Laboratorio 3 | Análisis Asintótico y modificación de algoritmos

Geordie Quiroa

August 16, 2018

1 - Problema de Heapsort

Realicen un cambio al algoritmo de heapsort donde no se destruya el heap pero aun así logre ordenar el arreglo.

Escribirlo en forma de pseudocodigo.

¿Es el running-time de este algoritmo mejor, igual, o peor que el de heapsort?

Algoritmo modificado

Algorithm 1 Heapsort alterado

```
1: procedure HEAPSORT(A)
2:     n ← A.length/2
3:     Build − Max − Heap(A)
4:     for i = 1 to n do
5:         Max − Heapify(A, i)
```

Algorithm 2 Max-Heapify modificado

```
1: procedure ITERATIVE-HEAPIFY(A, i)
2:     A is an array of integers
3:     for i ← 1 to (A.length/2) do
4:         l ← Left(i)
5:         r ← Right(i)
6:         if l <= A.length and A[l] > A[i] then
7:             largest ← l
8:         else
9:             largest ← i
10:        if r <= A.length and A[r] > A[largest] then
11:            largest ← r
12:        if largest != i then
13:            if subtree − verify − heap(largest) == 1 -esta función
    verifica que los valores del subtree sean menores al largest
    then
14:                exchange A[i] ← A[largest]
15:                Max − heapify(A, largest)
```

Running Time
El running time de este algoritmo es igual que el original, ya que la nueva función subtree-verify-heap(largest) va a evaluar si los subtrees del nodo largest cumplen con la propiedad de max-heap para

poder intercambiar el valor del nodo i con el largest y no romper la propiedad de max heap, si no es así, ejecutaría un max-heap para los subtrees de tal forma que se cumpla la propiedad. Debido a que el primer ciclo se recorre n/2 veces, y el nuevo max heapify contiene una función que evalúa los subtrees para ver si los hijos cumplen con la propiedad para realizar el cambio (logn). Por lo tanto, el running time se mantiene como $\theta(\mathbf{log}(\mathbf{n}))$.

2 - Quicksort | Running Time

1. ¿Cual es el running time de Quicksort cuando todos los valores son iguales?
2. Utilizar el pseudocódigo de partition para ilustrar la operacion de esa función en el arreglo:
   - A = [13,19,9,5,12,8,7,4,21,2,6,11]
3. ¿Por que es mas utilizado Quicksort y no Heapsort sabiendo que el worst-case running time de Heapsort es mejor?

Running Time | Escenario en el cual todos los valores son iguales

El running time para el algoritmo Quicksort cuando todos los valores a ordenar son iguales es de $(\mathbf{N^2})$, ya que al llamar la función de Quicksort con los parámetros iniciales (A, p=1, r=A.length), va a evaluar si el valor (p) es menor a (r) y como la condición p<r se va a cumplir en la validación, va a ejecutar la función partición(); esta función partición tiene un ciclo for que recorre el array de (p) a (r), en este caso, de 1 a A.lengh, es decir (n) veces. Al terminar de ejecutarse la partición asigna el valor de la variable (q) que va a dar el nuevo parámetro para ejecutar dos veces quicksort [quicksort(A, p, r=q-1) y quicksort(A, q+1, r)]. Cada uno de estos quicksorts va a ejecutar la función partición() con la diferencia que el ciclo for de la primera partición se va a ejecutar cómo mínimo (n-q) veces y el ciclo de la segunda partición se va a ejecutar cómo mínimo (q) veces, ya que en ninguno de los dos ciclos for se va a cumplir la comparación entre valores del arreglo que evitaría que se siga ejecutando el ciclo for. Por lo que la suma de veces que se van a ejecutar los ciclos for de las 3 llamadas a partición(), es de [(n)+(n-q)+(q)] = 2n. Esto quiere decir que cada valor del arreglo se va a evaluar dos veces, y por lo tanto el running time de este algoritmo es de $(\mathbf{N^2})$; siendo este caso

uno de los casos que conforman el escenario worst case de quicksort, por lo que al ser un worst case, el Running Time con su respectiva notación es de $\theta(\mathbf{N^2})$.

$$\textbf{Running Time} = \theta(\mathbf{N^2})$$

Ilustración | Ejecución de partición en array

Los valores en rojo dentro de cada arreglo representa un swap en esas posiciones, y cada número al lado del arreglo representa el valor de J en la iteración del ciclo for.

Arreglo original
A = [13,19,9,5,12,8,7,4,21,2,6,11] 1
A = [13,19,9,5,12,8,7,4,21,2,6,11] 2
A = [9,19,13,5,12,8,7,4,21,2,6,11] 3
A = [9,12,13,5,19,8,7,4,21,2,6,11] 4
A = [9,12,13,5,19,8,7,4,21,2,6,11] 5
A = [9,12,8,5,19,13,7,4,21,2,6,11] 6
A = [9,12,8,7,19,13,5,4,21,2,6,11] 7
A = [9,12,8,7,4,13,5,19,21,2,6,11] 8
A = [9,12,8,7,4,13,5,19,21,2,6,11] 9
A = [9,12,8,7,4,2,5,19,21,13,6,11] 10
A = [9,12,8,7,4,2,6,19,21,13,5,11] 11
A = [9,12,8,7,4,2,6,11,21,13,5,19] Ejecuta la instrucción fuera del ciclo for.

El último arreglo representa el arreglo final, luego de ejecutar partition() y sus instrucciones.

La última instrucción de partition Retorna i+1 (8), y es el swap que se observa en el último arreglo.

¿Quicksort sobre Heapsort?
Las razones por las que Quicksort es más utilizado que heapsort, a pesar de que el worst case de Quicksort sea de orden cuadrático mientras que el de heapsort es logartítmico, se debe a lo siguiente.

La primera razón es que los worst case scenarios de quicksort están bien definidos y son 3, cuando: el arreglo está parcial o completamente ordenado de manera ascendente (1) o descendente (2) o la mayoría sino es que todos los valores del arreglo son iguales (3). La segunda razón se debe a los recursos que utiliza cada al-

goritmo al realizar las operaciones. En el caso de Quicksort, el algoritmo tiene la característica de utilizar menos recursos, ya que el ordenamiento lo hace "in situ", esto quiere decir que realiza los cambios en el mismo arreglo y sólo utiliza memoria para almacenar el valor de los contadores, las llaves de las posiciones del arreglo que va a comparar y un sólo valor del arreglo; el valor de la posición en el arreglo que va a cambiar. Por último, el average running time de quicksort es de la misma complejidad que heaposort con la diferencia que es más eficiente en cuanto a consumo de recursos. Por estas tres razones, quicksort tiene preferencia sobre heapsort.

3 - Quicksort Python

| Adjunto en el repo