ANÁLISIS COMPLEJIDAD

Análisis de complejidad temporal del algoritmo HeapSort

- Peor Caso:

El peor caso se da cuando el arreglo ya está ordenado de manera ascendente o todos los elementos son iguales, esto hace que el pivote siempre quede en el extremo derecho de la partición, y el problema solo se reduce en uno de tamaño.

Línea	Instrucción	# veces que se ejecuta
	Heapsort(int[] arr)	
0	n = arr.Length	1
1	for (int i = n / 2 - 1; i >= 0; i--)	Lg n
2	heapify(arr, n, i);	O(nLgn)
3	for (int i = n - 1; i >= 0; i--)	n
4	temp = arr[0];	n-1
5	arr[0] = arr[i];	n-1
6	arr[i] = temp;	n-1
7	heapify(arr, i, 0);	O(nLgn)

T(n) = C1 + C2Lg n + C3nLg n + C4n + C5n-1 + C6n-1 + C7n-1 + C8nLg n

= C1 + Lg nC2 + nLgn(C3 + C8) + C4n + n – 1 (C5 + C6 + C7)

En donde la complejidad temporal en el peor caso es T(n) = O(n Lg n)

Línea	Instrucción	# veces que se ejecuta
	heapify(int[] arr, int n, int i)	
0	largest = i;	1
1	l = 2 * i + 1;	1
2	r = 2 * i + 2;	1
3	if (l < n && arr[l] > arr[largest])	1
4	largest = l;	1
5	if (r < n && arr[r] > arr[largest])	1

6	largest = r;	1
7	if (largest != i)	1
8	swap = arr[i];	1
9	arr[i] = arr[largest];	1
10	arr[largest] = swap;	1
11	heapify(arr, n, largest);	n Lg n

T(n) = C1 + C2 + C3 + C4 + C5 + C6 + C7 + C8 + C9 + C10 + C11 + C12Lg n

En donde la complejidad temporal en el peor caso T(n) = O (n Lg n)

Análisis de complejidad espacial del algoritmo HeapSort

El algoritmo se encarga de organizar los datos para que formen una pila en su lugar, con el elemento más pequeño en la parte posterior. Luego intercambia el último elemento del montón con el primero de este, para después barajar dicho elemento hacia abajo del montón hasta que esté en una nueva posición adecuada y haga de él montón uno nuevo mínimo con el elemento mas pequeño que queda en la última posición de la matriz.

Para la ejecución de este algoritmo solo se necesita O(1) de espacio ya que el montón se arma adentro del arreglo para poder ordenarse.

Análisis de complejidad temporal del algoritmo CombSort

- Peor Caso:

El peor caso esta probado usando el método de la incomprensibilidad basado en la complejidad Kolmogorov dando como resultado una complejidad temporal de O(n^2), sin embargo en el mejor de los casos dado que el combSort es similar al algoritmo de ordenamiento burbuja el cual itera sobre la lista varias veces, intercambiando los elementos que están fuera de orden sucesivamente, con la diferencia de que este empieza buscando los elementos con ciertos números de índices llamadas brechas definidas por (n/c) donde n es el numero de elementos y c el factor de encogimiento el cual es usualmente 1.3. Después de cada iteración, este numero se divide nuevamente por c y se anula hasta que finalmente el algoritmo empiece a mirar los elementos adyacentes, dando como una complejidad de O(nlogn)

Línea	Instrucción	# veces que se ejecuta
	Combsort(int[] array)	
0	n = array.Length;	1
1	gap = n;	1
2	swapped = true;	1
3	while (gap != 1 || swapped == true)	n
4	gap = getNextGap(gap);	n – 1
5	swapped = false;	n -1
6	for (int i = 0; i < n - gap; i++)	n^2 – 1
7	if (array[i] > array[i + gap])	n^2 – 2
8	int temp = array[i];	n^2 – 2
9	array[i] = array[i + gap];	n^2 – 2
10	array[i + gap] = temp;	n^2 – 2
11	swapped = true;	n^2 – 2
12	return array;	1

T(n) = C1 + C2 + C3 + C4n + C5(n – 1) + C6(n – 1) + C7(n^2 – 1) + C8(n^2 – 2) + C9(n^2 – 2) + C10(n^2 – 2) + C11(n^2 – 2) + C12(n^2 – 2) + C13

= C1 + C2 + C3 + C4 + (n – 1) (C5 + C6) + (n^2 – 1) C7 + (n^2 – 2) (C8 + C9 + C10 + C11 + C12)

+ C13

En donde la complejidad temporal en el peor caso T (n) = O (n^2)

Línea	Instrucción	# veces que se ejecuta
	getNextGap(int gap)	
0	gap = (gap * 10) / 13;	1
1	if (gap < 1)	1
2	return 1;	1
3	return gap;	1

T(n) = C1 + C2 + C3 + C4

En donde la complejidad temporal en el peor caso T (n) = O(1)

Análisis de complejidad espacial del algoritmo CombSort

El algoritmo se encarga de eliminar pequeños valores cerca del final de la lista, el espacio se inicia con la longitud de la lista a ordenar dividida por el factor de encogimiento, y la lista se ordenará en base a dicho valor. Después el espacio se divide por el factor de encogimiento una vez más, la lista se ordena con este nuevo espacio y el proceso se repite hasta que este sea 1 y la lista este completamente ordenada. Para este algoritmo, el espacio usado se define con O(1) ya que su ordenamiento se realiza en base dentro del mismo arreglo.

Análisis de complejidad temporal del algoritmo MergeSort

- Peor Caso:

El tiempo total para el algoritmo de mergeSort es la suma de los tiempos de mezcla para todos los niveles. Si hay l niveles en el árbol, el tiempo total del algoritmo seria l * cn. El algoritmo comienza con subproblemas de tamaño n y se reduce repetidamente a la mitad hasta llegar a subproblemas de tamaño 1, después analizándolo con la búsqueda binaria podemos obtener que la variable l se reduce a la expresión l = cn*(log2n + 1). Usando la connotación big-Θ se descarta el termino menor (+1) y la constante del coeficiente (c) dándonos un tiempo de O(nlog2n).

Línea	Instrucción	# veces que se ejecuta
	merge(int[] arr, int l, int m, int r)	
0	n1 = m - l + 1;	1
1	n2 = r - m;	1
2	L = new int[n1];	1
3	R = new int[n2];	n
4	for (int n = 0; n < n1; n++)	n – 1
5	L[n] = arr[l + n];	n - 2
6	for (int n = 0; n < n2; n++)	n – 1
7	R[n] = arr[m + 1 + n];	n – 2

8	int i = 0;	n – 2
9	int j = 0;	n – 2
10	int k = l;	n – 2
11	while (i < n1 && j < n2)	n – 1
12	if (L[i] <= R[j])	n - 2
13	arr[k] = L[i];	n - 2
14	i++;	n - 2
15	while (i < n1)	n – 1
16	arr[k] = L[i];	n - 2
17	i++;	n - 2
18	k++;;	n - 2
19	while (j < n2)	n - 1
20	arr[k] = R[j];	n - 2
21	j++;	n - 2
22	k++;	n - 2

T(n) = C1 + C2 + C3 + nC4 + (n-1)C5 + (n-2)C6 + (n-1)C7 + (n-2)C8 + (n-2)C9 + (n-2)C10 + (n-2)11 + (n-1)C12 + (n-2)C13 + (n-2)C14 + (n-2)C15 + (n-1)C16 + (n-2)C17 + (n-2)C18 + (n-2)C19 + (n-1)C20 + (n-2)C21 + (n-2)C22 + (n-2)C23

= C1 + C2 + C3 + nC4 + (n-1) (C5 + C7 +C12 + C16 +C20) + (n – 2) (C6 + C8 + C9 + C10 + C11 + C13 + C14 + C15 + C17 + C18 +C19 + C21 + C22 + C23)

La complejidad para el algoritmo merge en el peor caso es O(n)

Línea	**Instrucción**	**# veces que se ejecuta**
	sort(int[] arr, int l, int r)	
0	if (l < r)	n
1	int m = (l + r) / 2;	n
2	sort(arr, l, m);	n log2n
3	sort(arr, m + 1, r);	n log2n

4	merge(arr, l, m, r);	O(n)
5	return arr;	1

$T(n) = C1n + C2n + C3\ n \log_2 n + C4\ n \log_2 n + C5O(n) + C6$

$= n (C1 + C2) + n \log_2 n (C3 + C4) + O(n)C5 + C6$

La complejidad para el algoritmo sort en el peor caso es $O(n \log_2 n)$

Análisis de complejidad espacial del algoritmo MergeSort

El algoritmo mergeSort es recursivo, por lo que su espacio ocupa una pila O(logn), tanto para la matriz que usa como para los casos de la lista vinculada. Dicho esto, cabe aclarar que para el caso de la matriz también asigna un espacio O(n) adicional, el cual domina el espacio O(logn) que es requerido para la pila. Por lo que se puede concluir que el espacio ocupado del mergeSort en una matriz es O(n) mientras que en una lista seria de O(logn)