Índice

1. Pregunta 1

1.1. Calcular las distancias

Si tuvieras que calcular las distancias cada vez que comparas dos puntos, el coste temporal se dispararía. Por lo tanto, he computado la distancia al principio para cada punto y las he introducido en el *struct punto* como una variable (es decir, como parte del estado del *"objeto"* punto).

De esta forma simplemente tenemos que comparar *doubles* y se reduce a realizar *quicksort* como ya sabemos.

1.2. Ventajas e inconvenientes

Según las medidos que he tomado los distintos métodos son muy similares en cuanto a coste temporal, sin embargo el método 2 es tremendamente costoso espacialmente.

Es posible que el que sean tan similares sea causado porque el método que he usado para medir tiempos/memorias no sea lo suficiente preciso.

1.3. Costes temporales/espaciales

El coste temporal de todos los métodos es el mismo, $n \log n$. Esto se puede calcular por métodos recurrentes.

El coste espacial de quicksort sería n aproximadamente, coincidiendo con el del método 2.

2. Pregunta 2

2.1. Recursión final

Para eliminar la segunda recursión he tenido que sustituirla por otro método, en este caso uno iterativo. He necesitado guardar los parámetros lo, hi del subarray correspondiente a segunda recursión para poder aplicar *particion* sobre el subarray posteriormente de forma iterativa.

Esto lo he hecho añadiéndolo a una estructura de datos auxiliar (una lista de subarrays con parámetros $lo, hi, next$ de la que guardaba un puntero tanto a la cabeza como a la cola) antes de llamar a la primer recursión. Tras esta, en lugar de la segunda he puesto un bucle que recorra la lista de la cabeza a la cola aplicando *particion* a los subarrays de la lista (solo si se da que $hi > lo \leq 0$) y, añadiendo los dos subarrays resultantes al final de la lista cada vez que aplicamos *particion*. El bucle termina cuando no hay subarrays válidos ($hi > lo \leq 0$) restantes.

2.2. Inserción directa

Programar la inserción directa ha sido más fácil. Simplemente la función *quicksort*1 antes de realizar el algoritmo *quicksort* comprueba si la longitud del subarray ($hi - lo$) es menor que la constante c (que le viene como parámetro). De así serlo aplica *insertionsort*, en caso contrario continua con *quicksort* llamándose a sí misma recursivamente y aplicando *particion*.

3. Pregunta 3

3.1. Correción

Para comprobar la corrección de un programa he introducido un input simple a cada función de forma sistemática y comprobado que el output era el correcto.

Para las únicas funciones para las que lo he hecho diferentes ha sido para los algoritmos de ordenamiento, para los cuales he introducido un array de puntos desordenado manualmente y, tras aplicar el algoritmo, he escrito un bucle for que imprima los tamaños de cada uno de los elementos. De esta forma es más fácil ver que están correctamente ordenados y no hay que hacer cuantas manualmente (porque ya había comprobado previamente que la función que calcula el tamaño, $tamf$, funcionaba correctamente).

3.2. Medidas

Para realizar las medidas he usado la librería $sys/resource.h$ y $sys/time.h$ con las función $getrusage()$. He sacado los parámetros de $usage.ru_maxrss$ (memoria física máxima usada en kylobytes), $usage.ru_stime.tv_sec$(segundos del CPU consumidos) y $usage.ru_stime.tv_usec$ (microsegundos del CPU consumidos).

Esta no ha sido una buena idea, pues las medidas de tiempo son bastante imprecisas (en más de una ocasión no distinguían entre el input de 10000 y el de 100000) y las de espacio parecen ser afectadas por el sistema de paginación de la máquina. El coste espacial cambia de medida a medida (con un mismo input) porque la máquina tiene huecos para la memoria dinámica en diferentes sitios y puede abrir más o menos bloques para almacenar la misma cantidad de información.

La input lo he generado de forma aleatoria con un programa $random.c$ que he escrito. Su funcionamiento es el típico, genera enteros pseudoaleatorios tomando el tiempo como semilla y los transforma a flotantes.

Todos los algoritmos han tenido el mismo input para que no haya diferencias entre ellos $1000.dat, 10000.dat, 100000.dat$.

4. Resultados

En la primera linea tenemos el tamaño del input, en las demás el coste temporal/computacional de cada uno de los algoritmos.

TIEMPO	10^3	10^4	10^5
quicksort	0.015	0.031	0.015
método 1	0	0.015	0.031
método 2	0.031	0.015	0.015

MEMORIA	10^3	10^4	10^5
quicksort	724	936	3048
método 1	724	936	3048
método 2	30564	30564	30564

5. Pregunta 4

5.1. Conclusiones

Con la calidad de los datos tomados es difícil sacar conclusiones. La única conclusión alcanzable es que el segundo método tiene un gigantesco coste espacial comparado con los otros. Aunque esto es posible que sea por como se mide (la memoria dinámica donde está la lista de puntos está más dispersa que el stack donde se guarda los datos de llamada en la recursión).

El método 1 teóricamente debería mejorar siempre la eficiencia espacial del algoritmo, empeorando su eficiencia temporal para c grandes y mejorándolas para c pequeñas. El segundo método no debería afectar al coste temporal sino al espacial.

5.2. Lenguaje

He escrito el programa en lenguaje C, lo que obviamente ha dado problemas a la hora de medir la eficiencia temporal/espacial del programa. También me ha dado problemas a la hora de trabajar con la estructura de datos auxiliar debida al uso de punteros y a la forma de C de informar de errores.