Belén Carolina Saldías Fuente

Ejercicio 1:

Complejidad void mergeSort()

En primer lugar hay que darse cuenta que el mergeSort hace un llamado a otra función que es: merge.

Sean e: índice inicial de la posición del arreglo al que se le realizará merge

w: índice de la posición final del arreglo al que se le realizará merge

m: (w+e)/2 = índice de al medio acotado por abajo.

Con un contadores inicializados p = e, y q = m, se recorre el arreglo desde p a m y desde q a w. Es decir, desde el inicio a la mitad y desde la mitad al final. Como este recorrido se realiza solo una vez, tenemos que la primera parte del merge como pasa solo una vez por cada elemento es O(n), ya que cada operación que se realiza en una iteración es O(1), reasignar valores y aumentar contadores.

Luego de esta parte, el merge copia el arreglo temporal - en el que fueron copiados los elementos en la posición correcta respectiva - en el arreglo original A, es decir pasa una vez por cada dato correspondiente del arreglo original entre los índices e y w, esto es O(N).

Como mergeSort es una función recursiva, si complejidad puede calcularse así:

T(n) = cte + T(n/2) + T(n/2) + k*n; k es otra cte.

Ya que en el mergeSort de un arreglo de largo N hay asignaciones de variables que son O(1), luego 2 llamados a mergeSort con arreglos de tamaño N/2 y posteriormente un llamado a merge que es el algoritmo cuya complejidad es la anteriormente calculada.

$$T(n) = cte + 2 * T\left(\frac{n}{2}\right) + k * n$$

$$T(n) = cte + 2 * [\ \ cte + 2 * T\left(\frac{n}{4}\right) + k * \frac{n}{2}\ \ ] + k * n$$

$$T(n) = cte + 2 * cte + 4 * T\left(\frac{n}{4}\right) + k * n\ \ + k * n$$

$$T(n) = 3 * cte + 4 * T\left(\frac{n}{4}\right) + 2 * k * n$$

...

Para poder obtener el resultado de esta recursividad tenemos que llegar al caso base, considerando así T(1) = 1. Ya que en ese caso solo hay una comparación y ninguna llamada recursiva.

De esta manera T(1) = T(n/n); si consideramos $n = 2^i$ entonces i = log(n). Resolviendo lo anterior entonces:

$$T(n) = \left(2^i - 1\right) * cte + 2^i * T\left(\frac{n}{2^i}\right) + i * k * n$$

Reemplazando $n = 2^i$

$$T(n) = (\mathrm{n} - 1) * cte + \mathrm{n} * T(1) + \log(n) * k * n$$

$$T(n) = (\mathrm{n} - 1) * cte + \mathrm{n} + k * n * \log(n)$$

$$\boldsymbol{T(n) = O(n + n * \log(n)) = O(n * \log(n))}$$

Así, en el pre-procesamiento de los datos se lleva $\boldsymbol{O(N) + O(N * \log(N))}$

Complejidad int solution()

Es en este método donde se hacen 2 llamados a mergeSort, por lo tanto partimos de la base O(N*log(N)).

Luego de eso hay un **while** con otro **while** dentro, esto podría orientar a que es N*2, sin embargo hay que poner atención en que solo entra al segundo **while** N veces, ya que posiciones[contador][1] es igual 0 solo en el inicio de cada disco (y hay N discos). Además, el **while** interior no siempre se ejecuta N veces, ya que como los datos están ordenados cada vez que ya encontró cuántos discos comienzan en el rango de otro (entre el inicio y fin de otro), para el siguiente disco no comparará con el disco anterior.

Así se tiene (para el peor caso):

Para el primer inicio compara para 2*N – 2 posiciones si: posiciones[i][1] == 0.

Para el segundo inicio compara para 2*N – 1 – 2 si: posiciones[i][1] == 0.

Para el tercer inicio compara para 2*N – 2 – 2 si: posiciones[i][1] == 0.

Resolviendo esto se llega a una sumatoria en que da proporcional a logN, ya que a lo más una vez se recorre el arreglo completo, y en cada iteración se recorren menos elementos del arreglo. Así, se

hacen N de estas iteraciones cada vez más pequeñas, por lo tanto se tiene N*log(N), ya que los datos ordenados causan que todo sea más rápido, así usamos con el ordenamiento y se queda finalmente **en O(N*log(N))**

Memoria:

Se utilizan 2 listas bidimensionales, una con los datos originales y otra temporal en el **merge**(), ambas de dimensión cte*N*3, así el espacio usado es O(N), lo demás son variables independientes de N.

Ejercicio 2:

Complejidad void buscar():

Este algoritmo es básicamente BFS, por cada vértice se revisa a todas las aristas que llega, sin repetir vértices, ya que al buscar el camino más corto, la primera vez que llega ya es por el camino más corto, ya que en primera instancia, dado un vértice, descubre todos los que están a distancia 1 de él, luego los que están a distancia 1 de los que están a distancia 1 de él y por tanto están a distancia 2 de él, y así sucesivamente.

Cada **queue()** y **enqueue()** es O(1) ya que es solo un cambio de punteros en cada operación.

Por lo tanto es de complejidad O(V + E), ya que pasa a lo más una vez por cada vértice y arista, donde V es el número de vértices y E es el número de aristas.

Memoria:

Se crea una lista con los vértices referenciándolos para poder identificarlos por su id, esto es O(V), luego se crea un grafo con tantos vértices y aristas como existen, por lo tanto es O(cte1*V + cte2*V + cte*E) = O(V + E), donde V es el número de vértices y E es el número de aristas.