Informe Práctica 1

1.

El problema de ordenar las distancias a un punto en vez de enteros es que las distancias normalmente serán números decimales. En coma flotante decimal, 0.1+0.1+0.1-0.3 es exactamente igual a cero. En coma flotante binaria, el resultado es 5.5511151231257827e-017. Aunque cercanas a cero, las diferencias impiden pruebas de igualdad confiables y las diferencias pueden acumularse y en puntos muy cercanos al punto de referencia, podrían no ordenarse correctamente.

Quicksort tiene la importante ventaja en términos de eficiencia, debido a que es capaz de poder tratar con una lista grande de elementos de forma muy rápida O(nlogn). Además, dado a que ordena en el lugar, su principal ventaja es que no requiere de almacenamiento adicional, aunque en las llamadas recursivas utiliza muchos recursos. El inconveniente es que, en el peor de sus casos, su rendimiento es igual al algoritmo de ordenación por inserción, O(n^2), que se da cuando el array ya estuviera ordenado o si todas las distancias fueran las mismas.

El algoritmo de ordenación por inserción tampoco necesita un array auxiliar para realizar la ordenación por tanto no requiere almacenamiento adicional y el requerimiento de memoria es mínimo. Es muy rápido para listas pequeñas (n<20), pero no para listas grandes, se realizan numerosas comparaciones y el orden es de O(n^2).

Algunos algoritmos como MergeSort utiliza un array de al menos n/2, por lo que necesita almacenamiento adicional para el array auxiliar, o el algoritmo de ordenación burbuja que requiere muchas lecturas y escrituras

en memoria. Estos algoritmos se podrían modificar para usar menos espacio en memoria, pero a cambio de hacerlos más lentos.

2.

Método 1:

He pasado como argumento en Quicksort el número límite y al hacer la comprobación de que el pivote de la izquierda sea menor que el de la derecha, también deberá comprobar que el límite establecido sea mayor o igual que 0, ya que en cada llamada iterativa se disminuye en 1 el límite hasta que llegue a 0. En ese caso, se utilizaría el ordenamiento por inserción para terminar de ordenar el vector.

Método 2:

Para evitar el peor caso en el que una parte del array tenga 0 elementos y la otra n-1, se puede limitar el espacio a O(logn). Se basa en "tail call elimination", esto es si la llamada recursiva es lo último ejecutado por la función. Se debe añadir un bucle while y el índice de partición quedaría igual. Después hay que ordenar los elementos separadamente antes y después de la partición.

3.

Primero he ejecutado los programas de inserción, Quicksort y la modificación del programa del Método 1 con límite establecido a 1 y la modificación del Método2 con el siguiente vector de tamaño n = 34 y el punto inicial de referencia (1,2)

(4,6), (5,3), (3,3), (7,9), (1,4), (2,2), (7,6), (8,2), (9,3), (-3,7), (2,2), (0,0), (5,10), (-15,1), (-3,0), (6,1), (7,7), (9,9), (0,2), (12,9), (5,9), (32,7), (-22,4), (-10,7),

(30,22), (19,12), (-16,8), (-5,-2), (-7,17), (80,3), (-41,30), (-25,26), (-1,-18), (30,20)

Método 1:

Se ha obtenido que el mejor tiempo para el ordenamiento es el algoritmo de Quicksort con un tiempo de ejecución de 5e-05 segundos. Después del de ordenación por inserción con 0.00017 segundos y el último la modificación realizada con 0.00067 segundos de ejecución. Como el límite establecido es muy bajo, prácticamente todo el vector ha sido ordenado con el algoritmo de inserción que no resulta efectivo para listas grandes. Si modificamos el límite a 20, se puede reducir a la mitad el tiempo de ejecución.

Método 2:

Para el mismo vector utilizado en el método anterior, la mejora en el rendimiento de esta modificación es significativa ya que el tiempo de ejecución ha sido de 3e-05 segundos, el mejor obtenido de todos los algoritmos probados hasta ahora.

Después he ejecutado los mismos programas para la lista de puntos (4,6), (5,3), (3,3), (7,9), (1,4), (2,2), (7,6), (8,2), (9,3).

Se han obtenido los siguientes tiempos:

Inserción: 1e-05 segundos

Quicksort: 2e-05 segundos

Método 1: 0.00047 segundos (límite 20) 0.00024 segundos (límite 1)

Método 2: 1e-05 segundos

Como se puede observar, para arrays de tamaño pequeño, el algoritmo de inserción funciona mejor que el algoritmo de Quicksort, aunque la mejora no es muy evidente, al igual que la mejora del método 2. El método 1 es el que peor funciona, sin embargo, si modificamos el límite, puede mejorar sustancialmente ya que, para un límite bajo, la mayor parte de la ordenación la realizará con inserción en vez de con Quicksort y eso da mejores resultados.

4.

Eliminando una llamada recursiva se ha mejorado significativamente el rendimiento del algoritmo ya que se ha pasado de un orden de complejidad de O(nlogn) a un orden de O(logn).

Añadiendo la llamada a la función de ordenación por inserción se han empeorado los tiempos con respecto al algoritmo de inserción o al algoritmo Quicksort por separado. Dependiendo para qué tamaño del array se esté utilizando, la variación del límite puede conllevar una mejora muy significativa en el rendimiento de esta modificación. Esta modificación es peor ya que además de todas las llamadas recursivas que se estaban haciendo en Quicksort, se añade una llamada a otra función que tiene dos bucles for anidados. En el peor de los casos, dependiendo del límite, podría obtenerse el caso en el que Quicksort tenga que ordenar una lista pequeña, lo cual es aproximadamente de O(n^2) y que la parte de inserción tenga que ser una lista larga, lo cual también es O(n^2).

He usado el lenguaje de Python. No ha habido problemas.