Laboratorio 3

José Alejandro Guzmán Zamora

16 de agosto de 2018

- Problema de Heapsort
  Realice un cambio al algoritmo de Heapsort en el que no se destruya el heap pero aun así logre ordenar el arreglo.

  El algoritmo de Heapsort, al momento de entrar en el ciclo for, realiza una 'destrucción del heap' con la instrucción exchange. Esta instrucción se lleva a cabo para mover todos los valores mayores al final del arreglo conforme va avanzando el algoritmo. El siguiente pseudocódigo es el primer intento de hacer un cambio al algoritmo para no destruir el heap. Lo que hace el algoritmo es que aprovecha las propiedades del heap y aparta los valores mayores de cada nivel. Como cada nivel tiene valores mayores en comparación al siguiente nivel, se pueden ordenar por separado. Sin embargo, la cantidad de valores en cada nivel no es constante, esta aumenta en potencia de 2. Este incremento en la cantidad de valores causa que al final este algoritmo sea un ordenamiento de cada nivel:

```
HEAPSORTV2(A)
1    BUILD-MAX-HEAP(A)
2    for i from 2 to floor(lg(len(A))):
3       for j from (len(A)^(i-1)) to ((len(A)^i) - 1) :
4          for k from j to ((len(A)^i) - 1) :
5             if A[k] > A[j]:
6                exchange A[k] with A[j]
```

  Se puede notar a simple vista que el algoritmo es relativamente ineficiente. Basado en un análisis superficial, los loops interiores representan una complejidad de $O(n^2)$. Sin embargo, los niveles que se encuentran de primero, iteran sobre una cantidad de valores menores a n. Por esta razón se puede concluir que el running time del algoritmo completo es de $O(n^2)$;se asemeja mucho al algoritmo de Insertion Sort, con la diferencia de que el ordenamiento se hace por segmentos. No obstante la separación del arreglo no es muy útil.

Dependiendo de cuál sea el criterio de crítica para el término 'destrucción del heap' el siguiente algoritmo también puede aplicar a solucionar el problema expuesto. Lo que hace esta versión de Heapsort es que en lugar de hacer el exchange del primer y último valor, solo mueve el valor más grande del heap a otro arreglo. El problema es que al quitar el primer valor del heap, puede ser que no se cumplan las propiedades. Por esa razón es necesario hacer Heapify para que el primer valor se mueva hacia abajo basado en las otras cantidades del arreglo.

HEAPSORTV3(A)

```
1    BUILD-MAX-HEAP(A)
2    arrayf = []
6    for i from 1 to A.length():
3        arrayf.insert(A[1])
4        A = A[1 :]  /*slice para quitar el primer elemento del arreglo*/
5        MAX-HEAPIFY(A,1)
```

Este algoritmo tiene la misma complejidad que el Heapsort original, sin embargo, también lleva a cabo un desplazamiento en el heap por lo que no se puede considerar como una evasión total de la destrucción del heap.

- Preguntas de Quicksort
  Cuál es el running time de Quicksort cuando todos los valores son iguales?

  Cuando todos los valores son iguales, se sabe trivialmente que el arreglo ya está ordenado. No obstante, el algoritmo de Quicksort no puede determinar a priori si un arreglo está ordenado o no. Por esta razón el procedimiento se debe de llevar a cabo como en cualquier otra situación. Vale la pena mencionar que a pesar de que el algoritmo ya está ordenado, ocurre algo negativo en cuanto la eficiencia del algoritmo. El running time de Quicksort depende mayormente de la partición; es decir, una partición balanceada puede causar una complejidad de $O(n * lg(n))$ mientras que una partición desbalanceada puede causar un running time representativo del peor de los casos.
  Cuando todos los valores son iguales, la partición basada en utilizar el valor $A[r]$, siendo r el último valor del subarreglo, resulta en un desbalance total. La división de cada partición ocupa tiempo lineal individualmente. El conflicto es que como todos los valores son iguales, los subproblemas resultantes son uno de n-1 elementos y otro de ningún elemento. Al final la partición también se lleva a cabo en cantidad lineal, el running time en este caso es

$$O(n^2)$$

  Tanto esta situación en la que todos los valores son iguales como en la que el arreglo está completamente ordenado, ocurre el peor de los casos. Para estos, la idea de divide and conquer no tiene sentido.

Utilice el pseudocódigo para ilustrar la operación de partition en el siguiente arreglo.
$A = [13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11]$

Partition(A, 1, 12)
x = 11 , i = 0

|13 19 9 5 12 8 7 4 21 2 6 11
i j,p

Siempre se compara el valor de del arreglo en el índice j con el pivote, que en este caso es 11.

13 |19 9 5 12 8 7 4 21 2 6 11
i p j

Se comparó 19 con 11.

9| 19 |13 5 12 8 7 4 21 2 6 11
i,p j

El siguiente valor que se comparó fue 9, es menor a 11. Los valores entre j e i son mayores al pivote. Los valores de i a la izquierda son menores al pivote.

9 5| 13 |19 12 8 7 4 21 2 6 11
p i j

9 5| 13 19 |12 8 7 4 21 2 6 11
p i j

9 5 8| 19 12 |13 7 4 21 2 6 11
p i j

9 5 8 7| 12 13 |19 4 21 2 6 11
p i j

9 5 8 7 4| 13 19 |12 21 2 6 11
p i j

9 5 8 7 4| 13 19 12 |21 2 6 11
p i j

9 5 8 7 4 2| 19 12 21 |13 6 11
p i j

9 5 8 7 4 2 6| 12 21 13 |19 11
p i j

Intercambio final:

9 5 8 7 4 2 6 11 21 13 19 12

Por qué es más utilizado Quicksort y no Heapsort sabiendo que el worst-case running time de Heapsort es mejor?

A pesar de que en el peor de los casos Quicksort toma tiempo de $O(n^2)$, un análisis más profundo demuestra que en el mejor de los casos, y más importante, en los casos promedios, la complejidad es de $O(n * lg(n))$. Además, es fundamental considerar que al hacer el análisis asintótico, se evita utilizar constantes; sin embargo, las que se eliminan en el análisis promedio de Quicksort son relativamente pequeñas. Otro aspecto importante que se relaciona con lo práctico del algoritmo es que ordena sobre el mismo arreglo, puede trabajar sin requerir mucha memoria.

- Quicksort

Realice una implementación de Quicksort en Python desarrollando su propio algoritmo de partición.