Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2

28 de abril de 2024

Petean Marina
110564

Romano Nicolas
110830

Zielonka Axel
110310

1. Programación Dinámica para el Reino de la Tierra

1.1. Análisis del problema

El problema pide ...

1.2. Ecuacion de recurrencia y algoritmo propuesto

Aplicamos una regla sencilla que permite obtener un óptimo local para el estado actual, y la aplicamos de forma iterativa para llegar a un óptimo general.

Esto lo logramos ordenando el arreglo por $a = \frac{b}{t}$ de mayor a menor, lo que nos permite al recorrer el mismo obtener en cada iteración un óptimo local (es decir, el elemento que optimiza la suma ponderada en ese momento). Al recorrer de manera ordenada el arreglo de batallas y calcular la sumatoria utilizando el óptimo local, que es el elemento actual según nuestra regla sencilla, conseguimos llegar al óptimo global (la suma ponderada mínima).

```
def minimizar_suma(batallas):

    batallas_indices = crear_arreglo_con_indice(batallas)
    batallas_ordenadas = sorted(batallas_indices, key=lambda batalla: -(batalla
    [0][1] / batalla[0][0]))
    f = []
    for i in range(len(batallas_ordenadas)):
        if i == 0:
            f.append(batallas_ordenadas[i][0][0])
        else:
            f.append(f[i - 1] + batallas_ordenadas[i][0][0])

    suma_ponderada = 0
    for i in range(len(batallas_ordenadas)):
        suma_ponderada += f[i] * batallas_ordenadas[i][0][1]

    return suma_ponderada, batallas_ordenadas

def crear_arreglo_con_indice(arr):
    arreglo_con_indice = []
    for indice, valor in enumerate(arr):
        arreglo_con_indice.append([valor, indice])
    return arreglo_con_indice
```

1.3. Por qué es Programación Dinámica?

1.4. Cómo afecta la variación de las rágafas y de la función de defensa a los algoritmo?

La variabilidad de los valores t_i y b_i no afectan la optimalidad del algoritmo, pues estamos ordenando por una relación entre el peso y el tiempo que siempre resulta en darnos el orden óptimo. En términos generales lo que buscamos es tener ordenado de mayor a menor peso, y de menor a mayor tiempo de manera simultanea, es por esto que al realizar b/t y ordenarlo de mayor a menor, funciona en todos los casos, cosa que no pasaria ordenando primero por peso y luego por tiempo, en lugar de por la relación entre ambos. Por ejemplo, si la duración de una batalla (t) es extremadamente grande, teniendo un peso grande en promedio, y b/a nos da un numero muy pequeño en comparacion a las demás, esta batalla estará en las ultimas del orden, pues así ese tiempo muy grande aparecerá en menos términos. Si lo hicieramos ordenando por tiempos de menor a mayor y luego por pesos de menor a mayor, el peso nos estaría importando más a la hora de ordenar, cuando en realidad ese tiempo tan grande hará que la suma ponderada se vuelva mucho más grande. Pasaría lo mismo si ordenaramos primero por peso, y luego por tiempo y tuvieramos un tiempo pequeño en promedio y un peso extremadamente chico.

Por lo tanto, nuestro algoritmo es óptimo sin importar como varien t_i y b_i. Lo único en lo que esta variabilidad influirá es en que posición estará la $batalla_i$ en el orden.

2. Análisis de Complejidad

2.1. Algoritmo planteado

A continuación se muestra el código de solución del problema.

2.2. Complejidad temporal

La complejidad de este algoritmo se puede calcular realizando la suma de los sub-procesos que suceden dentro de él. Lo primero que sucede es que se crea un nuevo arreglo *batallas indices* que tiene como elementos a los mismos que *batallas*, agregando el índice de cada batalla a cada tupla. Crear un arreglo, y hacerle n asignaciones tiene complejidad de $O(n)$ siendo n la cantidad de batallas (el único propósito es poder mostrar el número de batalla al mostrar el orden óptimo).

Luego se crea un nuevo arreglo, *batallas ordenadas* resultado de ordenar *batallas* de mayor a menor según $\frac{b_i}{t_i}$, utilizando la función *sorted()* de Python. Este algoritmo tiene una complejidad de $O(nlog(n))$. A continuación se realiza la asignación de la felicidad (F_i) de cada batalla, y como es crear un arreglo, recorrerlo completo y asignarle n variables esto tiene una complejidad $O(n)$

Por último, se realiza el cálculo de la suma ponderada, que es la sumatoria de los productos de las felicidades de cada batalla sus respectivos pesos. Estamos recorriendo el arreglo y haciendo operaciones constantes, por lo tanto eealizar n este proceso de suma tiene complejidad $O(n)$

Entonces, como estos procesos no son en simultáneo (no hay estructuras anidadas una dentro de otra), sus complejidades se suman y prevalece la mayor: $O(n) + O(nlog(n)) + O(n) + O(n) = O(nlog(n))$

Por lo tanto, la complejidad del algoritmo propuesto es de $O(nlog(n))$ siendo n la cantidad de batallas.

2.2.1. Análisis del efecto en la complejidad de la variabilidad de los valores t_i y b_i

Puesto que la complejidad no depende de las variables t_i y b_i, si no de n, siendo n la cantidad de batallas, como varíen estas no altera en nada la complejidad.

2.2.2. Mediciones y gráficos

Tiempos de ejecución con respecto a cantidad de batallas
Podemos observar que el grafico sigue la forma de la función xlog(x), lo que corrobora la complejidad teórica indicada.

Tiempos de ejecución a igual N
Con este gráfico analizamos 5 posibles variaciones que pueden tomar t y b de las batallas y como varía el tiempo de ejecución del programa.

2.3. Complejidad espacial????

3. Testing

Para poder ver si nuestro algoritmo funcionaba correctamente, era necesario proveer de ejemplos de sets en donde variaban los tamaños, así como también los valores tanto del peso como del tiempo de cada batalla. En el directorio ***archivos*** del repositorio de Github se pueden encontrar además

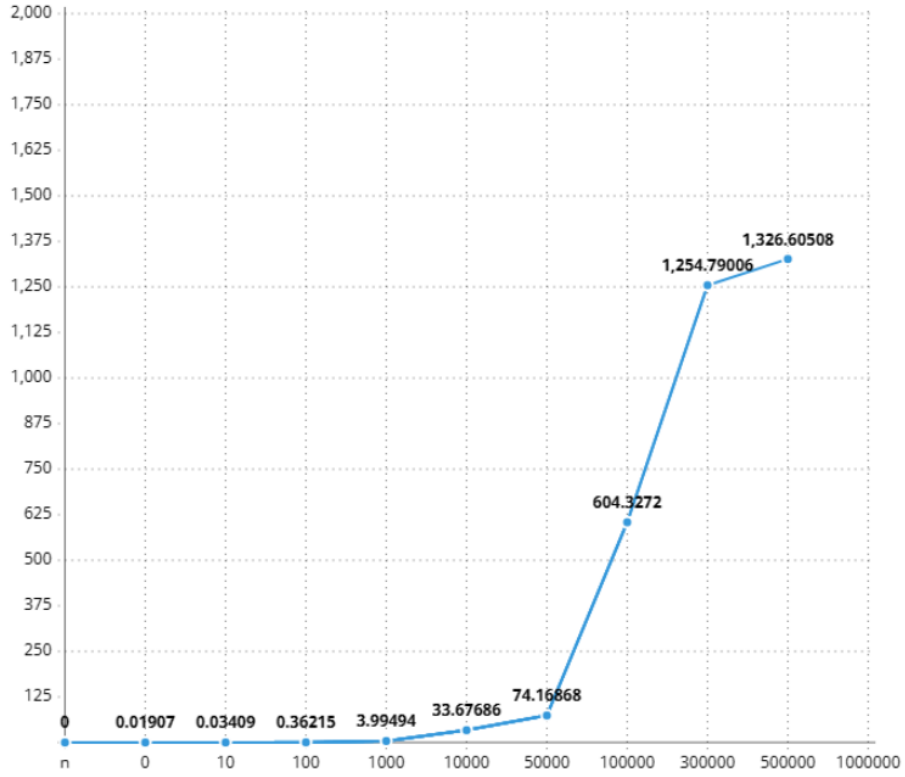

Figura 1: Gráfico de tiempos de ejecución (milisegundos) respecto a cantidad de batallas (n)

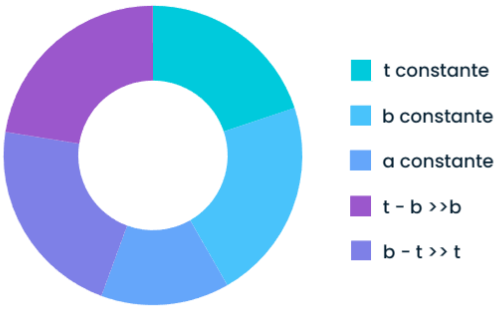

Figura 2: Gráfico circular de modos de variación

de los archivos provistos por la cátedra algunos archivos con distintos sets de batallas, que varían en tamaño y en los valores de los datos.

Por otro lado, en el directorio ***test*** del repositorio, está el archivo que se encarga de leer los ejemplos presentados por la cátedra y chequear que el resultado sea el esperado.

Para nuestro propio set de datos, los resultados esperados son:
Para n = ...

4. Conclusiones

Hemos desarrollado y análizado nuestro algoritmo por programación dinámica propuesto para obtener el orden óptimo en el que defenderse de la nación del Fuego para maximizar la cantidad de soldados matados. Demostramos que ... a los tiempos de ejecución. Hemos estudiado ... Por lo tanto, nuestro algoritmo de programación dinámica nos permite determinar siempre en qué momentos debemos realizar estos ataques de fisuras para eliminar a tantos enemigos en total como sea posible.