Tecnológico de Monterrey

Proyecto Final

Análisis y Diseño de Algoritmos

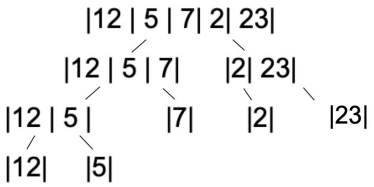
Dr. Salvador Elías Venegas Andraca

Rubén Sánchez A01021759

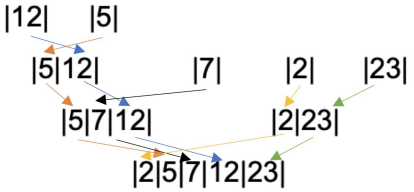
Cristopher Cejudo A01025468

Merge Sort

Merge Sort es uno de de los algoritmos más óptimos que se conoce, y es un algoritmo del tipo Divide and Conquer, pues consiste en dividir un arreglo en dos partes recursivamente hasta que solo quede un elemento en el arreglo, y, posteriormente, comparar los elementos de los arreglos uno a uno, para entonces dado un orden definido volver a unir los arreglos divididos hasta volver a tener un arreglo del tamaño original pero con sus elementos ordenados. Por ejemplo, en el siguiente arreglo desordenado de números: 12 | 5 | 7| 2| 23, queriéndolo ordenar de menor a mayor, MergeSort primero dividiría todo el arreglo de modo:



Y, posteriormente, el algoritmo volvería a unir los arreglos de la siguiente manera:

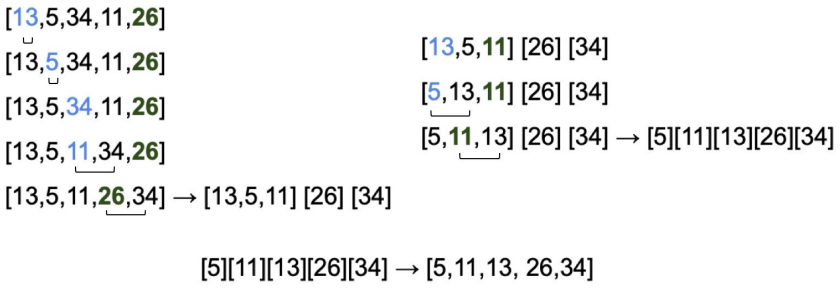


Es importante remarcar que este algoritmo se comporta de manera estable, lo que significa que MergeSort no depende de los elementos y este siempre ejecutará el mismo número de pasos, y, por lo tanto, comprende la misma complejidad tanto en el peor, mejor y caso promedio. Dado que el algoritmo divide los arreglos en mitades, y posteriormente vuelve a unir la partes en un tiempo lineal, la complejidad de MergeSort se encuentra dentro de *O(nlogn)*.

A pesar de que MergeSort puede ser utilizado para ejecutar un ordenamiento a diferentes niveles de estructuras de datos, su aplicación más óptima se obtiene al querer ordenar listas ligadas debido a la alocación y manejo de memoria, el cual difiere al momento de ordenar un arreglo.

Quick Sort

QuickSort al igual que MergeSort, consiste en un algoritmo del tipo Divide and Conquer, sin embargo, a diferencia de este, QuickSort es un algoritmo que igualmente es llamado recursivamente para la división de un arreglo, pero utiliza un elemento como pivote para este propósito. La selección del pivote que QuickSort utiliza es completamente arbitraria, esto significa que cualquier elemento del arreglo puede cumplir la función de pivote. En el siguiente ejemplo se busca realizar un ordenamiento del menor al mayor número del arreglo [13,5,34,11,26] utilizando el último elemento como pivote.



Cada elemento del arreglo es comparado con el pivote, si el elemento encontrado es mayor, permanece en la misma posición, pero si el elemento encontrado es menor, entonces se realiza un intercambio entre este y el elemento mayor del pivote encontrado anteriormente (si no se ha encontrado un elemento mayor, entonces el elemento menor permanece en la misma posición). Por último, una vez terminadas las comparaciones, se realiza un último intercambio entre el pivote y el primer elemento mayor a este. De esta forma los elementos menores se encontrarán a la izquierda del pivote, y los mayores a la derecha del pivote. Se vuelve a realizar QuickSort en los arreglos izquierdo y derecho hasta que el arreglo se encuentra finalmente ordenado.

A diferencia de MergeSort, QuickSort es considerado un elemento inestable, pues depende completamente de los elementos, sin embargo llega a considerarse el algoritmo más rápido de ordenamiento. Tanto en el caso promedio como en el mejor caso, QuickSort se encuentra dentro de $O(nlogn)$. Sin embargo, en el peor caso (se elige siempre el mayor o el menor número como pivote), QuickSort se encuentra dentro de $O(n^2)$.

Bubble Sort

Bubble Sort es un algoritmo de ordenamiento que intercambia números adyacentes de forma repetitiva, si no están en el orden correcto, para que al final todos estén ordenados. En términos más claros, en cualquier recorrido por el arreglo de números, lo que hace Bubble Sort es checar los números de par en par para intercambiar los que no estén bien ordenados. Los pasos de Bubble Sort son (teniendo un arreglo A):

1- Comparar A[0] con A[1]. Si el número en A[0] es más grande, los elementos de A[0] y A[1] son intercambiados.

2- Moverse al siguiente par de números (A[1] y A[2], por ejemplo) e intercambiarlos si no están en el orden correcto. Esto se hace con cada par de números adyacentes en el arreglo. En un arreglo con n números, se pueden comparar n-1 parejas.

3- Hacer los pasos 1 y 3 “n” veces, donde n es el número de números en el arreglo.

Para poder calcular la complejidad de Bubble Sort, es importante definir el paso elemental y este es la comparación de números adyacentes. Tomando en cuenta los pasos previamente explicados, se podría afirmar que, en el peor caso, Bubble Sort hace n - 1 comparaciones n veces, pero, si se analiza al algoritmo más a fondo, es posible concluir que no es necesario hacer n-1 comparaciones en cada recorrido por el arreglo. Como después del primer recorrido por el arreglo el número que queda al final es el más grande del arreglo, ya no es necesario hacer una comparación en el último par del arreglo. Por lo tanto, en el primer recorrido se pueden hacer n-1 comparaciones, en el segundo n-2, en el tercero n-3 y así sucesivamente hasta sólo hacer una comparación. Entonces, el número de comparaciones que hace Bubble Sort es igual a la sumatoria de los números de 1 a n-1:

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$, $\frac{(n-1)(n-1+1)}{2} = \frac{n(n-1)}{2} = \frac{n2-n}{2} = n^2/2 - n/2$

Después de quitar las constantes y dejar sólo al término con la potencia más alta, siguiendo las propiedades de un análisis asintótico, se concluye que Bubble Sort

tiene una complejidad de $O(n^2)$ en el peor caso. En el mejor caso, Bubble Sort haría n-1 comparaciones que equivale a O(n).

Radix Sort

Radix Sort es un algoritmo de ordenamiento que ordena números basándose en sus dígitos, empezando por el dígito que está en la posición de las unidades y acabando con el dígito más significativo que tenga el mayor número del arreglo. Por ejemplo, si en un arreglo el número más grande es el 501, entonces Radix Sort, en el último recorrido por el arreglo, ordenará los números según el valor que tengan en el espacio de la centenas. Radix Sort usa a Counting Sort, otro algoritmo de ordenamiento, como subrutina. Lo que hace Counting Sort, cuando está dentro de Radix Sort, es ordenar números de acuerdo a un dígito en específico. Por ejemplo, si se hiciera Counting Sort con el dígito que está en las decenas, un arreglo quedaría así: 44, 105, 121, 23, 411 -> 1<u>0</u>5, 4<u>1</u>1, 1<u>2</u>1, <u>2</u>3, <u>4</u>4. En el ejemplo, aunque 121 es más grande que 23, 121 queda antes porque estaba antes en el arreglo original. Esta propiedad indica que Counting Sort es un algoritmo de ordenamiento estable (de lo contrario, Radix no podría funcionar). Entonces, lo que hace Radix Sort es correr Counting Sort en el arreglo por cada dígito existente en los números, empezando con las unidades, y al final, el arreglo queda ordenado.

Ejemplo:

44, 105, 3, 121, 4, 23, 411 -> 12<u>1</u>, 41<u>1</u>, <u>3</u>, 2<u>3</u>, 4<u>4</u>, <u>4</u>, 10<u>5</u>

121, 411, 3, 23, 44, 4, 104 -> <u>0</u>3, <u>0</u>4, 1<u>0</u>4, 4<u>1</u>1, 1<u>2</u>1, <u>2</u>3, <u>4</u>4

<u>0</u>3, <u>0</u>4, 1<u>0</u>4, 4<u>1</u>1, 1<u>2</u>1, <u>2</u>3, <u>4</u>4 -> <u>0</u>03, <u>0</u>04, <u>0</u>23, <u>0</u>44, <u>1</u>04, <u>1</u>21, <u>4</u>11

Para calcular la complejidad de Radix Sort, primero se tiene que tener en cuenta la complejidad de Counting Sort. Normalmente, la complejidad de Counting Sort sería O(n + k) donde n es la cantidad de números de entrada y k es el rango de números (si el número más grande fuera 1000, k sería 1001). Cuando es una subrutina de Radix Sort, en vez de ser equivalente al rango de números la k es equivalente a la base de los números. Si se usa base 10 para representar a los números, la complejidad de Counting Sort es O(n+b) donde b es igual a la base. El número de veces que se repetirá Counting Sort dependerá del número de dígitos diferentes

necesarios para ordenar el arreglo. Si el dígito más significativo es d, entonces Counting Sort se hace “d” veces y la complejidad de Radix Sort es de O(d(n+b)).

Heapsort

Heapsort es un algoritmo de ordenamiento que convierte a un arreglo en un heap (montículo) para poder ordenar los números del arreglo tratado. Un heap es un árbol binario en el que todos los nodos hoja, nodos sin hijos, se encuentran en los dos últimos niveles y en el que hay un nodo tal que este nodo tiene al menos un hijo y todos los nodos a su izquierda tienen dos hijos. Un max-heap es un heap en el que el elemento dentro de cada nodo padre es mayor a los elementos dentro de los hijos del nodo. Gracias a esta propiedad, el nodo raíz de un max-heap contiene el elemento más grande del heap. Para poder usar esta estructura como herramienta para ordenar números, esta se puede implementar en un arreglo que siga ciertas reglas como: la raíz está en el primer espacio, los hijos de un nodo se encuentran en los espacios 2i+1 y 2i+2 donde i es el índice del padre, y el padre de un nodo hijo está en el índice i/2. Entonces, lo que hace Heapsort es:

1- Poner los números en un heap y aplicar “heapify” en cada nodo para que el heap sea max-heap. Heapify es una subrutina recursiva en la que se checa si un nodo es más grande que sus hijos y, en caso contrario, su valor es intercambiado con el de su hijo más grande. Heapify se aplica en el hijo modificado y así sucesivamente.
2- Intercambiar la raíz del heap con el último espacio del heap.
3- Extraer el número en la última posición del heap y reducir el tamaño del heap.
4- Aplicar heapify en el nodo raíz.
5- Repetir los pasos 2, 3 y 4 hasta que sólo quede un nodo.

Los nodos extraídos durante el algoritmo acaban quedando en orden ascendente según sus valores y se puede decir que los números dados inicialmente ya están ordenados. En cuanto a la complejidad, crear el max-heap al principio tiene una complejidad de O(n logn) ya que se hacen “n” llamadas a heapify el cual tiene una complejidad de O(logn). Después, intercambiar nodos y reducir el heap tienen complejidad O(1). Los tres últimos pasos (intercambiar, reducir y heapify) se hacen

n-1 veces, así que la complejidad de Heapsort en el peor caso es: O(n logn) + (n-1)(O(1)+O(1)+O(logn)) = O(n logn) + (n)(O(logn)) = O(2(n logn)) = O(n logn).

Referencias

- Brilliant.org. (s.f.). Bubble Sort. Recuperado el 14 de mayo de 2019 desde https://brilliant.org/wiki/bubble-sort/
- CS Dojo. (2017). Radix Sort Algorithm Introduction in 5 Minutes. Recuperado el 14 de mayo de 2019 https://www.youtube.com/watch?v=XiuSW_mEn7g
- GeeksforGeeks. (s.f.). Merge Sort. Recuperado el 14 de mayo de 2019 https://www.geeksforgeeks.org/merge-sort/
- GeeksforGeeks. (s.f.). QuickSort. Recuperado el 14 de mayo de 2019 https://www.geeksforgeeks.org/quick-sort/
- Heapsort. (s.f.). Recuperado el 14 de mayo de 2019 desde https://www.ecured.cu/Heapsort
- Martínez, C. (2012). Algorítmica: Heaps y heapsort. Recuperado el 14 de mayo de 2019 desde http://algorithmics.lsi.upc.edu/docs/pqueues.pdf