Héctor Manuel Takami Flores
A01377647 – ISC11 Análisis y Diseño de Algoritmos
(Ago 2017, Grupo 1)
Proyecto Primer Parcial: Algoritmos de Ordenamiento

Bubble Sort

Es un sencillo algoritmo de ordenamiento. Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado.

Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada. Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas".

También es conocido como el método del intercambio directo. Dado que solo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo uno de los más sencillo de implementar. [1]

Al algoritmo de la burbuja, para ordenar un vector de n términos, tiene que realizar en promedio el mismo número de comparaciones:

$$A(n) = \frac{n^2 - n}{2}$$

Esto es, el número de comparaciones $f(n)$ no depende del orden de los términos, si no del número de términos, siendo así que para cada caso su complejidad estará dada por:

$$B(n) = O(n) \quad , \quad W(n) = O(n^2)$$

Héctor Manuel Takami Flores
A01377647 – ISC11 Análisis y Diseño de Algoritmos
(Ago 2017, Grupo 1)
Proyecto Primer Parcial: Algoritmos de Ordenamiento

Quick Sort

Es un algoritmo creado por el científico británico en computación C. A. R. Hoare, basado en la técnica de divide y vencerás, que permite, en promedio, ordenar n elementos en un tiempo proporcional a n log n. El algoritmo trabaja de la siguiente forma:

1. Se elige un elemento de la lista de elementos a ordenar, al que llamaremos pivote.
2. Reacomodamos los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.
3. La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
4. Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

Como se puede suponer, la eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.

En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño. En este caso, el orden de complejidad del algoritmo es $O(n\ log_2 n)$.

En el peor caso, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de $O(n^2)$. El peor caso dependerá de la implementación del algoritmo, aunque habitualmente ocurre en listas que se encuentran ordenadas, o casi ordenadas. Pero principalmente depende del pivote, si por ejemplo el algoritmo implementado toma como pivote siempre el primer elemento del array, y el array que le pasamos está ordenado, siempre va a generar a su izquierda un array vacío, lo que es ineficiente. [2]

No es extraño, pues, que la mayoría de optimizaciones que se aplican al algoritmo se centren en la elección del pivote. La complejidad del algoritmo para caso es la siguiente:

$$B(n) = O(n\ log_2 n) \quad , \quad A(n) = O(n\ log_2 n) \quad , \quad W(n) = O(n^2)$$

Héctor Manuel Takami Flores
A01377647 – ISC11 Análisis y Diseño de Algoritmos
(Ago 2017, Grupo 1)
Proyecto Primer Parcial: Algoritmos de Ordenamiento

Radix Sort

Es un algoritmo de ordenamiento que ordena enteros procesando sus dígitos de forma individual. La mayor parte de los ordenadores digitales representan internamente todos sus datos como representaciones electrónicas de números binarios, por lo que procesar los dígitos de las representaciones de enteros por representaciones de grupos de dígitos binarios es lo más conveniente.

Existen dos clasificaciones de radix sort: el de dígito menos significativo (LSD) y el de dígito más significativo (MSD). Radix sort LSD procesa las representaciones de enteros empezando por el dígito menos significativo y moviéndose hacia el dígito más significativo. Radix sort MSD trabaja en sentido contrario.

Radix Sort es un algoritmo de clasificación no comparativo con complejidad asintótica $O(nd)$. Es uno de los algoritmos de clasificación lineal más eficientes y rápidos. El tipo Radix se desarrolló para ordenar enteros grandes. Cada entero se trata como una cadena de dígitos por lo que también puede llamarlo como algoritmo de clasificación de cadenas.

El límite inferior para el algoritmo de clasificación basado en comparación es $O(n\ log_2 n)$ como tipo de mezcla, ordenación rápida, ordenación de heap. Así que el tipo de radix es eficiente como un algoritmo de clasificación de comparación hasta que el número de dígitos sea menor que $log_2 n$. No se puede utilizar la ordenación de conteo si un rango de valores clave es grande (supongamos que el rango es de 1 a n^2), por lo que la ordenación de raíz es la mejor opción para ordenar en tiempo lineal.

En ordenación de raíz, primero ordenamos los elementos basados en el último dígito (dígito menos significativo). Luego el resultado se ordena de nuevo por segundo dígito, continúe este proceso para todos los dígitos hasta que alcancemos el dígito más significativo. Utilizamos la ordenación de conteo para ordenar elementos de cada dígito, por lo que la complejidad en el tiempo es $O(nd)$. [3]

La complejidad el algoritmo esta dada en cada caso por:

$$B(n) = O(n\ log_2 n) \quad , \qquad A(n) = O(\ log_2 n)\ , \qquad W(n) = O(nd)$$

Héctor Manuel Takami Flores
A01377647 – ISC11 Análisis y Diseño de Algoritmos
(Ago 2017, Grupo 1)
Proyecto Primer Parcial: Algoritmos de Ordenamiento

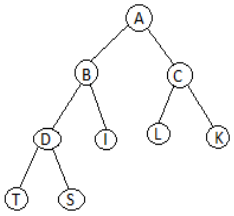
Heap Sort

Heap Sort es uno de los mejores métodos de clasificación que se encuentra y sin escenarios cuadráticos en el peor de los casos. El algoritmo de ordenación Heap se divide en dos partes básicas:

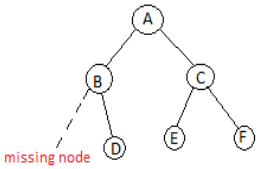
1. Creación de un montón de la lista sin clasificar.
2. Creación una matriz ordenada eliminando repetidamente el elemento más grande o más pequeño del montón, e insertándolo en la matriz. El montón se reconstruye después de cada extracción.

Heap es una estructura de datos basada en árboles especial que satisface las siguientes propiedades especiales de montón:

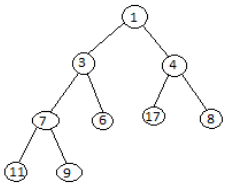
- ***Propiedad de forma***:
  La estructura de datos del arreglo es siempre un árbol binario completo, lo que significa que todos los niveles del árbol están llenos.
- ***Propiedad Heap***: Todos los nodos son [mayores o iguales] o [menos que o iguales] a cada uno de sus hijos. Si los nodos principales son mayores que sus nodos hijos, heap se denomina Max-Heap y si los nodos principales son más pequeños que sus nodos hijos, heap se denomina Min-Heap.



Complete Binary Tree

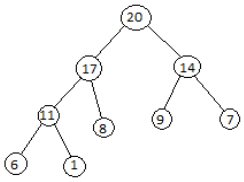


In-Complete Binary Tree



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.



Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

Inicialmente al recibir una lista no ordenada, el primer paso en ordenar heap es crear una estructura de datos Heap (Max-Heap o Min-Heap). Una vez que se construye el montón, el primer elemento del Heap es mayor o menor (dependiendo de Max-Heap o Min-Heap), por lo que ponemos el primer elemento del montón en nuestra matriz. Luego volvemos a hacer montón utilizando los elementos restantes, para recoger de nuevo el primer elemento de la pila y ponerlo en la matriz. Seguimos haciendo lo mismo repetidamente hasta que tenemos la lista ordenada completa en nuestra matriz. [4]

La complejidad del algoritmo en cada uno de sus casos es la siguiente:

$$B(n) = O(n\ log_2 n)\ , \qquad A(n) = O(n\ log_2 n)\ , \qquad W(n) = O(n\ log_2 n)$$

Héctor Manuel Takami Flores
A01377647 – ISC11 Análisis y Diseño de Algoritmos
(Ago 2017, Grupo 1)
Proyecto Primer Parcial: Algoritmos de Ordenamiento

Merge Sort

Es un algoritmo de ordenamiento externo estable basado en la técnica divide y vencerás. Es de complejidad $O(n\ log_2 n)$.

Conceptualmente, el ordenamiento por mezcla funciona de la siguiente manera:

Si la longitud de la lista es 0 ó 1, entonces ya está ordenada. En otro caso:

1. Ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla.
2. Mezclar las dos sublistas en una sola lista ordenada.

Conceptualmente, un algoritmo merge sort funciona de la siguiente manera:

- Divide la lista sin clasificar en n sublistas, cada una contiene 1 elemento (una lista de 1 elemento se considera ordenada).
- Reúne repetidamente sublistas para producir nuevas sublistas ordenadas hasta que queden sólo 1 sublista. Esta será la lista ordenada. [5]

La complejidad de este algoritmo en cada caso es:

$$B(n) = O(n\ log_2 n)\ \ , A(n) = O(n\ log_2 n),\ \ W(n) = \ O(n\ log_2 n)$$

Referencias

[1] S/A. (19 de agosto de 2017). *Wikipedia*. Obtenido de https://en.wikipedia.org/wiki/Bubble_sort

[2] Dartmouth Computer Science. (2017). *KhanAcademy*. Obtenido de https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/overview-of-quicksort

[3] S/A. (2015). *GeeksForGeeks*. Obtenido de http://www.geeksforgeeks.org/radix-sort/

[4] S/A. (2017). *StudyTonight.com*. Obtenido de http://www.studytonight.com/data-structures/heap-sort

[5] S/A. (03 de septiembre de 2017). *Wikipedia*. Obtenido de https://en.wikipedia.org/wiki/Merge_sort

Cormen, L. R. (2001). Introduction to Algorithms . MIT Press, 2009.