Act 1.3 - Actividad Integral de Conceptos Básicos y Algoritmos Fundamentales

Programación de estructuras de datos y algoritmos fundamentales (Gpo 4)

Thomas Freund Paternostro

A00831997

Alumnos Grupo:

José Ángel Rentería Campos //A00832436

Santiago Andrés Serrano Vacca //A01734988

Thomas Freund Paternostro //A00831997

Fecha de entrega:

10/09/2021

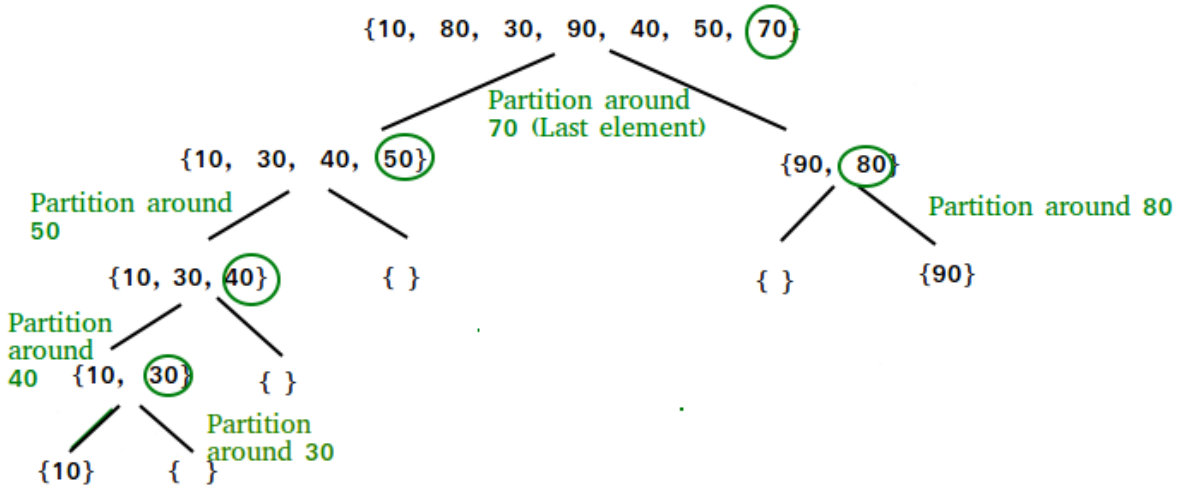
```
172   void quickSort(vector<Bitacora> &v, int inicio, int fin)
173   {
174     if (inicio < fin)
175     {
176       int p = Particion(v, inicio, fin);
177       quickSort(v, inicio, p - 1);
178       quickSort(v, p + 1, fin);
179     }
180   }
```

Al ser el quicksort un método de ordenamiento recursivo, tenemos que:

$$T(n) = n + T(n-1)$$
$$T(n-1) = (n-1) + T(n-2)$$
$$T(n-2) = (n-2) + T(n-3)$$
$$T(n-3) = (n-3) + T(n-4)$$
$$\dots$$
$$T(3) = 3 + T(2)$$
$$T(2) = 2 + T(1)$$
$$T(1) = 0$$

Entonces…

$$T(n) = n + (n-1) + (n-2) + (n-3) + (n-4)\dots + 3 + 2$$
$$= 1/2(n(n+1)) - 1 = O(n^2)$$



Explicación: Este es un algoritmo que por medio de recursividad hace un llamado a otra función que hace particiones donde el objetivo es dividir y vencer donde se puede resolver un problema en nlogn hasta n^2 como fue demostrado en la parte superior. En la imagen se puede evidenciar cómo dividir las particiones y por medio de esto identifica el mayor y los agrupa de nuevo en la función de particiones. (GeeksforGeeks)

```
150   //Fun Particion que es llamada
151   int Particion(vector<Bitacora> &v, int inicio, int fin)
152   {
153     int mitad = fin;
154     int j = inicio;
155     for (int i = inicio; i < fin; ++i)
156     {
157       //si el key en el inidice actual es menor al del
        medio
158       if (v[i] < v[mitad])
159         {
160           //swap  con el incial
161           swap(v[i], v[j]);
162           ++j;
163         }
164     }
165     //swap con el de la mitad
166     swap(v[j], v[mitad]);
167     return j;
168   }
```

Línea	Costo	Repeticiones (peor caso)
153	C1	1
154	C2	1
155	C3	n+1
158	C4	n
161	C5	n
162	C6	n
166	C7	1
167	C8	1

$$T(n) = C1 + C2 + C3(n+1) + C4(n) + C5(n) + C6(n) + C7 + C8$$
$$T(n) = C1 + C2 + C3 * n + C3 + C4 * n + C5 * n + C6 * n + C7 + C8$$
$$T(n) = (C3 + C4 + C5 + C6)n + (C1 + C2 + C3 + C7 + C8)$$

Complejidad: **O(n)**

Explicación: Esta función toma el valor inicial que sería cero y el final que es n-1. Donde se intercambian los valores si el valor almacenado es menor y este se repetirá n veces dado que tiene un loop for que va de 0 a n-1 que seria n veces dado que el último término automáticamente llegaría a ser el mayor.

```
182 int busquedaBinaria(vector<Bitacora> list, int data) {
183   int izquierda = 0;
184   int derecha = list.size() - 1;
185   int mid;
186   while (izquierda <= derecha) {
187     // Calcular el punto medio
188     mid = (izquierda + derecha) / 2;
189
190     //Compara si el dato del medio es igual al dato
      considerado
191     if (list[mid].key == data) {
192       // Retorna la posicion mid
193       return mid;
194     } else {
195       // Comparacion que determina si el dato es menor
      al del medio de la lista
196       if (data < list[mid].key) {
197         //Cambio de limite derecho a mid menos uno
198         derecha = mid - 1;
199       } else {
200         //Cambio de limite izquierdo a mid mas uno
201         izquierda = mid + 1;
202       }
203     }
204   }
205   return mid;
206 }
```

Linea	Comp	Caso
183	1	C1
184	1	C2
185	1	C3
186	1+T(n/2)	C4
191	1	C5
193	1	C6
196	1	C7
198	n-1	C8
201	1	C9
205	n-1	C10
186	1	C11

Nota: todo dentro del while se repite hasta que el el while concluya por lo menos estas líneas tienen prácticamente una complejidad similar al de la C4.

Donde todos los C# se suman y se vuelven una constante y se pueden ignorar, dado que se evalúa el peor caso y se lleva al límite donde $\lim_{n \to \infty}$.

Casos para C8

$$T(n) = 1\,,\ if\ n = 1$$
$$T(n) = 1 + T(n/2),\ if\ n > 1$$

Ya que esta es una función recursiva, tenemos que encontrar una solución general a través de un patrón.

$$T(n) = 1 + T(n/2)$$
$$= 1 + 1 + T(n/2\ /2)$$
$$= 2 + T(n/4)$$
$$= 1 + 2 + T(n/4/2)$$
$$= 3 + T(n/8)$$
$$\ldots$$

Solución General

$$T(n) = k + T(n/2^{k})$$

Usando el caso base.

$$n/2^{k} = 1$$
$$log_2 n = k$$

Sustituímos k:

$$T(n) = log_2 n + T(n/2^{log_2 n})$$
$$T(n) = log_2 n + 1$$
$$T(n) = log_2 n\ + 1$$
$$T(n) = log_2 n$$

Por lo tanto, la complejidad de la recursión es:

$$O(log_2 n)$$

Explicación: La búsqueda binaria utiliza el loop while como un recurso donde el valor mínimo y máximo se está aproximando a uno exacto que es el que se está buscando.

Reflexión:

Se consideró trabajar con mergesort, pero no veiamos ventajas a su uso sobre quicksort y este hacia el trabajo necesario de una manera más eficiente. La búsqueda binaria fue uno de los aspectos más difíciles de la actividad dado que se tuvo que pensar en un sistema donde se podía buscar hasta el segundo (aunque no fue requerido) para tener un programa más completo.

A lo largo de esta actividad se pudo trabajar con los diferentes algoritmos que fueron estudiados esta semana. Entender más a fondo cómo utilizarlos en contextos prácticos y tener que modificar los para poder obtener resultados precisos y llegar a una solución esperada. Se pudo entender a mayor profundidad su importancia la importancia de la

eficiencia de cada uno y entender cuando aplica cada uno y por qué debería ser descartada la opción de algunos sobre otro dependiendo del caso. Se trabajó con dos algoritmos que son quicksort y búsqueda binaria.

Referencias:

GeeksforGeeks. (2021a, junio 28). *Binary Search*. https://www.geeksforgeeks.org/binary-search/

GeeksforGeeks. (2021b, agosto 10). *QuickSort*. https://www.geeksforgeeks.org/quick-sort/