Laboratorio #3

Victor Farfan

August 16, 2018

1 Problema #1

Input: array N, array V
Output: array V ordenado descendentemente con los valores de N
def HeapSortAlternate():
while *len(N) > 0* **do**
  v.append(ExtractMax(N));
end
return *V*

El tiempo de ejecución de este algoritmo sigue siendo $O(n * log(n))$ porque recorremos todo el arreglo N para hacer el ExtractMax(), de ahi la "n" en nuestro tiempo de ejecución. Y como con ExtractMax() llamamos a la función Heapify() para borrar el elemento extraído y estamos recorriendo un árbol, tenemos que esta parte del algoritmo corre en $O(log(n))$. Este algoritmo usa más memoria debido a que no ordena el arreglo in-place, pero el tamaño del arreglo N cada vez es menor gracias a ExtractMax().

2 Problema #2

1. $O(n * log(n))$
2.
3. Porque se toma en cuenta el tiempo promedio de ejecución, que es el que más probablemente encontremos en la práctica, en ese caso el tiempo de ejecución es $O(n * log(n))$ y ordena los elementos de manera in-place por lo que utiliza una menor cantidad de memoria.

3 Problema #3

```
class Quiock:
    def __init__(self):
        self.A = [5,10,15,32,55,21,40,2,3,76,89,28,9,7]

    def quicksort(self, p, r):
        if p < r:
            q = self.partition(p, r)
            self.quicksort(p, q)
            self.quicksort(q + 1, r)

    def partition(self, p, r):
        pivot = self.A[p]
        while True:
            while self.A[p] < pivot:
                p += 1
            while self.A[r] > pivot:
                r -= 1
            if p >= r:
                return r
            self.A[p], self.A[r] = self.A[r], self.A[p]
            p += 1
            r -= 1

    def solve(self):
        self.quicksort(0,len(self.A)-1)
        print(self.A)

quick = Quiock()
quick.solve()
```