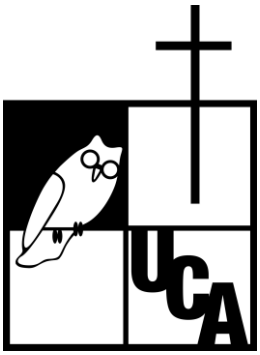
Universidad Centroamericana “José Simeón Cañas”

Análisis de algoritmos



Taller 2

Alumnos:

Alvarenga Hércules, Diego Javier 00027222

Mijango Serrano, Fabio Alberto 00156022

Navarro Domínguez, Daniel de Jesús 00110017

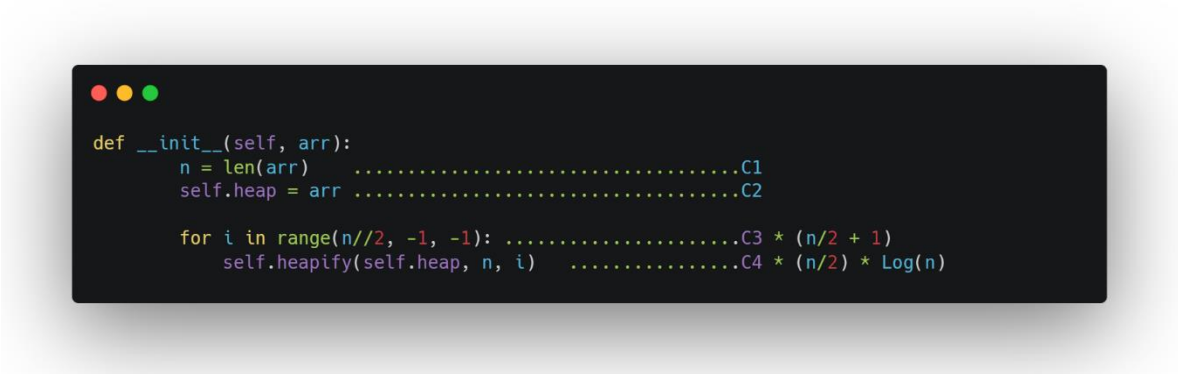
Catedrático:

Emmanuel Araujo

12 de octubre, 2024

Análisis del algoritmo.

Método __init__():



El ciclo hace n/2 iteraciones, y en cada iteración se llama a heapify y este toma **O(log n)** en el peor caso.

$$T(n) = C1 + C2 * \left(\frac{n}{2} + 1\right) + C3 * \left(\frac{n}{2}\right) * \log_2 n$$

$$T(n) = C1 + C2 * \left(\frac{n}{2}\right) + C2 + C3 * \left(\frac{n}{2}\right) * \log_2 n$$

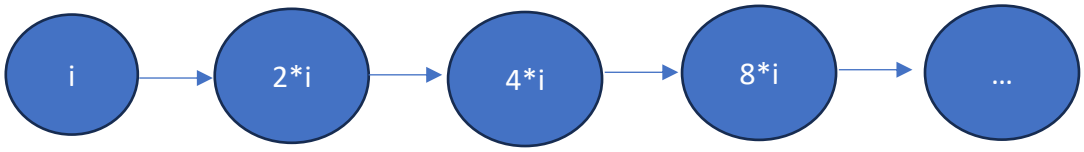
$$\therefore \mathrm{T(n)} = n \log_2 n$$

Función Heapify:

```
def heapify(self, arr, n, i):
        smallest = i        .....C1
        left = 2 * i + 1    .....C2
        right = 2 * i + 2   .....C3

        if left < n and arr[i] > arr[left]: .....C4
            smallest = left .....C5 * Max(0,1)

        if right < n and arr[smallest] > arr[right]:    .....C6
            smallest = right    .....C7 * Max(0,1)

        if smallest != i:   .....C8
            arr[i], arr[smallest] = arr[smallest], arr[i]   .....C9 * Max(0,1)
            self.heapify(arr, n, smallest)
```

Para completar de analizar la complejidad de esta función, hay que analizar la recursividad; Se utilizara una manera distinta al método del árbol.

Para empezar la recursividad el análisis, la cantidad de datos no se ve reducida por cada iteración, entonces al ver la forma que toma $smallest$ (El cual es el parámetro i que se envía en la función) por cada iteración, asumiendo el peor caso, donde al menos un cuerpo de condicional "if" se ejecutó, se ve de esta forma:



Se puede notar que el parámetro i por cada recursión aumenta cuadráticamente, por lo que, hasta llegar al punto donde la recursión termina (Ósea, cuando $2 * i$ es mayores que n):

n = cantidad de datos
k = número de iteración

$$2^k * i > n$$

$$\log_2(2^k * i) > \log_2 n$$

$$k * \log_2 2 + \log_2 i > \log_2 n$$

- $\log_2 i$ es una constante, por lo que puede desaparecer.

$$k > \log_2 n$$

```
def heapify(self, arr, n, i):
        smallest = i        .....C1
        left = 2 * i + 1    .....C2
        right = 2 * i + 2   .....C3

        if left < n and arr[i] > arr[left]: .....C4
            smallest = left .....C5 * Max(0,1)

        if right < n and arr[smallest] > arr[right]:    .....C6
            smallest = right    .....C7 * Max(0,1)

        if smallest != i:   .....C8
            arr[i], arr[smallest] = arr[smallest], arr[i]   .....C9 * Max(0,1)
            self.heapify(arr, n, smallest)  .....C10 * Log(n)
```

Entonces:

$$T(n) = C1 + C2 + C3 + C4 + C5 * Max(0,1) + C6 + C7 * Max(0,1) + C8 + C9 * Max(0,1) + C10 * Log(n)$$

$$C = C1 + C2 + C3 + C4 + C5 + C6 + C7 + C8 + C9$$

$$T(n) = C10 * Log_2(n) + C$$

∴ El tiempo de ejecución de la función recursiva es $\log_2 n$

Método insert:

$$T(n) = C1 + C2 + C3 * \log_2 n + C4 * \log_2 n + C5 * \log_2 n$$

$$T(n) = C1 + C2 + \log_2 n * (C3 + C4 + C5)$$

$$T(n) = C1 + C2 + C * \log_2 n$$

$$T(n) = \log_2 n$$

Esto significa que el tiempo de ejecución del método de inserción aumenta logarítmicamente con el número de elementos en el heap.

Función Heap_Sort:

$$T(n) = C1 + C2 + C3 * (n-1) + C4 * (n-2) \\ +C5 * (n-2) * \log_2 n$$

$$T(n) = C1 + C2 + C3 * n - C3 + C4 * n - 2 * C4 \\ +C5 * n * \log_2 n - 2 * C5 * \log_2 n$$

$$T(n) = C5 * n \log_2 n - 2 * C5 * \log_2 n + C3 * n + C4 * n \\ +C1 + C2 - C3 - 2 * C4$$

$$\therefore \mathrm{T(n)} = n \log_2 n$$

La complejidad temporal del método de ordenación utilizando Heap Sort es **O(n $\log_2 n$)**. Este comportamiento se debe a que estamos realizando una serie de operaciones de heapify para cada uno de los elementos en el ciclo de ordenación.

Archivo Main:

```python
import csv

from Employee import Employee
from Heap import Heap

entries = []                                                    C1
with open('entries.csv', 'r') as f:                             C2
    reader = csv.DictReader(f, fieldnames=['name', 'salary'])   C3
    for row in reader:                                          C4 * n
        entries.append(Employee(row['name'], int(row['salary']))) C5 * (n – 1)

h = Heap(entries)                                               C6
for entry in h.sort():                                          C7 * n log( n)
    print(entry)                                                C8 * ( n - 1)
```

$$T(n) = C1 + C2 + C3 + C4 * n + C5 * (n - 1) + C6 + C7 * n\log_2 n + C8 * (n - 1)$$

$$T(n) = C1 + C2 + C3 + C4 * n + C5 * n - C5 + C6 + C7 * n\log_2 n + C8 * n - C8$$

$$C = C1 + C2 + C3 + C4 - C5 + C6 - C8$$

$$T(n) = C1 + C2 + C3 + C4 * n + C5 * n - C5 + C6 + C7 * n\log_2 n + C8 * n - C8$$

$$T(n) = C4 * n + C5 * n + C7 * n\log_2 n + C8 * n + C$$

$$T(n) = C7 * n\log_2 n + C4 * n + C5 * n + C8 * n + C$$

$$\therefore T(n) = n\log_2 n$$

El desarrollo de este proyecto represento un reto interesante para la optimización de recursos y conseguir un algoritmo eficiente. Con el problema de de la empresa Salem y su creciente equipo fue necesario encontrar una solución junto al algoritmo HEAP que sea lo mas rápido posible y ahorre la máxima cantidad de memoria posible, y no solamente cumpla los requisitos mínimos funcionales.

El diseño de este programa el cual integra HEAP como método de ordenamiento de datos, buscar ayudar al equipo con un código simple pero eficiente, y debido a su ejecución en tiempos adecuados, proporciona una herramienta útil para la empresa que servirá incluso a planes futuros.

En resumen, luego del análisis, se llego a la conclusión que el programa resolvió el problema de manera exitosa y eficiente, siendo así con una complejidad temporal de O(n log n), lo que asegura un rendimiento optimo incluso al manejar grandes volúmenes de datos, contribuyendo al éxito y sostenibilidad a largo plazo de la empresa.