Bubble Sort

El algoritmo Bubble Sort compara los elementos de un arreglo repetidamente para ordenarlo. Bubble Sort va comparando parejas, si el elemento en n es mayor que el elemento en n+1 el algoritmo intercambia las posiciones de estos dos elementos. El algoritmo recorre n veces el arreglo y en cada recorrido hace n-1 comparaciones.

Tomemos como ejemplo el siguiente arreglo [5, 1, 6, 2, 9, 5, 10]

Primera iteración:
[5, 1, 6, 2, 9, 5, 10]
Segunda iteración:
[1,5,2,6,5,9,10]
Tercera iteración:
[1, 2, 5, 5, 6, 9, 10]
Cuarta iteración:
[1, 2, 5, 5, 6, 9, 10]
Quinta iteración:
[1, 2, 5, 5, 6, 9, 10]
Sexta iteración:
[1, 2, 5, 5, 6, 9, 10]
Séptima iteración:
[1, 2, 5, 5, 6, 9, 10]

El arreglo ordenado es [1, 2, 5, 5, 6, 9, 10].

Complejidad

Bubble Sort hace n-1 operaciones en el mejor de los casos, es decir, cuando el arreglo ingresado ya está ordenado, por lo tanto su complejidad en el mejor de los casos es de $O(n)$. Sin embargo, en el peor de los casos hace $n * (n - 1)$ operaciones que resulta en $n^2 - n$ operaciones, por lo que su complejidad se vuelve $O(n^2)$.

Quick Sort

Quick sort es un algoritmo de ordenamiento recursivo que se basa en la división del arreglo a partir de un valor conocido como *pivote* el cual se calcula de manera que sea lo más cercano a la mitad del arreglo para así comparar los números del arreglo. La función del pivote es mandarlo al final de arreglo y empezar a checar desde el inicio si es un número de la izquierda o de la derecha siendo que los de la izquierda son menores al pivote y los de la derecha son mayores al pivote para hacer que cada lado del pivote se arregle comparando cada término desde la posición cero del arreglo hasta una antes del pivote. Cuando el valor de la izquierda es mayor que el de la derecha es cuando se termina de comparar y se cambia el pivote con el último número leído de izquierda.

[5, 1, 6, 2, 9, 5, 10] el pivote empieza en 10
[5, 1, 6, 2, 9, 5, 10] se compara con todos y se ve que es el mayor el mismo 10
[5, 1, 6, 2, 9, 5, 10] ahora el pivote es uno menos en el arreglo, es 5
[5, 1, 6, 2, 9, 5, 10] se compara con 5, 1, 6 y 2. checa que el número 2 (valor de la izquierda es mayor al de la derecha) y lo cambia.
[5, 1, 2, 5, 9, 6, 10] el número izquierda cambia a la posición con el pivote
[1, 2, 5, 5, 9, 6, 10] el pivote ahora es 6
[1, 2, 5, 5, 9, 6, 10] se compara el 6 con 1,2,5,5,9. El 9 es mayor al pivote para cambiarlo
[1, 2, 5, 5, 6, 9, 10] el arreglo está ordenado

Complejidad

En el peor caso O(n^2) y depende de como se acomode el pivote de manera eficiente y eso es si se tienen que hacer las comparaciones de un número que ya es el más grande con los demás para volver a que está en la posición correcta por ejemplo con el arreglo [5, 1, 6, 2, 9, 5, 10] cuando tomamos que el pivote es 10 se compara con todos los demás pero el más alto sigue siendo el pivote por lo que se tiene que hacer una comparación completa de todos los números con ningún cambio haciendo poco eficiente el código. En el caso comun seria de 0(nlog(n)) en el pivote en medio y hacer los pivotes en medio al mismo tiempo.

Radix Sort

El código de ordenamiento *Radix Sort* o también conocido como ordenamiento por raíz es un código que dado un arreglo los va ordenando respecto al dígito inicial de cada uno de los números a comparar yendo de décima en décima.El problema de *Radix sort* es la incapacidad de ordenar números negativos ya que no puede comparar el símbolo negativo con otros números y si se requiere organizar números negativos se vuelve obligatorio el añadir un paso elemental.

El algoritmo comienza calculando el número máximo de nuestro arreglo para saber cuántas veces vamos a realizar counting sort. Se procede a hacer counting sort, que consiste en generar un arreglo con valores del 0 al 9 y después calcula cuántos valores de cada índice tiene el arreglo que se quiere ordenar para posteriormente hacer una suma de estos valores modificando el arreglo count, el arreglo count usa el algoritmo llamado counting sort para poder acomodar el arreglo temporal count con los índices de nuestro arreglo original.

Tomemos como ejemplo el siguiente arreglo [5, 1, 6, 2, 9, 5, 10]

Arreglo original	[5, 1, 6, 2, 9, 5, 10]
Arreglo count	[1, 1, 1, 0, 0, 2, 1, 0, 0, 1]
Suma del arreglo count	[1, 2, 3, 3, 3, 5, 6, 6, 6, 7]
Se ordena el arreglo	[10, 1, 2, 5, 5, 6, 9]
Arreglo count	[6, 1, 0, 0, 0, 0, 0, 0, 0, 0]
Suma de arreglo count	[6, 7, 7, 7, 7, 7, 7, 7, 7, 7]
Se ordena el arreglo	[1, 2, 5, 5, 6, 9, 10]
Arreglo ordenado:	[1, 2, 5, 5, 6, 9, 10]

En este caso, se repite el counting sort ya que el arreglo posee un 10 (valor máximo).

Complejidad

Tenemos “n” y “d” como variables del código, n es la cantidad de números dentro del arreglo y d es la cantidad de dígitos para representar cada número, también tenemos que tener una variable RADIX que nos daría la base 10 ya que trabajamos con números base 10. para radix sort es O((n+b)(d)) ya que para cada paso elemental se tiene que multiplicar la cantidad de dígitos por la suma de números del arreglo y la base ya que va de 0 a 9. Solo es rápido si la cantidad de dígitos es limitada, si “b” es menor a “n” sigue siendo rápido.

Heap Sort

Heap sort es un algoritmo de ordenamiento basado en la estructura de datos dada por un árbol binario. El algoritmo hace un paso elemental recursivo llamado *binary heap* que construye un árbol binario con el arreglo, dado tal que su raíz sea el mayor número del arreglo. Una vez creado este árbol también llamado binary max heap, intercambiamos la raíz con el último nodo del árbol y reducimos el tamaño del árbol en 1. De esta manera aseguramos que el mayor valor queda guardado al final del arreglo. Repetimos estos pasos mientras el tamaño del heap sea mayor a 1.

Tomemos como ejemplo el siguiente arreglo [5, 1, 6, 2, 9, 5, 10]

Antes del intercambio [10, 9, 6, 2, 1, 5, 5]
Después del intercambio [5, 9, 6, 2, 1, 5, 10]
Antes del intercambio [9, 5, 6, 2, 1, 5, 10]
Después del intercambio [5, 5, 6, 2, 1, 9, 10]
Antes del intercambio [6, 5, 5, 2, 1, 9, 10]
Después del intercambio [1, 5, 5, 2, 6, 9, 10]
Antes del intercambio [5, 2, 5, 1, 6, 9, 10]
Después del intercambio [1, 2, 5, 5, 6, 9, 10]
Antes del intercambio [5, 2, 1, 5, 6, 9, 10]
Después del intercambio [1, 2, 5, 5, 6, 9, 10]
Antes del intercambio [2, 1, 5, 5, 6, 9, 10]
Después del intercambio [1, 2, 5, 5, 6, 9, 10]
Arreglo ordenado: [1, 2, 5, 5, 6, 9, 10]

Complejidad:
La construcción del heap toma $log(n)$ ya que se trata de un árbol binario. Así mismo, el proceso consta de n elementos, definiendo un tiempo de $nlog(n)$.
Aunado a esto, ordenar el arreglo tarda lo mismo que la construcción del heap por lo que su complejidad es $nlog(n)$.
Como ambos procesos tienen el mismo orden (construir y ordenar), concluimos que el orden final del HeapSort es $O(nlog(n))$.

Merge Sort

MergeSort es un algoritmo que usa el método “divide y vencerás” para ordenar un arreglo. El algoritmo recibe un arreglo y comienza a dividir el arreglo por la mitad de manera recursiva hasta que la cardinalidad de los arreglos generados es 1, entonces comienza el proceso de “merge” donde comienza a unir los arreglos generados en orden hasta obtener un arreglo final ordenado.

Tomemos como ejemplo el siguiente arreglo [5,1,6,2,9,5,10]

El arreglo comienza a dividirse de la siguiente manera:

[5, 1, 6] [2, 9, 5, 10]
[5] [1, 6] [2, 9] [5, 10]
[5] [1] [6] [2] [9] [5] [10] Cardinalidad de 1
[5] [1, 6] [2, 9] [5, 10]
[1, 5, 6] [2, 5, 9,10]
[1, 2, 5, 5, 9, 5, 10]

Arreglo ordenado: [1, 2, 5, 5, 9, 5, 10]

Complejidad

Los pasos elementales que realiza MergeSort son la división del arreglo y la combinación de los subarreglos. La combinación de los arreglos ocurre en tiempo lineal n veces. La división del arreglo por otra parte ocurre en $logn$. Empezamos con un arreglo de tamaño n que comenzamos a dividir hasta tener subarreglos de tamaño 1, por lo que la división del arreglo nos queda como $\frac{n}{2^k}$ donde k es la potencia que usamos para subdividir el arreglo.
Asumiendo que $\frac{n}{2^k} = 1$ tenemos que $k = log_2 n$. Por lo tanto, la complejidad de nuestro algoritmo se define como $O(nlogn)$

Bibliografía

Radixsort:
Kumra, M. (s.f). HeapSort, de GeeksforGeeks, sitio web: sitio web:
https://www.geeksforgeeks.org/radix-sort/

Quicksort:
Kumra, M. (s.f). HeapSort, de GeeksforGeeks, sitio web:
https://www.geeksforgeeks.org/can-quicksort-implemented-onlogn-worst-case-time-complexity/

Kumra, M. (s.f). HeapSort, de GeeksforGeeks, sitio web:
https://www.geeksforgeeks.org/quick-sort/

Bubble Sort:
Sin autor. Bubble Sort, de GeeksforGeeks Sitio web:
https://www.geeksforgeeks.org/bubble-sort/

Heapsort:
Kumra, M. (s.f). HeapSort, de GeeksforGeeks Sitio web:
https://www.geeksforgeeks.org/heap-sort/

Mergesort:
Sin autor. Merge Sort, de GeeksfoGgeeks Sitio web:
https://www.geeksforgeeks.org/merge-sort/