Laboratorio 2

José Alejandro Guzmán Zamora

9 de agosto de 2018

- Problema de Sorting

  Ordenar la lista de páginas web por su pagerank de mayor a menor. Adicionalmente agregue un contador para ver cuantas veces se iteró el programa y compararlo al input.
  website_page_rank = [3, 39, 61, 91, 57, 22, 75, 89, 9, 90, 63, 78, 28, 73, 20]

  Después de implementar el algoritmo se obtiene lo siguiente

  >python mergesort.py
  $[91, 90, 89, 78, 75, 73, 63, 61, 57, 39, 28, 22, 20, 9, 3]$

  En el caso de Merge-Sort, uno de los pasos más importantes es el loop que une los subarreglos anteriormente partidos, este loop es parte de la función Merge utilizada por el algoritmo principal. Al colocar un contador en el ciclo se obtiene un valor final de 59 para el input de 15 ranks. El valor final puede generar confusión porque se sabe a priori que la complejidad de la función Merge es de $O(n)$. No obstante, hay que tomar en cuenta que en este caso el contador es una variable global y por lo tanto está haciendo el conteo de todo el algoritmo Merge Sort.

  A simple vista, se podría decir que la eficiencia está basada en la complejidad $O(n)$ multiplicado por una constante, en este caso específico:

  $59/15 = 3{,}93$
  Resulta ser aproximádamente $4 * n$

  Sin embargo, si se duplica el tamaño del input, el contador finaliza con un valor de 148:

  $148/30 = 4{,}93$
  Resulta ser aproximádamente $5 * n$

La constante a la que se le multiplica n cambió, lo que indica que no es correcto asignarle una complejidad de $O(n)$. Para el algoritmo de Merge-Sort el análisis de recursión realizado por los autores de la Tercera Edición de *Introduction to Algorithms*, resulta en una complejidad $O(n * lg(n))$ y efectivamente se puede comprobar con la variable de contador en el ejemplo utilizado:

$$15 * lg(15) = 58{,}6$$
$$30 * lg(30) = 147{,}2$$

- Verificación de Heap

Escriba un programa que verifique si el siguiente arreglo es un heap.
possible_heap = [16,14,10,8,7,9,3,2,4,1]

A continuación se muestra el pseudocódigo de la implementación realizada:

Verificación de Heap (arreglo)

```
1       for i from 1 to floor(len(sequence)/2)):
2           if ((2 * i) + 1) <=len(sequence):
3               if (sequence[2 * i]>sequence[i]) or (sequence[(2 * i) + 1] >sequence[i]):
4                   return False
5           else:
6               if (sequence[2 * i] > sequence[i]):
6                   return False
8        return True
```

-

La tarea de verificar si un arreglo es un heap es relativamente sencilla considerando que hay ciertas propiedades que un arreglo debe de cumplir para ser un heap. En este caso, el algoritmo recorre la primera mitad del arreglo para asegurarse que los correspondientes $(i*2)y((i*2)+1)$ no sean mayores al índice que se está comparando. Por consiguiente el contenido del ciclo se ejecuta $n/2$ y en cada iteración se hacen como máximo 2 comparaciones. Considerando lo mencionado el running time de la función es $O(n)$.

- Heapify

Escribir un algoritmo de MAX-HEAPIFY que utilice un constructo de control iterativo en lugar de recursión.

Heapify(A,i):

```
1       While x = true:
2           l = LEFT(i)
3           r = RIGHT(i)
4           if l <= heap-size[A] and A[l] > A[i]:
5               largest = l
6           else largest = i
7           if r <= heap-size[A] and A[r] > A[largest]:
8               largest = r
9           if largest != i:
10              exchange A[i],A[largest]
11           else x = false
```

El cambio principal del algoritmo es que en lugar de utilizar recursión se basa en un loop que verifica el valor de la variable x. En el caso de que i sea el 'largest' el procedimiento debe terminar y la variable x obtiene el valor de false.