Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 1
Algoritmos Greedy en la Nación del Fuego

8-04-2024

Álvarez Julián 110825

Ascencio Felipe Santino 110675

Olivera Nicolás 110797

Introducción

1. Análisis del enunciado

En este trabajo se pide "ayudar al señor del fuego" en cuanto al orden en el que debe realizar sus conquistas.

Cada batalla de todas las que tiene que librar tiene un peso "b" y un tiempo "t" (de los cuales ninguno de los 2 puede ser menor o igual a "0").

El orden que se debe obtener para realizar estas batallas, es el que minimiza la sumatoria:

$$\sum_{i=1}^{n} b_i F_i.$$

Figura 1: Sumatoria del enunciado.

Sabiendo que el valor de "Fi" con "i=1" es igual a "t1", y luego se va sumando los siguientes tiempos tal que, por ejemplo, "Fj" con "j=2" es igual a "t1 + t2".

2. Análisis problema

El problema consiste en determinar el orden óptimo de las batallas de manera que se minimice la suma ponderada de los tiempos de finalización, considerando tanto el tiempo que lleva cada batalla como su importancia.

Se tiene información sobre el tiempo que necesita el ejército para ganar cada batalla (tiempo "ti") y la importancia de cada batalla (peso "bi"). También se sabe que el momento en que se termina una batalla i (Fi) depende del momento en que se termina la batalla anterior más el tiempo que lleva la batalla i.

En este caso nos piden usar un algoritmo greedy, el cual busca tomar la mejor decisión en cada paso (optimo local) para minimizar la suma ponderada de los tiempos de finalización de las batallas (optimo global). El algoritmo consiste en lo siguiente:

En este informe se pasa a detallar el paso a paso seguido para encontrar el "Algoritmo Greedy" que resuelva el enunciado de forma "óptima".

3. Pequeña consideración

La explicación de la distribución de los archivos de "GITHUB", y la explicación de ejecución de los programas, se encuentra en el "README" del repositorio.

Desarrollo

4. Primeros pasos

Para obtener el "Algoritmo Greedy" óptimo que resuelve el problema, se pensó en unas primeras soluciones posibles, algunas de estas fueron:

- Ordenar por "t" creciente.
- Ordenar por "t" decreciente.
- Ordenar por "b" creciente.
- Ordenar por "b" decreciente.

Si bien a estos 4 algoritmos se les encontró contraejemplos, se encontró que la idea de identificar una característica que nos permita ordenar, realizar dicho ordenamiento, y luego ir realizando la sumatoria con el orden obtenido, era la indicada para obtener el "Algoritmo Greedy" solicitado para resolver el problema.

5. Detección del algoritmo óptimo

Luego de varios intentos, pruebas con distintos cálculos y órdenes, se detectó que lo mejor era realizar la división entre el peso "b" y el tiempo "t" que involucraban cada batalla (utilizando el concepto de "razón de importancia"), y ordenar de mayor a menor.

La razón de importancia representa cuánto impacto genera una batalla por cada unidad de tiempo que lleva completarla. Al ordenar las batallas de acuerdo a esta razón, estamos priorizando aquellas que generan más impacto en menos tiempo. Esto es importante porque el Señor del Fuego desea minimizar el tiempo total de finalización de todas las batallas mientras maximiza su importancia.

Así, lo que se hace es maximizar la importancia por unidad de tiempo, al ordenar, se vuelve eficiente el algoritmo en la toma de decisiones y, además, optimiza recursos ya que, al priorizar las batallas con mayor razón de importancia, se está optimizando el uso de los recursos disponibles, como el tiempo y el esfuerzo del ejército del Maestro Fuego.

Entonces, con esto claro, ya se puede desarrollar el concepto de "Algoritmo Greedy" necesario, para resolver el problema.

Importante: En el programa "ejemplonombres.py" están tanto las 4 formas de ordenamiento nombradas en el subtítulo "Primeros pasos", como la propuesta como óptima, y la de su orden inverso, a fin de ver como en el mismo ejemplo, muestran resultados de sumatoria distinto, y como el planteado como óptimo, es el algoritmo que consigue el menor valor de sumatoria, tal y como lo pide el enunciado.

6. Análisis del algoritmo

Se pasa a explayar todo el algoritmo, paso a paso, para luego analizar si efectivamente cumple con las condiciones de un "Algoritmo Greedy".

Primero: Se procede a leer, y conservar en una tupla, todos los pares "peso" y "tiempo" de cada batalla.

Segundo: Se realiza la división "bi/ti" de todos los elementos, y se los pasa a ordenar de mayor a menor.

Tercero: Para cada batalla en el orden determinado en el paso 2 se hace lo siguiente:

a) Calcular el tiempo de finalización de la batalla.

b) Sumar el producto de la importancia de la batalla por su tiempo de finalización a la suma ponderada

Cuarto: Se imprime por pantalla tanto el resultado de la sumatoria como el orden elegido para la misma.

Como se observa, este algoritmo sigue las normas para ser "Greedy", ya que parte de la lectura de la información, la ordena mediante a una regla simple, y luego simplemente va realizando la operación que le pedimos (en este caso la sumatoria) con cada batalla según el orden que le hayamos asignado por el resultado de su correspondiente división.

7. Análisis de optimalidad

Ordenar por razón de importancia (bi/ti) proporciona una posible solución, pero se necesitaría probar que esta solución es igual de buena que cualquier otra hipotética solución óptima. Para esto se pasa a demostrar por el método de inversiones la optimalidad del algoritmo propuesto. Es decir, se pueden realizar inversiones sobre la solución óptima para demostrar la equivalencia con la solución propuesta.

Decimos que se tiene una inversión si dentro del orden de batallas tenemos dos cambiadas de lugar, tal que la primera batalla que aparece (i) tiene con respecto a la segunda (j) "bi/ti"menor a "bj/tj", ya que si la razón de importancia (i) fuera mayor estaría ordenado de la misma manera que nuestro algoritmo (descendentemente).

Como el algoritmo greedy ordena las batallas en función de su razón de importancia, siempre se cumple que si:

$\frac{b_i}{t_i} > \frac{b_j}{t_j} \Rightarrow b_i \cdot t_j > b_j \cdot t_i$

Entonces la solución greedy planteada, la batalla (i) aparece antes que la (j), por lo que la suma ponderada será:

$$\Sigma_{n=1}^{n=j} F_n * b_n$$

Figura 2: Sumatoria para la justificación.

Suma ponderada = ... + Fi * bi + Fj * bj + ...

Teniendo en cuenta que "y" es la duración de las batallas anteriores a la posición "i", entonces, la parte anterior se puede escribir como:

bi * (ti+y) + bj * (ti+y+tj) (1)

Mientras que en la otra solución optima hipotética con inversión, cambian de lugar la (i) con la (j) por ende:

bj * (tj+y) + bi * (tj+y+ti) (2)

Si desarrollamos (1) y (2) queda:

(bi * ti + bj * tj + bi * y + bj * y) + bj * ti (1)

(bi * ti + bj * tj + bi * y + bj * y) + bi * tj (2)

Lo encerrado con paréntesis indica que esas partes son iguales. El (2) indicaría el orden original de la solución optima hipotética que tiene esa inversión. Mientras que en (1) realizamos el intercambio para que quede igual a nuestra solución greedy propuesta.

Sabiendo desde un principio que "bj * ti"menor a "bi * tj", podemos darnos cuenta que la suma ponderada en (2) es mayor que la de (1) (al intercambiar (i), (j) en la solución optima hipotética), y como nuestro objetivo es minimizar la suma ponderada, llegamos a una contradicción. Intercambiamos batallas invertibles (que son contiguas) y al invertirlas no aumentamos la suma ponderada, (podría mejorar). En este caso llegamos a una contradicción ya que la solución "optima" hipotética resultó no serlo.

Una pequeña aclaración, como se ve en la próxima hoja, el algoritmo consigue y ordena de forma distinta los valores, pero obteniendo el mismo resultado.

Sabemos que hacer "bi/ti."ordenando de mayor a menor nos dará lo mismo que hacer "ti/bi."ordenando de menor a mayor, como utilizamos la función "sorted()"que ordena de menor a mayor, utilizamos la división "ti/bi", pero el resultado es el mismo y no afecta en nada a ningún análisis previo.

8. Código del algoritmo óptimo

```python
# PRE: El string recibido por parametro debe ser valido.
# POST: De ser posible, la funcion devuelve el vector "batallas" cargado con los datos del archivo.
def guerra(archivo):
    try:
        nombre_archivo = archivo + ".txt"
        with open(nombre_archivo) as archivo:
            lineas = archivo.readlines()
            batallas = []
            for linea in lineas: #O(n)
                datos = linea.strip().split(",")
                if datos[0].isnumeric() and datos[1].isnumeric():
                    tiempo = float(datos[0].strip())
                    peso = float(datos[1].strip())
                    batallas.append((tiempo, peso))
        return batallas
    except IOError:
        print("Error al abrir el archivo")
        return None

#PRE: El vector "batallas" debe estar inicializado correctamente.
#POST: Resuelve la sumatoria e imprime por pantalla tanto el resultado como el orden de las batallas.
def resultado(batallas):
    batallas_ordenadas = sorted(batallas, key=lambda x: x[0] / x[1]) #O(n Log(n))
    finalizacion_tot = 0
    suma_ponderada = 0

    print("\nEl orden de las batallas para minimizar la suma ponderada de los tiempos de finalizaci n es:")
    for batalla in batallas_ordenadas: #O(n)
        tiempo, peso = batalla
        finalizacion_tot += tiempo
        print(tiempo, peso)
        suma_ponderada += peso * finalizacion_tot
    print("La sumatoria total es:", suma_ponderada)

def main():
    archivo = input("ingrese nombre del archivo\n")
    batallas = guerra(archivo) #O(n)
    if batallas != None:
        resultado(batallas)

main()
```

9. Código del algoritmo de los gráficos

```python
import time
import math
import matplotlib.pyplot as plt

def guerra(archivo):
    try:
        nombre_archivo = archivo + ".txt"
        with open(nombre_archivo) as archivo:
            lineas = archivo.readlines()
            batallas = []
            for linea in lineas: #O(n)
                datos = linea.strip().split(",")
                if datos[0].isnumeric() and datos[1].isnumeric():
                    tiempo = float(datos[0].strip())
                    peso = float(datos[1].strip())
                    batallas.append((tiempo, peso))
        return batallas
    except IOError:
        print("Error al abrir el archivo")
        return None

def ordenar_batallas(batallas):
    batallas_ordenadas = sorted(batallas, key=lambda x: (x[0]/x[1]))
    tiempo_total = 0
    suma_ponderada = 0
    for batalla in batallas_ordenadas:
        tiempo_total += batalla[1]  # Suma la duraci n de la batalla actual al tiempo total
        suma_ponderada += tiempo_total * batalla[0]

    return batallas_ordenadas, suma_ponderada


# Realizar mediciones de tiempos
archivo = input("ingrese nombre del archivo\n")
batallas = guerra(archivo) #O(n)
n = len(batallas)
tiempos = []
for i in range(1, n+1):
    inicio = time.time()
    orden, suma_ponderada = ordenar_batallas(batallas[:i])
    fin = time.time()
    tiempos.append(fin - inicio)

# Graficar los tiempos de ejecuci n
escala = 10**7
tiempos_escalados = [t * escala for t in tiempos]

plt.figure(figsize=(12, 8))
plt.plot(range(1, n+1), tiempos_escalados, marker='o')
plt.plot(range(1, n+1), [i * math.log(i) for i in range(1, n+1)], 'r--', label='O(n log n)')
plt.xlabel('N mero de batallas')
plt.ylabel('Tiempo de ejecuci n (s)')
plt.title('Tiempo de ejecuci n del algoritmo')
plt.grid(True)
plt.show()
```

10. Código del algoritmo de los nombres

```python
import os

def guerra():
    try:
        with open("batallas.txt") as archivo:
            lineas = archivo.readlines()
            batallas = []
            for linea in lineas: #O(n)
                datos = linea.strip().split(",")
                terreno = datos[0].strip()
                peso = float(datos[1].strip())
                duracion = float(datos[2].strip())
                batallas.append((terreno, peso, duracion))
        return batallas
    except IOError:
        print("Error al abrir el archivo")
        return None

def orden_1(batallas):
    batallas_ordenadas = sorted(batallas, key=lambda x: x[1] / x[2])
    finalizacion_tot = 0
    for i, batalla in enumerate(batallas_ordenadas):
        terreno, peso, duracion = batalla
        finalizacion_tot += duracion
        batallas_ordenadas[i] = (terreno, peso, finalizacion_tot)
    return batallas_ordenadas

def orden_2(batallas):
    batallas_ordenadas = sorted(batallas, key=lambda x: x[2], reverse= True)
    finalizacion_tot = 0
    for i, batalla in enumerate(batallas_ordenadas):
        terreno, peso, duracion = batalla
        finalizacion_tot += duracion
        batallas_ordenadas[i] = (terreno, peso, finalizacion_tot)
    return batallas_ordenadas

def orden_2_reves(batallas):
    batallas_ordenadas = sorted(batallas, key=lambda x: x[2])
    finalizacion_tot = 0
    for i, batalla in enumerate(batallas_ordenadas):
        terreno, peso, duracion = batalla
        finalizacion_tot += duracion
        batallas_ordenadas[i] = (terreno, peso, finalizacion_tot)
    return batallas_ordenadas

def orden_3(batallas):
    batallas_ordenadas = sorted(batallas, key=lambda x: x[1])
    finalizacion_tot = 0
    for i, batalla in enumerate(batallas_ordenadas):
        terreno, peso, duracion = batalla
        finalizacion_tot += duracion
        batallas_ordenadas[i] = (terreno, peso, finalizacion_tot)
    return batallas_ordenadas

def orden_3_reves(batallas):
    batallas_ordenadas = sorted(batallas, key=lambda x: x[1], reverse= True)
    finalizacion_tot = 0
    for i, batalla in enumerate(batallas_ordenadas):
        terreno, peso, duracion = batalla
        finalizacion_tot += duracion
        batallas_ordenadas[i] = (terreno, peso, finalizacion_tot)
    return batallas_ordenadas

def orden_4(batallas):
    batallas_ordenadas = sorted(batallas, key=lambda x: x[2] / x[1]) #O(n Log(n))
    finalizacion_tot = 0
    for i, batalla in enumerate(batallas_ordenadas): #O(n)
        terreno, peso, duracion = batalla
```

```
69          finalizacion_tot += duracion
70          batallas_ordenadas[i] = (terreno, peso, finalizacion_tot)
71      return batallas_ordenadas
72
73  def resultado(batallas_ordenadas):
74      sumatoria = 0
75      print("\nEl orden de las batallas para minimizar la suma ponderada de los
        tiempos de finalizaci n es:")
76      for batalla in batallas_ordenadas:
77          terreno , peso, duracion = batalla
78          sumatoria += peso * duracion
79          print(terreno)
80      print("La sumatoria total es:", sumatoria)
81
82  def main():
83      batallas = guerra() #O(n)
84      if batallas != None:
85          print("\ntest 1")
86          resultado(orden_1(batallas)) #O(n Log(n))
87          print("\ntest 2")
88          resultado(orden_2(batallas))
89          print("\ntest 2_reves")
90          resultado(orden_2_reves(batallas))
91          print("\ntest 3")
92          resultado(orden_3(batallas))
93          print("\ntest 3_reves")
94          resultado(orden_3_reves(batallas))
95          print("\ntest 4")
96          resultado(orden_4(batallas))
97
98  main()
```

11. Análisis de complejidad computacional

Primero que nada, para evaluar la complejidad computacional se va a utilizar la notación "BIG O".

Como se puede observar en el "algoritmo óptimo", vemos que principalmente hay 2 funciones con complejidades computacionales a analizar (se obvia el "main" porque simplemente lo que hace es llamar a las 2 siguientes funciones a analizar):

- **guerra()**: La función "guerra", lo que hace es mediante al nombre recibido por parámetro, buscar si existe el archivo a utilizar para el problema, y de ser así, lee toda la información que contiene (por eso es importante que el archivo tenga la estructura correcta) y la va almacenando en una tupla "batallas" para luego devolver dicha tupla. Todo este trabajo tiene una complejidad de "O(n)", siendo "n" las "n líneas" que tenga que leer del archivo.
- **resultado()**: La función "resultado" recibe la tupla "batallas", y tiene principalmente 2 iteraciones a analizar, la primera es que mediante la condición de la "razón de importancia", ordena la tupla con la función "sorted", de la cual esperamos una complejidad computacional de "O(n.log(n))" (igual a mergesort o quicksort), siendo "n" las "n batallas" que tenga la tupla. Luego, mediante a un "for" se itera todo la tupla de "batallas" para, por un lado, imprimir el orden de las batallas elegido, y por el otro, ir realizando la sumatoria pedida, para finalmente imprimir también el resultado de la sumatoria. Este procedimiento tiene una complejidad de "O(n)", siendo "n" las "n batallas" que tenga la tupla. Entonces, al final del análisis de esta función nos encontramos con una complejidad computacional de "O(n) + O(n.log(n))", que sabemos que tendiendo al infinito, la parte relevante y que nos va a dictar la complejidad computacional es "O(n.log(n))".

Luego de este análisis, podemos definir que la complejidad computacional de nuestro algoritmo óptimo es "O(n.log(n))".

Aclaración importante: En el algoritmo funcional, se tuvo que realizar una mínima modificación de planteo del algoritmo, que no modifica para nada el análisis previo.

La función "sorted" ordena de menor a mayor, pero a nosotros al hacer la "razón de importancia" de "bi/ti" nos interesaba ordenar de mayor a menor, por esto, a fin de utilizar la función "sorted" de base sin modificación alguna, lo que se hizo fue realizar la división "ti/bi", esta división ordenada de menor a mayor nos da exactamente el mismo orden que la anterior ordenada de mayor a menor.

Luego de esta aclaración, seguimos con el desarrollo.

12. ¿Los valores de "bi" y "ti" afectan a la optimalidad y la complejidad computacional esperada?

Luego de un análisis, con pruebas propias, donde se ponían valores específicos de "bi" y "ti", buscando tener casos bordes de tamaños muy grandes y muy pequeños de cada uno, no se pudo observar algún cambio aparente y/o relevante para la complejidad o la optimalidad del algoritmo planteado. En todos los casos siempre se llego al mejor resultado posible, y con la complejidad teórica esperada. Además, con la ayuda de la herramienta "Python Matplotlib", se pudieron desarrollar los siguientes gráficos:

12.1. Prueba de 50 elementos

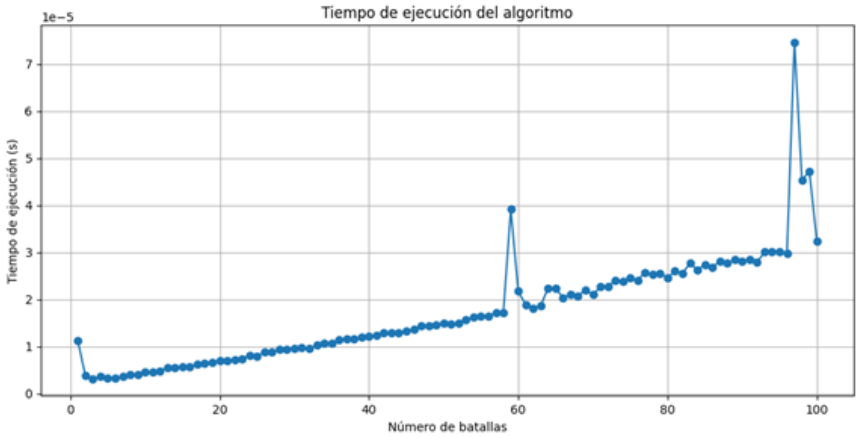

Figura 3: Gráfico de prueba de tiempo de ejecución de los 50 elementos.

12.2. Prueba de 100 elementos

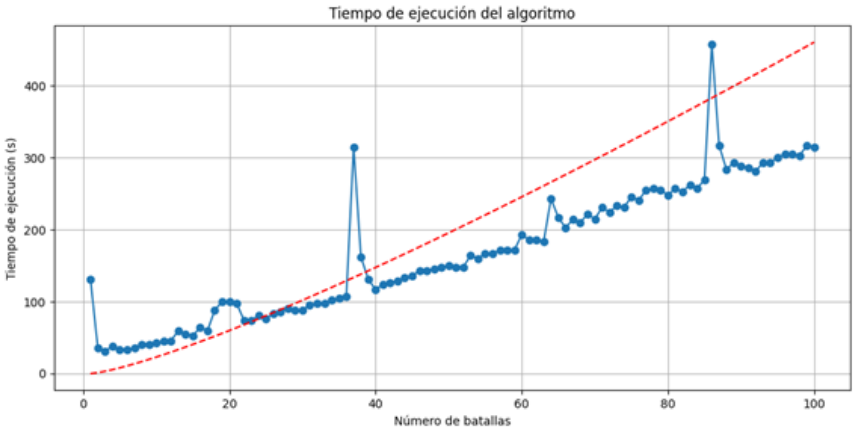

Figura 4: Gráfico de prueba de tiempo de ejecución de los 100 elementos.

12.3. Prueba de T$_A lto$

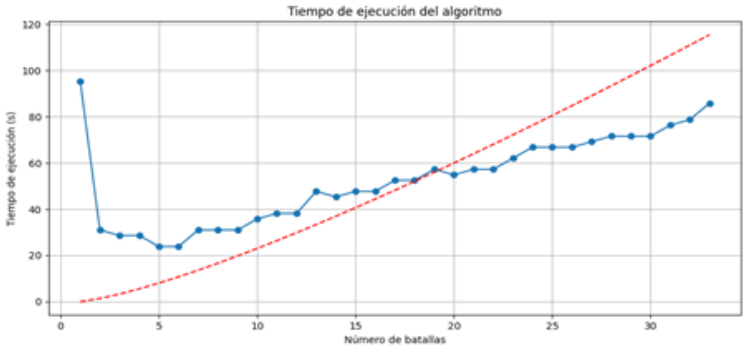

Figura 5: Gráfico de prueba de tiempo de ejecución de test T$_A lto$.@

12.4. Prueba de T$_B ajo$

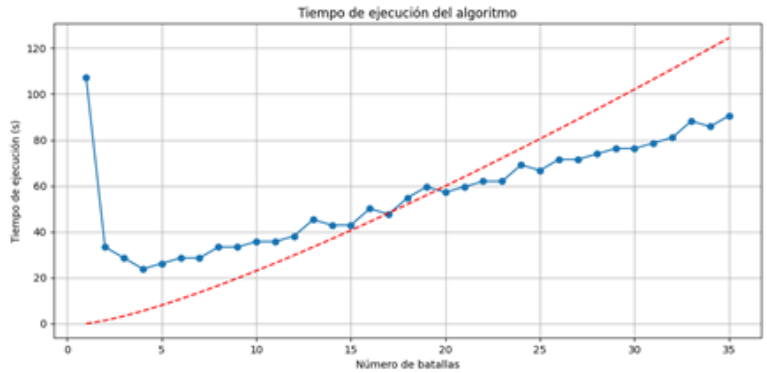

Figura 6: Gráfico de prueba de tiempo de ejecución de test T$_B ajo$.@

12.5. Prueba de B$_A lto$

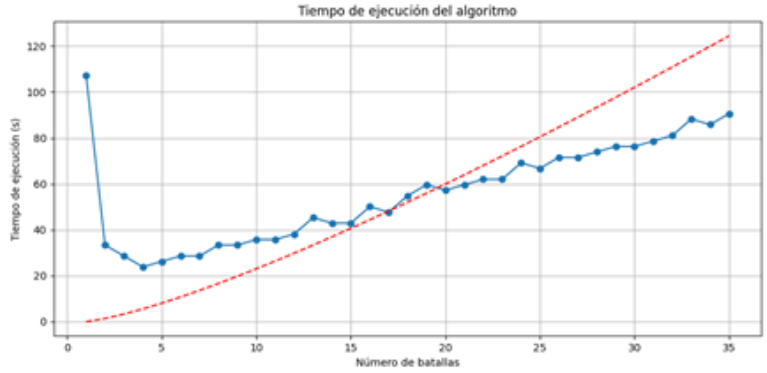

Figura 7: Gráfico de prueba de tiempo de ejecución de test B$_A lto$.@

12.6. Prueba de $B_{B}ajo$

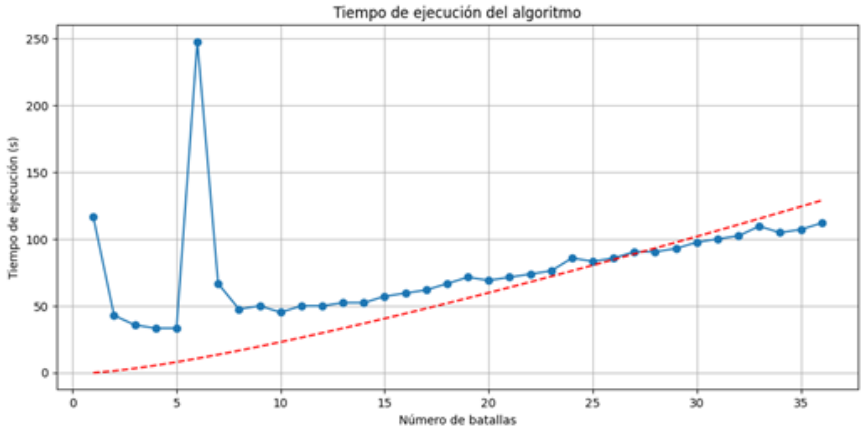

Figura 8: Gráfico de prueba de tiempo de ejecución de test $B_{B}ajo$.@

Aclaración: La escala de la curva azul en todos los gráficos se aumentó un 10**7 para poder apreciarla mejor en comparación con la curva roja teórica de función n log(n).

A fin de corroborar, que lo que estamos viendo es una función "n.log(n)", se verifico con la herramienta "GeoGebra", y se pudo demostrar que los resultados obtenidos son favorables a la complejidad computacional teorizada en un principio.

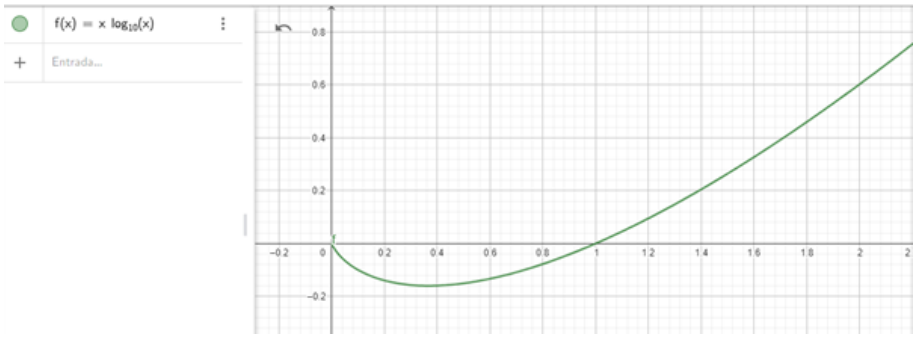

Figura 9: Gráfico de Geogebra de la función "n.log(n)".

13. Testeo del algoritmo

El algoritmo fue probado tanto con las pruebas brindadas por la cátedra, como por las propias creadas (las batallas con nombre, y las de "bi" o "ti" específicos) y siempre cumplió con la condición de devolver la menor sumatoria posible.

Igualmente, en el "README" se encuentra el instructivo para corroborar el óptimo funcionamiento del algoritmo.

Conclusión

14. Conclusiones

Al final del trabajo se consiguió completar todos los requisitos del enunciado, y se pudo obtener resultados exitosos de las hipótesis y propuestas que se fueron desarrollando a lo largo del trabajo.

Algunas de las ideas principales que se lograron completar son:

- Se llegó a la resolución del problema mediante a un "Algoritmo Greedy" óptimo.
- Se pudo verificar la optimalidad del algoritmo de forma analítica.
- Se realizó un código e implementación total del algoritmo pedido en formato funcional.
- Se probó que los valores de "b" y "t" no influyen en la complejidad computacional y tampoco en la optimalidad del algoritmo.
- - Se verificó mediante a gráficos la complejidad teórica, y la tendencia del gráfico a respetar el original de una función "n.log(n)".

Para concluir, se cumplió con todas las expectativas iniciales y se pudo resolver el problema de forma óptima.