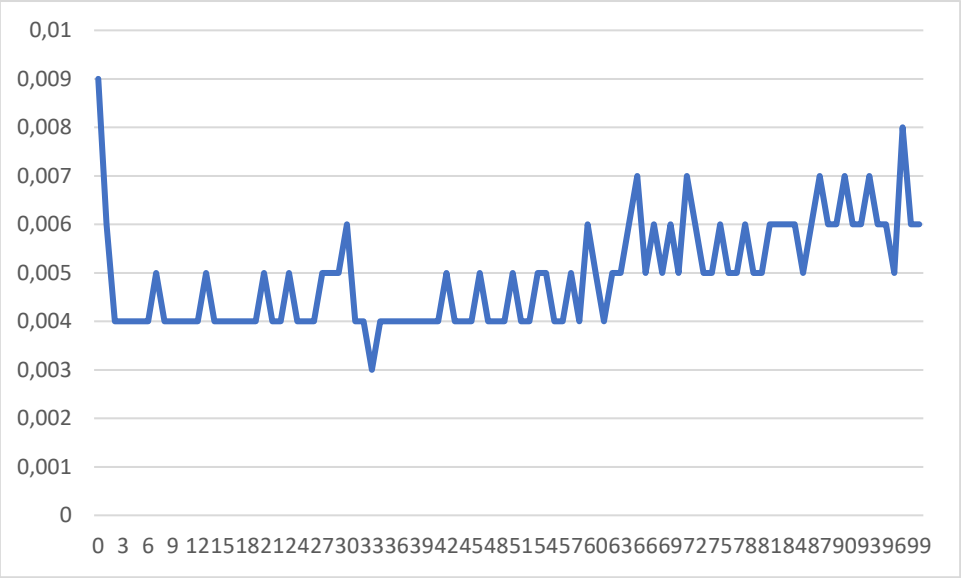


Por lo que parece el tiempo de ejecución es menor cuanto menor sea el valor de c. Vemos como evoluciona el tiempo para valores bajos de “c”:



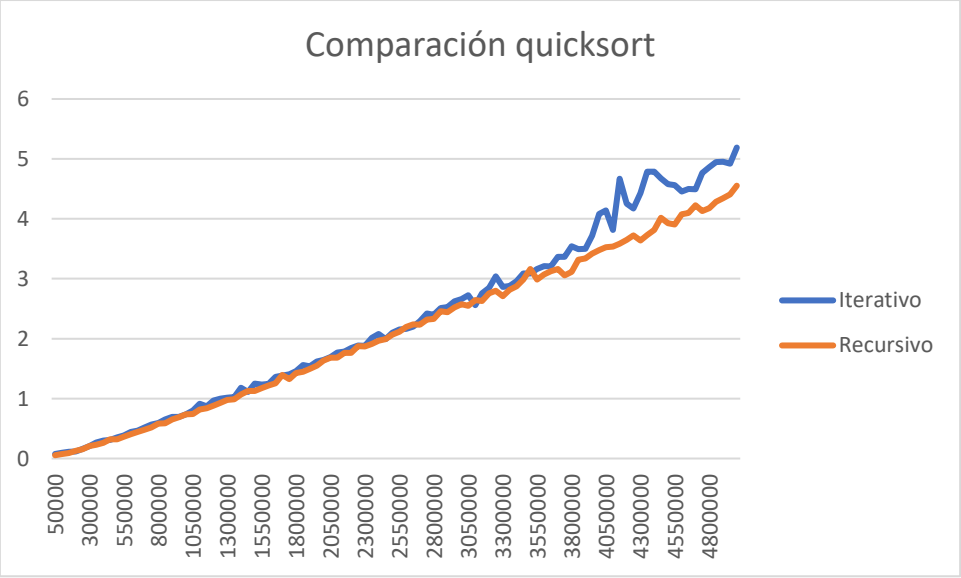
Como vemos en el gráfico no parece haber muchos cambios para valores bajos de c. Aunque con los valores cercanos a 100 vemos que ya va aumentado el tiempo de ejecución. También vemos que alcanza su valor mínimo cuando c=12, por lo que podríamos pensar que el valor óptimo de c será o este valor o cercano a él. Y el valor del tiempo cuando c=0 es bastante alto, lo que nos lleva a pensar que el algoritmo de inserción para arrays de poco tamaño funciona mejor que el Quicksort.

Hacemos lo mismo para tamaño 10000 y vemos para tamaños pequeños de c cual es el tiempo de ejecución:



En este caso el mínimo tiempo de ejecución es cuando c=33.

Para la segunda modificación ejecutamos el algoritmo en el que se ha cambiado la segunda llamada recursiva por un proceso iterativo para n desde 50000 a 5000000. Para un tamaño del array de 50000 me da un tiempo de ejecución de 0.076 segundos y para un tamaño del array de 5000000 me da un tiempo de ejecución de 5.189 segundos. Representamos estos datos comparándolos con los originales:



Como vemos en el gráfico no hemos vista que haya cambios significativos entre ambas maneras. Aunque para valores altos de n parece que funciona mejor el método con dos llamadas recursivas.

4.1. ¿Qué conclusiones has obtenido sobre los cambios en el coste al eliminar la recursión final o introducir el algoritmo de inserción directa en el quicksort?

Como hemos visto anteriormente, al eliminar la recursión final del algoritmo de Quicksort obtenemos resultados similares que al no hacerlo aunque para valores grandes de n funciona mejor el Quicksort con dos llamadas recursivos.

Al introducir el algoritmo de inserción directa en el Quicksort hemos observado que el valor c con menor tiempo de ejecución era c=12 para arrays de tamaño 50000. Para arrays de tamaño 10000 el menor tiempo de ejecución ha sido cuando c=33. Viendo estos resultados vemos que inserción directa solo funciona mejor que Quicksort para arrays de poco tamaño.

4.2. ¿Qué versiones son mejores o peores y por qué?

Las versiones de Quicksort inicial y la que cambias la segunda llamada recursiva por un método iterativa funcionan similar, aunque para arrays de gran tamaño parece que funciona mejor el Quicksort inicial.

La versión en la que introducimos el algoritmo de inserción directa para valores pequeños de c funciona mejor que el Quicksort normal, ya que si este fuera el caso el valor óptimo de c sería de 0, y no es así.

Como ya hemos visto el algoritmo de Quicksort funciona mejor que el algoritmo de inserción directa.

Viendo esto, llegamos a la conclusión de que la peor versión es la de inserción directa y la mejor es la de Quicksort que usa inserción directa con arrays de poco tamaño.

4.3. ¿Qué lenguaje has usado, qué problemas has tenido?

Para hacerlo he utilizado java. He tenido problemas para hacer los algoritmos en vez de con números enteros con distancias ya que al principio no tenía claro si usar los valores de las distancias o directamente de los puntos. Finalmente, he decidido hacerlo de la última forma. Con el Quicksort me ha costado entender para que se utilizaban las variables hi y lo que aparecían el pseudocódigo.