INFORME DE PRÁCTICA 1

Víctor Mulero Merino

1. ¿Qué problemas introduce en una ordenación comparar distancias a un punto determinado en vez de directamente valores enteros? ¿Qué ventajas e inconvenientes tienen las alternativas que usan menos o más memoria y por qué? ¿Qué coste/complejidad tiene cada alternativa en tiempo y memoria?

   En primer lugar, tenemos varias alternativas para hacer el algoritmo de ordenación de puntos. La primera alternativa sería que al comparar un punto con el punto pivote de Quicksort, se calculan en ese momento las distancias al punto de referencia. Esta solución apenas tiene coste en memoria, ya que las distancias que se calculan no se guardan, únicamente se usan para comparar. Solo se requiere el espacio del propio array de n elementos que hay que ordenar. Sin embargo, se ve perjudicado el coste de tiempo, ya que puede que se calcule la misma distancia varias veces a lo largo de la ejecución del algoritmo.

   Tiempo: O(n) | Espacio: O(n)

   Otra solución sería crear al inicio del algoritmo un vector de distancias y realizar las comparaciones según ese vector de distancias. En este caso, al intercambiar dos elementos (A[i] <-> A[j]) necesitaríamos también intercambiar las distancias correspondientes (distancias[i] <-> distancias[j]). Esta solución es más eficiente en tiempo que la anterior, ya que las distancias solo se calculan una vez. En cuanto al espacio, se requiere más memoria para guardar las distancias.

   Tiempo: O(n) | Espacio: O(2n) = O(n)

   Por último, se puede crear una clase Punto con 3 atributos: coordenada *x*, coordenada *y*, *distancia*. Cada objeto de tipo Punto se inicializa con las coordenadas *x* e *y*, y al inicio del algoritmo se calcula la distancia de cada Punto al punto de referencia. Esta opción es similar a la anterior, solo que al hacer un intercambio A[i] <-> A[j] no hay que intercambiar las distancias, ya que son atributos de cada elemento del array. Esta opción es la que se ha implementado en la solución.

   Tiempo: O(n) | Espacio: O(n)

2. ¿Cómo has eliminado la recursión final en el algoritmo de quicksort? ¿Cómo has introducido la inserción directa en quicksort?

Para eliminar la recursión final se ha usado una pila para transformar el proceso recursivo en un proceso iterativo. En la pila se incluyen dos límites, que hacen referencia a los índices que marcan los límites del array que hay que ordenar (elementos *low* y *high* de Quicksort). Se realiza la partición igual que en el Quicksort recursivo, y se incluyen en la pila los dos nuevos índices para seguir ordenando. Esto se realiza con un bucle *while*, y dura hasta que la pila no tenga elementos.

Para implementar la inserción directa en Quicksort he realizado dos comparaciones en la función recursiva. En cada llamada a la función se divide la lista que hay que ordenar en dos partes que hay que ordenar por separado, según un índice *p*. Se comprueba si la longitud de cada una de esas listas es menor o igual que el umbral *c* que se introduce como argumento, y en el caso de que se cumpla la condición, se usa *insertion sort*. En caso contrario, se continúa con las llamadas recursivas.

3. ¿Cómo has diseñado los experimentos para comprobar la corrección de los programas y para medir qué versión es mejor o peor? ¿Qué tamaños de problema has usado y que tiempos aproximados has obtenido?

Para comprobar que las funciones que he creado funcionan, he generado un vector de 100 puntos y lo he ordenado usando esas funciones. He comprobado que las distancias de los puntos quedan ordenadas en orden creciente.

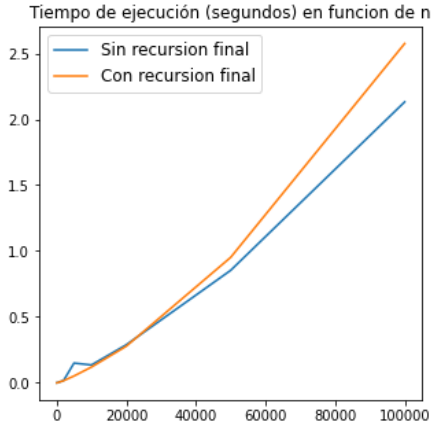
En el método 1, para saber que umbral *c* es el mejor, he probado el algoritmo con diferentes valores de *c*, desde *c = 0* hasta *c = 200* de 10 en 10. Para todos los valores de *c* se ha usado el mismo conjunto de 1000 puntos generado aleatoriamente. He medido el tiempo que tarda el algoritmo en ordenar el vector de puntos con cada umbral, y he realizado una gráfica:

Se observa que el valor de *c* para el cual se obtiene un menor tiempo de ejecución esta por debajo de 25.

He repetido lo anterior 10 veces, obteniendo cada vez el valor de *c* para el cual se tiene un tiempo menor. El valor de *c* que más veces aparece como valor óptimo es *c = 10.* Por tanto, el mejor valor para *c* estará en torno a ese valor.

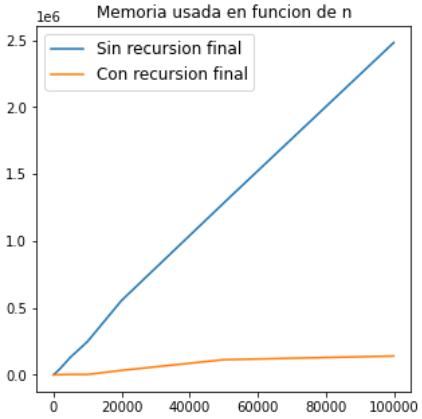
Para este método 1 se ha usado n = 1000, un número bastante bajo de puntos. Por tanto, los tiempos de ejecución son muy bajos. Se puede observar en el gráfico que al aumentar el valor de *c* aumenta el tiempo de ejecución, ya que se usa *insertion sort* en lugar de *quicksort*, que es bastante más rápido para n alto. Por tanto, no ha sido necesario probar para valores de n más altos, porque el valor del umbral *c* se encuentra mucho antes.

Para el método 2 he probado el rendimiento del *quicksort* con el método 2 con respecto al *quicksort* normal, que es puramente recursivo. He realizado experimentos con diferentes valores de n, generando n puntos aleatorios y realizando una copia para usar los mismos puntos en ambos algoritmos. He medido los tiempos de ejecución para cada algoritmo y he obtenido la siguiente gráfica:



Para valores pequeños de n el comportamiento de ambos algoritmos es similar. Sin embargo, para n altos se observa una ligera mejoría en el algoritmo que no usa recursión final (el del método 2).

También se ha medido el espacio de memoria ocupado, y como resultado tenemos la siguiente gráfica:



Como era de esperar, la implementación que utiliza un proceso iterativo con una pila consume mucha más memoria que el *quicksort* recursivo. Se debe a que se van guardando en la pila los valores límite para saber la zona del array que hay que ordenar.

Por tanto, la eliminación de la recursión final presenta una mejora en el tiempo, pero el espacio se ve muy perjudicado.

4. ¿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? ¿Qué versiones son mejores o peores y por qué? ¿Qué lenguaje has usado, qué problemas has tenido?

   En cuanto a la recursión final, la conclusión obtenida es que al eliminarla se produce una ligera mejoría en el tiempo. Para valores de n pequeños el comportamiento es similar, pero a la larga es más eficiente la solución que incluye un proceso iterativo en lugar de recursivo. Sin embargo, el coste en memoria de esta implementación es mucho peor que el algoritmo recursivo.

   La implementación del algoritmo de inserción directa en *quicksort* no es de utilidad, ya que no aporta ninguna mejora. Se ha concluido que los mejores valores de *c* están alrededor del 10, y se tarda prácticamente lo mismo en que ese array de tamaño 10 lo ordene *quicksort* que *insertion sort*.

   Por tanto, en el caso de tener que incluir alguna implementación, sería mejor eliminar la recursión final del *quicksort*, ya que si que es ligeramente mejor que el *quicksort* recursivo en cuanto a tiempo. Pero si se tiene en cuenta el espacio, no conviene usar el algoritmo sin recursión.

   Para resolver esta práctica he usado Python. No ha habido ningún problema. Lo único que no tenía claro era si necesitaba crear una clase Punto, o simplemente se podía crear una lista de tuplas, cada tupla simulando un punto con coordenadas *x, y*.