Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 0
Lineamientos sobre informes

30 de marzo de 2025

Pablo	Ezequiel Matias	Abel Tomas
Gaston Choconi	Gacitua	Modesti Funes
106388	112234	111932

1. Consideraciones

Este documento está pensado para que sirva de guía, e incluso template a la hora de realizar un informe para la materia (por no decir, cualquier documento formal en la facultad). La idea es utilizar LaTeX, pura y exclusivamente para que tenga un formato acorde. Es decir, no hecho en un simple procesador de texto (como pudiese ser Word).

Aquí sólo se mostraran algunas de las cosas que se pueden hacer con LaTeX. Si se quiere hacer algo específico, o de otra forma, abunda la documentación (y artículos de StackOverflow) al respecto.

Definimos ciertos lineamientos técnicos a considerar para los informes:

- En el informe debe estar incluido el código más importante (es decir, los algoritmos, no un main) cuando corresponda.
- Debe estar explicado cómo se hacen los sets de prueba. Inclusive, si notan alguna falencia con esta generación que pueda afectar de alguna forma el análisis del resultado, se espera que lo enuncien (hay trabajos donde esto no sucederá, pero hay trabajos donde sí, especialmente cuando vemos aproximaciones).
- Los gráficos deben estar bien generados: deben tener título, los ejes deben tener tanto qué refieren como qué unidad de medida usan (en caso de tenerla). En caso de tener gráficos superpuestos, debe quedar claro qué es cada gráfico.
- **Importante**: en caso de tener que realizar una reentrega (y por ende, tener que cambiar cosas del informe), salvo que haya que rehacer una enorme cantidad del mismo, no modificar el informe anterior sino agregar un anexo al final indicando las correcciones realizadas en la nueva entrega, para agilizar las correcciones.

A su vez, dejamos algunas breves líneas sobre cuestiones de redacción:

- Evitar errores ortográficos y gramaticales.
- El lenguaje debe ser técnico. No tiene que por eso ser un texto aburrido, pero evitar frases como "*la cosa que más me complicó fue...*", "*creo que...*".
- Usar plural de modestia, incluso si el trabajo fuera realizado por una única persona.

Ejemplo de Resolución

2. Algoritmo para encontrar el máximo

En este trabajo de ejemplo realizaremos el análisis teórico y empírico de diferentes algoritmos que sirven para resolver el mismo problema: obtener el valor del máximo elemento de un arreglo desordenado de n elementos.

2.1. Algoritmo Iterativo

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

```
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 $\mathcal{O}(1)$.

2.2. 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*.

```
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:

```
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:

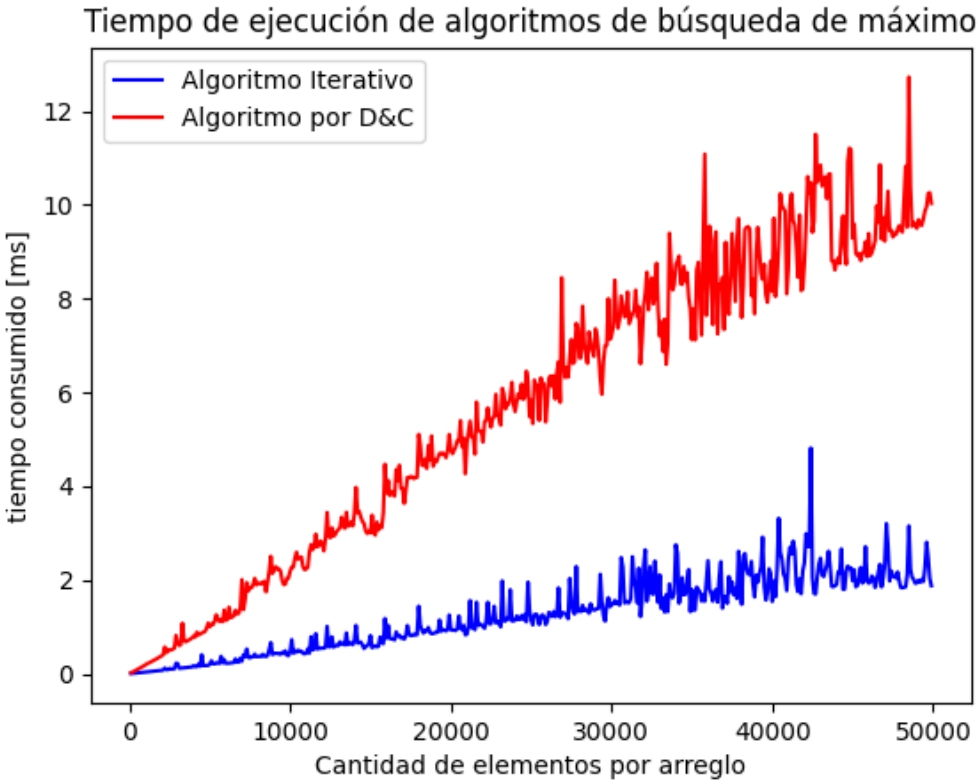
$$\mathcal{T}(n) = 2\mathcal{T}\left(\frac{n}{2}\right) + \mathcal{O}(1)$$

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.

3. 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.

4. Conclusiones

Acá irían las conclusiones de todo nuestro trabajo :)