TEORÍA DE ALGORITMOS
(75.29) CURSO BUCHWALD - GENENDER

Trabajo Práctico 0
Lineamientos sobre informes

30 de marzo de 2024

Zielonka Axel
110310

Petean Marina
padron

Romano Nicolas
110830

Ejemplo de Resolución

1. Algoritmo para minimizar la suma ponderada

En este trabajo de ejemplo realizaremos el análisis bla bla bla

1.1. titulo

A continuación se muestra el código de solución iterativa del problema.

```python
def maximo(datos):
    max_pos = 0
    for i in range(1, len(datos)):
        if datos[i] > datos[max_pos]:
            max_pos = i
    return datos[max_pos]
```

La complejidad del algoritmo propuesto para encontrar el máximo es $\mathcal{O}(n)$, debido a que para cada elemento del arreglo se realizan operaciones $\mathcal{O}(1)$.

1.2. Algoritmo por División y Conquista

A continuación, mostramos la implementación de un algoritmo que encuentra el máximo de un arreglo por División y Conquista. Es decir, busca el máximo que corresponde al subarreglo izquierdo, lo mismo para el derecho y se queda con el máximo entre ambos *sub máximos*.

```python
def maximo(datos):
    if len(datos) == 1:
        return 0

    izq = maximo(datos[:len(datos)//2])
    der = maximo(datos[len(datos)//2:])
    return izq if izq > der else der
```

La ecuación de recurrencia que corresponde a este algoritmo es:

$$\mathcal{T}(n) = 2\mathcal{T}\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

Esto es porque tenemos 2 llamados recursivos, cada lado con la mitad del problema, y al partir hacemos una slice, lo cual en Python realiza una copia, por lo cual demora tiempo lineal en aplicarse en cada caso.

Aplicando el teorema maestro, la complejidad nos queda en $\mathcal{O}(n \log n)$. En este caso, nos quedó peor complejidad que en el caso iterativo pura y exclusivamente por abusar del lenguaje de programación sin tomar en cuenta el tiempo que consume hacer un slice. Dejando de hacer esto, podemos mostrar la siguiente versión del algoritmo:

```python
def maximo(datos):
    return maximo_dyc(datos, 0, len(datos) - 1)

def maximo_dyc(datos, inicio, fin):
    if inicio == fin:
        return datos[inicio]

    medio = (inicio + fin) / 2
    izq = maximo_dyc(datos, inicio, medio)
    der = maximo_dyc(datos, medio + 1, fin)
    return izq if izq > der else der
```

En este caso, la ecuación de recurrencia es:

$$\mathcal{T}(n) = 2\mathcal{T}\left(\frac{n}{2}\right) + \mathcal{O}(1)$$

Aplicando el teorema maestro, nos queda que la complejidad es $\mathcal{O}(n)$.

Es importante notar que en otros lenguajes de programación esto podría no ser necesario (por ejemplo, Go). En algunos lenguajes se puede operar usando Slices que consuman $\mathcal{O}(1)$ de tiempo (a cambio de utilizar la misma memoria que el arreglo original), o bien usando aritmética de punteros como puede ser el caso de C. Independientemente del caso, es importante notar que el algoritmo de división y conquista es lógicamente igual (o extremadamente similar), pero tenemos que considerar cuestiones de implementación del lenguaje elegido a la hora de definir las complejidades.

2. Mediciones

Se realizaron medicions en base a crear arreglos de diferentes largos, yendo de 100 en 100 elementos, donde los elementos en cada caso fueron generados por los valores pseudoaleatorios del lenguaje (el módulo `random`).

Como se puede apreciar, ambos algoritmos tienen una tendencia efectivamente lineal en función del tamaño de la entrada, si bien el algoritmo iterativo es más veloz en cuestión de constantes.

3. Conclusiones

Acá irían las conclusiones de todo nuestro trabajo :)