

TEORÍA DE ALGORITMOS
(TB024) CURSO BUCHWALD - GENENDER

Trabajo Práctico 2

Programación Dinámica para el Reino de la Tierra

6 de mayo de 2024

Alan Ramiro Cueto Quinto
104319

Carlos Matias Sagastume
110530

Thiago Fernando Baez
110703

1. Introducción

En este trabajo práctico tenemos el objetivo de programar un algoritmo haciendo uso de la técnica de diseño "Programación dinámica". Ba Sing Se, la capital del Reino de la Tierra, se enfrenta a una inminente amenaza: un ataque masivo de la Nación del Fuego. Los Dai Li, la fuerza de seguridad especializada de la ciudad, están a cargo de defenderla utilizando una combinación de habilidades de artes marciales. Utilizando técnicas de Tierra-control, los Dai Li han logrado detectar los planes de la Nación del Fuego para llevar a cabo un asalto en ráfagas. Durante un período de n minutos, se espera que lleguen oleadas de soldados enemigos a la ciudad, con x_i representando el número de soldados que llegarán en el i -ésimo minuto. Sin embargo, los Dai Li tienen una ventaja: la capacidad de utilizar ataques de fisuras, poderosos golpes que pueden destruir a los soldados enemigos acumulando energía durante cierto tiempo. La efectividad de estos ataques está determinada por una función $f(j)$, que indica cuántos soldados pueden ser eliminados después de acumular energía durante (j) minutos. Cuando se decide utilizar un ataque de fisuras en un momento específico, k -ésimo minuto, y han transcurrido (j) minutos desde el último uso, solo se eliminará un número de soldados igual al mínimo entre la cantidad de soldados que llegan en ese minuto x_k y la capacidad de la fisura $f(j)$. Después de usar un ataque de fisuras, se agota toda la energía acumulada. El problema se reduce a determinar cuándo usar los ataques de fisuras para maximizar el número total de enemigos eliminados. Utilizando esta información, podemos establecer una estrategia óptima para decidir cuándo realizar los ataques de fisura a lo largo de los n minutos del ataque enemigo.

2. Algoritmo para encontrar el máximo

```
1  SALTO_DE_PAGINA = "\n"
2  POS_MINUTO_0 = 0
3  MINUTO_INICIAL = 1
4  MAXIMO_INICIAL = 0
5
6
7  def pd(archivo):
8      minutos, oleadas, valoresFuncion = leer_archivo(archivo)
9      valoresFuncion = optimizar_funcion(valoresFuncion, oleadas)
10     optimo_minuto = [POS_MINUTO_0]
11     padres = {}
12     for i in range(MINUTO_INICIAL, minutos + 1):
13         maximo = MAXIMO_INICIAL
14         for j in range(i):
15             valorF = valoresFuncion[j]
16             oleada = oleadas[i-1]
17             eliminados = min(valorF, oleada) + optimo_minuto[i - j - 1]
18             if eliminados >= maximo:
19                 padres[i] = i - j - 1
20                 maximo = eliminados
21             if valorF >= oleada:
22                 break
23         optimo_minuto.append(maximo)
24     return optimo_minuto[minutos], reconstruir_estrategia(padres, minutos)
25
26 def optimizar_funcion(valores_funcion, oleadas):
27     maximo_enemigos = max(oleadas)
28     funcion_optimizada = []
29     for valor in valores_funcion:
30         funcion_optimizada.append(valor)
31         if valor >= maximo_enemigos:
32             break
33     return funcion_optimizada
34
35 def reconstruir_estrategia(padres, minuto):
36     res = []
37     minutos_en_que_atacan = set()
38     minutos_en_que_atacan.add(minuto)
39     actual = padres[minuto]
40     while actual > 0:
41         minutos_en_que_atacan.add(actual)
42         actual = padres[actual]
43     for i in range(1, minuto + 1):
44         if i in minutos_en_que_atacan:
45             res.append("Atacar")
46             continue
47         res.append("Cargar")
48     return ", ".join(res)
49
50
51 def leer_archivo(archivo):
52     oleadas = []
53     funcion = []
54     with open(archivo) as arch:
55         arch.readline()
56         n = int(arch.readline().rstrip(SALTO_DE_PAGINA))
57         for i in range(n):
58             oleadas.append(int(arch.readline().rstrip(SALTO_DE_PAGINA)))
59         for i in range(n):
60             funcion.append(int(arch.readline().rstrip(SALTO_DE_PAGINA)))
61     return n, oleadas, funcion
```

En este algoritmo lo que hacemos es asumir que tenemos la estrategia optima para los $n-1$ minutos anteriores, arreglo al cual llamaremos Opt (cuyo caso base es $\text{Opt}[0] = 0$ ya que en el minuto 0 todavía no se realizó ningún ataque por lo tanto no hubieron bajas enemigas) y sea x la cantidad de enemigos en el minuto n , entonces para el minuto n tenemos que ver que evaluar

entre las posibles cargas que puede tener el ataque en n y evaluar cual es el mejor, siendo para 0 carga la formula $c_i = \min(f(i), x) + \text{Opt}[n - i]$. $\text{Opt}[n]$ siendo n el ultimo minuto, nos dara el optimo que buscamos, ya que es aquel optimo que maximiza las bajas en toda la duracion de la batalla. y lo que nosotros queremos encontrar es el mayor de los c_i por lo tanto la ecuacion de recurrencia para obtener el maximo c_i para cada n es: $\text{Opt}[i] = \max(c_i), 1 \leq i \leq n$. Este algoritmo es de programacion dinamica ya que de manera inductiva se busca resolver el problema, en este caso de forma bottom-up.

2.1. Complejidad

La complejidad del algoritmo propuesto para maximizar el número total de enemigos eliminados es $\mathcal{O}(n^2)$. Sin incluir la lectura de datos desde el archivo, hay dos bucles for: el primero, que itera según la cantidad de minutos n en los que se aparecen las oleadas de enemigos, y el otro, se ejecuta en promedio $(\frac{n}{2})$ veces. Dentro del segundo for, solo hay operaciones constante $\mathcal{O}(1)$, dando como resultado, una complejidad de $\mathcal{O}(n^2)$, siendo n los minutos. Esto se debe a que siendo $T(n)$ el tiempo que toma el algoritmo: $T(n) = T(n - 1) + \mathcal{O}(n)$ Y como el resultado de esta ecuacion de recurrencia es: $\sum_{i=1}^n i = n(n + 1)/2 = n^2/2 + n/2$ y esto acotado nos da que el algoritmo es de complejidad $\mathcal{O}(n^2)$. Además las funciones auxiliares para reconstruir la estrategia, leer el archivo y optimizar los valores de la función dada son todos lineales por lo que no afectan a la complejidad final.

```
1 def pd(archivo):
2     minutos, oleadas, valoresFuncion = leer_archivo(archivo)
3     valoresFuncion = optimizar_funcion(valoresFuncion, oleadas)
4     optimo_minuto = [POS_MINUTO_0]
5     padres = {}
6     for i in range(MINUTO_INICIAL, minutos + 1):
7         maximo = MAXIMO_INICIAL
8         for j in range(i):
9             valorF = valoresFuncion[j]
10            oleada = oleadas[i-1]
11            eliminados = min(valorF, oleada) + optimo_minuto[i - j - 1]
12            if eliminados >= maximo:
13                padres[i] = i - j - 1
14                maximo = eliminados
15            if valorF >= oleada:
16                break
17        optimo_minuto.append(maximo)
18    return optimo_minuto[minutos], reconstruir_estrategia(padres, minutos)
```

3. Variabilidad de valores

3.1. Tiempo del algoritmo

La variabilidad en los valores de las llegadas de enemigos y las recargas si puede afectar el rendimiento del algoritmo:

Valores de las llegadas de enemigos (oleadas): Si las oleadas de enemigos son valores chicos, es decir, si los valores x_i son pequeños en general, es probable que el algoritmo encuentre rápidamente una solución óptima, ya que la funcion rapidamente se volvería mayor a la cantidad de enemigos por oleada y los ciclos que encuentran el máximo para cada minuto serían más cortos. Si la cantidad de los minutos n son muchos, como en el caso de prueba de los 5000 minutos de la cátedra, se puede apreciar como el algoritmo tarda más que con pocos casos, esto último ocurre porque la complejidad temporal del algoritmo es cuadrática respecto al tamaño de la entrada. Pero si hay oleadas inesperadamente grandes en momentos específicos, podría ser difícil para el algoritmo encontrar la estrategia óptima para maximizar la eliminación de enemigos ya que debería iterar a traves de mas valores de la función.

Valores de recarga: Si la función de recarga $f(j)$ es relativamente chica y aumenta lentamente,

es probable que el algoritmo deba iterar mas valores de la función ya que tardaría mas minutos en volverse mayor a las cantidad de enemigos actual. Si la función de recarga tiene valores altos o aumenta rápidamente, el algoritmo podría requerir menor tiempo para encontrar la mejor estrategia ya que con la función optimizar_funcion se dejarían de tomar en cuenta los valores de la función que sean demasiado grandes y entonces las iteraciones serían mas cortas.

3.2. Optimalidad

Si los valores de las llegadas de enemigos varían, es decir, si hay momentos con oleadas de enemigos grandes y otros con oleadas chicas, el algoritmo podría complicarse para encontrar una estrategia óptima debido a que no se podrían podar valores de la función. En oleadas chicas, el algoritmo podría llegar a podar algunos valores de la función dada por lo que funcionaría mas rápido, en cambio en oleadas grandes, probablemente se deban probar todos los valores de la funcion hasta n , siendo n el minuto de la batalla actual.

Variabilidad en los valores de recarga: La variabilidad en los valores de recarga también puede afectar la optimalidad del algoritmo. Como se menciono anteriormente si la funcion de recarga es, por ejemplo, exponencial o cuadrática luego de algunos minutos su valor sería muy grande y probablemente mayor al número de enemigos de cualquier oleada por lo tanto se podrían podar en la función optimizar_funcion por lo tanto para cada minuto se debería iterar a través de menos valores de la función por lo tanto cada iteración seria mas corta, agilizando el algoritmo y probablemente haciendo que este funcione mas veloz que $\mathcal{O}(n^2)$. Por el contrario, si los valores de recarga son bajos, el algoritmo probablemente no pueda podar ningun valor en la función antes mencionada por lo tanto la complejidad sería como la del peor caso, es decir, $\mathcal{O}(n^2)$.

4. Pruebas

Para las pruebas se usaron los siguientes algoritmos:

Para generar las oleadas de enemigos y la funcion de cada batalla se utilizo el siguiente algoritmo:

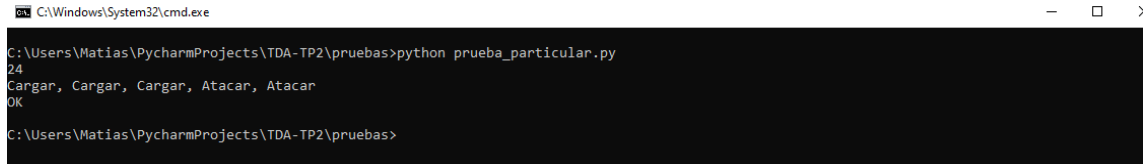
```
1 def random_generator(max_enemigos, max_recarga, minutos):
2     oleadas = []
3     valoresFuncion = []
4     for i in range(minutos):
5         oleadas.append(random.randint(0, max_enemigos))
6         valor_funcion = random.randint(0, max_recarga)
7         for i in range(minutos):
8             valoresFuncion.append(valor_funcion)
9             valor_funcion += random.randint(0, max_recarga)
10    return oleadas, valoresFuncion
```

Ahora se resolvera un caso de una batalla de 5 minutos generada por el algoritmo presentado y se comparará la estrategia obtenida con la del algoritmo: La batalla en cuestion tiene las siguientes oleadas: 25, 8, 4, 23, 9 en este respectivo orden y los valores de la funcion dada son: 1, 10, 14, 23, 31. Notemos que si atacamos sin recargar solo eliminaremos a un enemigo en cualquiera de las batallas, por lo que nunca sera conveniente atacar sin recargar. Notemos que si atacamos antes del minuto 4 eliminaremos a lo sumo 19 enemigos, y si esperamos hasta el minuto 4 eliminaremos almenos 23 por lo que siempre es conveniente atacar recien en el minuto 4 y finalmente si atacamos recien en el minuto 4 en el minuto 5 no nos queda mas que atacar y eliminar a un enemigo mas por lo tanto el maximo numero de bajas que podemos realizar son: 24. Entonces, la estrategia a seguir sera: Cargar, Cargar, Cargar, Atacar, Atacar. A este caso lo guardaremos en el archivo caso1.txt y mediante el siguiente codigo veremos cual es la solucion dada por el algoritmo:

```
1 import pd
2
3 optimo, estrategia = pd.pd("caso1.txt")
4 print(optimo)
5 print(estrategia)
6 assert optimo == 24 and estrategia == "Cargar, Cargar, Cargar, Atacar, Atacar"
```

```
7 print("OK")
```

Corriendo el código tenemos que:



```
C:\Windows\System32\cmd.exe
C:\Users\Matias\PycharmProjects\TDA-TP2\pruebas>python prueba_particular.py
24
Cargar, Cargar, Cargar, Atacar, Atacar
OK
C:\Users\Matias\PycharmProjects\TDA-TP2\pruebas>
```

5. Mediciones

Se realizaron mediciones en base a resolver batallas de entre 1 a 100000 minutos de duracion, de 100 en 100 generadas por el random_generator.py y se resueltas con el algoritmo, midiendo el tiempo que tardaba cada en resolver cada batalla y luego, se hizo un grafico siendo x el tamaño de la entrada (los minutos que dura la batalla) e y los milisegundos que tardo el algoritmo en encontrar la mejor estrategia. Además estas mediciones se realizaron con el algoritmo sin optimizaciones y con para ver las mejoras que trajeron las optimizaciones. Para aplicar el algoritmo de resolución de batallas y graficar lo mencionado anteriormente se utilizo el siguiente código: Para medir el tiempo que tardaba el algoritmo en encontrar la mejor estrategia para una batalla:

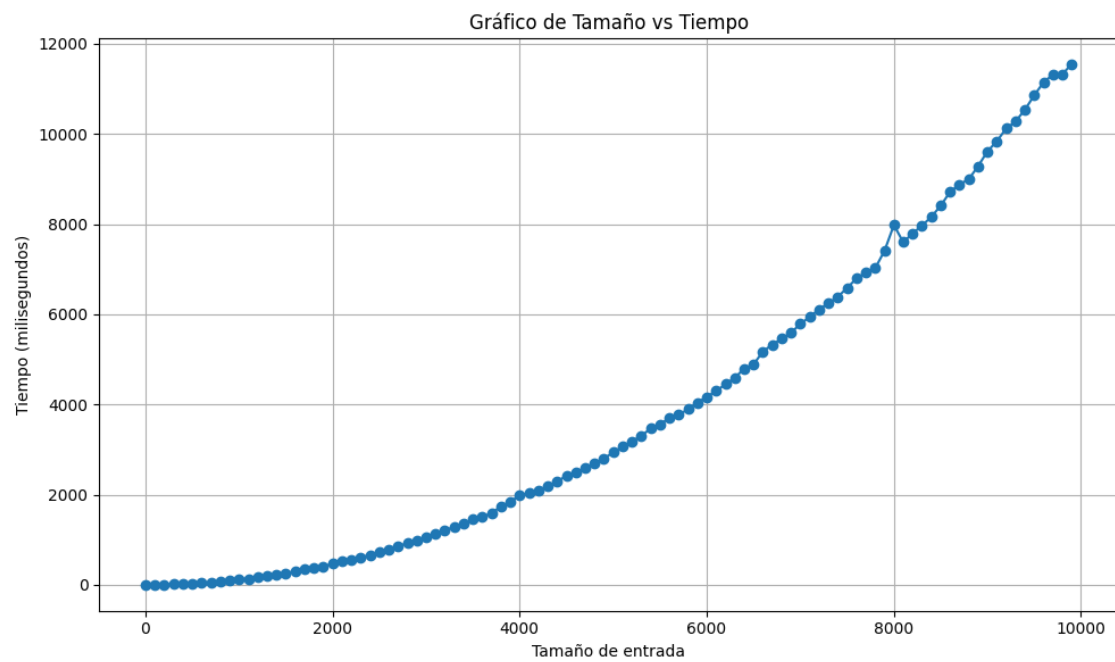
```
1 def crear_tamano_vs_tiempo(max_enemigos, max_recarga, minutos):
2     oleadas, valoresFuncion = random_generator(max_enemigos, max_recarga, minutos)
3     inicio = time.time()
4     pd(minutos, oleadas, valoresFuncion)
5     fin = time.time()
6     duracion = fin - inicio
7     return duracion * 1000
```

Para graficar la cantidad de elementos de una batalla y el tiempo tomado por el algoritmo en encontrar la estrategia además de los parametros pasados al generador de casos:

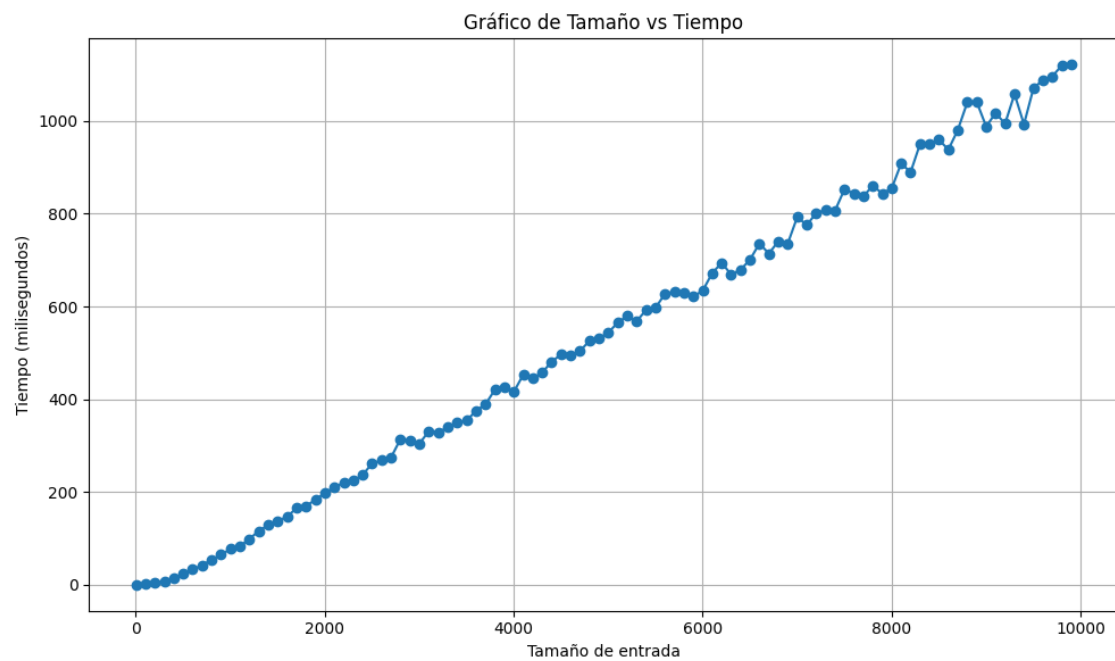
```
1 MAX_ENEMIGOS = 5000
2 MAX_RECARGA = 10
3 TAMANIO_INICIAL = 1
4 TAMANIO_FINAL = 10000
5 TAMANIO_SALTO = 100
6 def graficarFuncionTamanio():
7     listaTamanios = []
8     listaDuraciones = []
9     for i in range(TAMANIO_INICIAL, TAMANIO_FINAL, TAMANIO_SALTO):
10         tiempo = crear_tamano_vs_tiempo(MAX_ENEMIGOS, MAX_RECARGA, i)
11         listaTamanios.append(i)
12         listaDuraciones.append(tiempo)
13     plt.figure(figsize=(10, 6))
14     plt.plot(listaTamanios, listaDuraciones, marker='o', linestyle='--')
15     plt.title('Grafico de Tamanio vs Tiempo')
16     plt.xlabel('Tamanio de entrada')
17     plt.ylabel('Tiempo (milisegundos)')
18     plt.grid(True)
19     plt.tight_layout()
20     plt.show()
```

Y finalmente los resultados fueron:

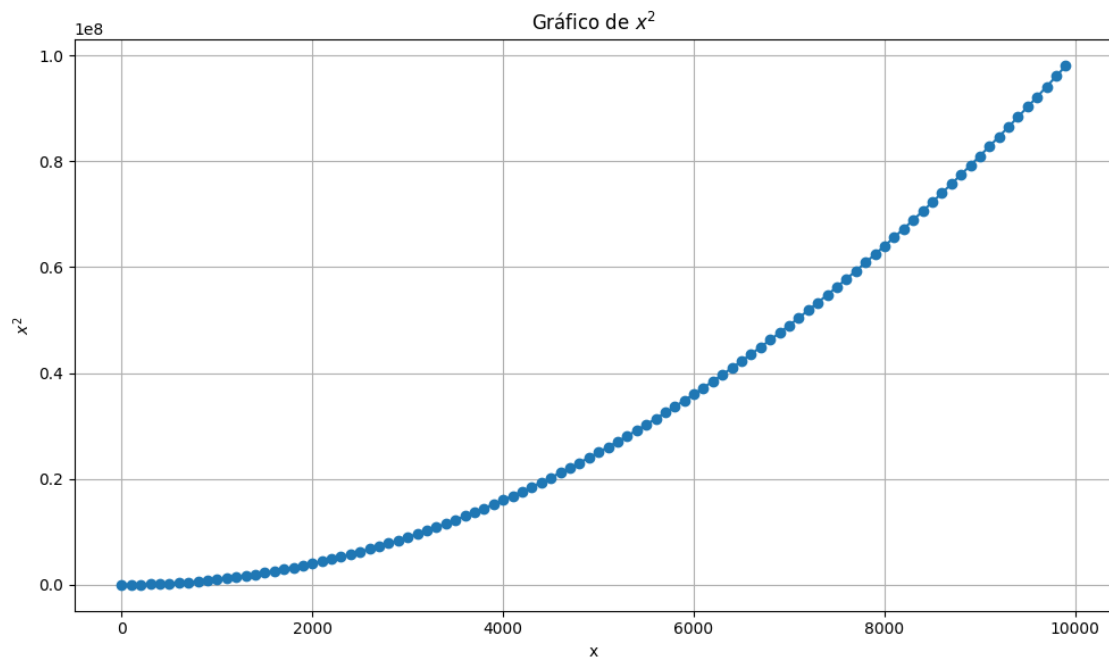
El grafico del algoritmo sin optimizaciones:



El grafico del algoritmo optimizado:



Y finalmente el grafico de x^2 :



Como se puede apreciar, el algoritmo tiene una tendencia menor a cuadrática cuando esta optimizado y cuadrática sin optimizaciones, por lo tanto se puede afirmar que el algoritmo optimizado generalmente es mejor que cuadrático además de que las optimizaciones proporcionan una mejora significativa al algoritmo.

6. Conclusiones

Después de haberse realizado y codificado el algoritmo por medio de la técnica de diseño "Programación Dinámica", con el objetivo de determinar cuando usar los ataques de fisuras para maximizar el número total de enemigos eliminados, se puede concluir que el resultado obtenido fue exitoso. El algoritmo pasa las pruebas propuestas por la cátedra de forma correcta y los valores de las cantidades de tropas eliminadas corresponden con los de los archivos de prueba proporcionados.

En cuanto al tiempo de ejecución, se determinó que el algoritmo tiene una complejidad computacional $O(n^2)$ en el peor de los casos pero generalmente es más veloz. Respecto a la variación de los valores de entrada, se concluyó que para valores grandes, el algoritmo tarda considerablemente más tiempo que cuando los tamaños de entrada son valores pequeños debido a su complejidad y para valores de función de recarga más grandes y valores de oleadas mas pequeños el algoritmo funciona más rapido.

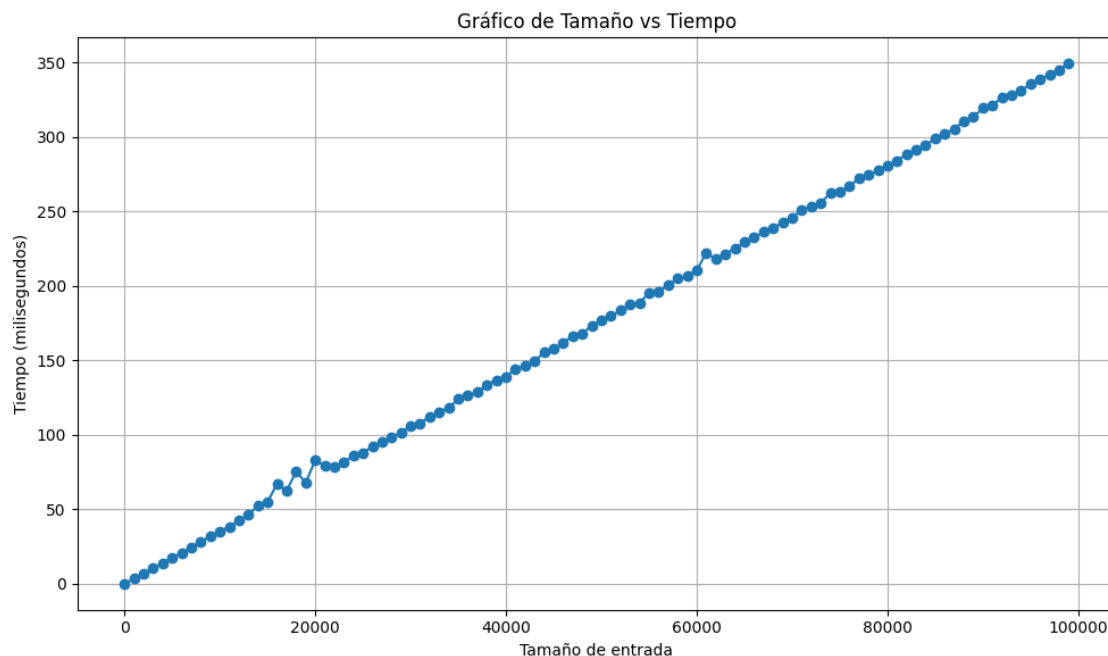
7. Correcciones

7.1. Introducción

Antes que nada es oportuno destacar que este problema resuelto por fuerza bruta seria de complejidad exponencial, ya que para cada batalla se puede elegir atacar o recargar, dandonos de esta manera 2^n combinaciones a evaluar, siendo n la cantidad de batallas. Además, en este problema podemos utilizar programación dinámica para resolverlo, las razones son: primero, que hay un número polinomial de subproblemas, los cuales serian encontrar la estrategia óptima para las batallas sin contar a la ultima batalla, luego suponiendo que tenemos la solución a todos los subproblemas podemos construir la solución al problema original en base a estas subsoluciones debido a como funcionan los ataques (esto se desarrollará mas en profundidad cuando se explique el algoritmo) y finalmente hay un orden natural de estos subproblemas de menor a mayor, siendo el menor de todos la estrategia optima cuando tenemos una batalla y el mayor de todos el problema original (ya que en cada subproblema estamos quitando la última batalla del subproblema inmediatamente mayor).

7.2. Variabilidad de los datos

Por como esta implementado el algoritmo, su complejidad es $O(n^2)$ pero puede haber casos en los que sea un poco mas veloz debido a la optimizacion que se le hizo, mas especificamente cuando se tiene una funcion de recarga que crece muy rapidamente y hordas de enemigo pequeñas. Para probar esto corrimos el algoritmo con los mismos parametros que usamos antes para graficar solo que ahora la funcion dada fue 2^n y en vez de hacerlo hasta 10000 minutos lo hicimos hasta 100000 y se puede observar claramente que aun para batallas de 100000, el algoritmo las resuelve mas rapido que a las batallas de 10000 elementos con la funcion generada aleatoriamente que usabamos para los ejemplos anteriores:



Por lo tanto el algoritmo puede funcionar de manera mas veloz cuando se logran estas 2 condiciones, que la cantidad de enemigos por oleada no sea numerosa y la funcion de recarga crezca muy rapido. Si no ocurre esto, el algoritmo deberia funcionar de acuerdo a su complejidad y no hay ningun caso en el que por la variabilidad de los valores, el algoritmo funcione peor que $O(n^2)$

ya que esta es su cota.

7.3. Llegando a la ecuacion de recurrencia

Para llegar a la ecuacion primero pensamos como se podia descomponer al problema en subproblemas mas pequeños, para esto se nos ocurrio que se podia reducir el problema quitando el ultimo minuto ya que sabiamos que a medida que sacasemos minutos, el problema se volveria mas sencillo de resolver, luego habiendo hecho eso todavia nos quedaba encontrar la forma de utilizar estos subproblemas para resolver el problema final, es entonces cuando se nos ocurre que para encontrar el optimo de cada minuto podriamos utilizar los optimos de los minutos anteriores ya que teniendo la oleada actual deberiamos ver si conviene atacar sin haber recargado o habiendo recargado m minutos y para esto podiamos usar los optimos de los minutos anteriores ya que si recargamos 1 minuto, sabemos que desde el minuto $n - 1$ hasta n no atacaremos pero si atacaremos anteriormente por lo tanto como tenemos el optimo para el minuto $n - 2$, entonces el optimo para el minuto n habiendo recargado 1 minutos es $Min(oleada[n], f[1] + OPT[n - 2])$ y esto lo tendriamos que probar con todos los tiempos de recarga posible, pero la forma de obtener el resultado seria la misma por lo tanto de esta manera llegamos a la ecuacion de recurrencia: $c_i = min(f(i), x) + Opt[n - i]$ siendo c_i recargar i minutos y como nosotros queremos el optimo entonces con la ecuacion: $Opt[n] = max(c_i), 1 \leq i \leq n$ tendremos el optimo de bajas para el minuto n y como nosotros en nuestro algoritmo hacemos una implementacion de estas ecuaciones con ciertas optimizaciones, entonces llegaremos a la solucion optima la cual sera el optimo del ultimo minuto de duracion (ademas el algoritmo tambien se encarga de reecontruir la solucion).

7.4. Ejemplo de ejecucion

Se mostrara un ejemplo de como funciona el algoritmo para facilitar su entendimiento: Se resolvera un caso de una batalla de 5 minutos generada por el algoritmo presentado y se comparará la estrategia obtenida con la del algoritmo: La batalla en cuestion tiene las siguientes oleadas: 25, 8, 4, 23, 9 en este respectivo orden y los valores de la funcion dada son: 1, 10, 14, 23, 31, se elegio este caso ya que anteriormente lo habiamos resuelto y por lo tanto sabemos que la cantidad maxima de bajas es 24. Aplicando el algoritmo tenemos:

1- El optimo para 0 minutos es 0 bajas

2- El optimo para el 1er minuto es 1 baja

3- El optimo para el 2do minuto es el maximo entre atacar sin cargar (1 baja a lo sumo) y sumar el optimo para el 1er minuto (2 bajas) o atacar habiendo cargado una vez (8 bajas ya que ese ataque aniquila a todos los enemigos) y sumar el optimo del minuto 0 (8 bajas), entonces el optimo para el 2do minuto es 8 bajas.

4- el optimo para el 3er minuto empieza siendo en la 1er iteracion el optimo del 2do minuto mas atacar sin recargar, por lo tanto inicialmente el optimo es 9. En la 2da iteracion las bajas actuales son el ataque con 1 minuto recarga mas el optimo del minuto 1 es decir 5, menor al optimo actual y en la 3er y ultima iteracion las bajas actuales son el ataque con 2 minutos de recarga mas el optimo del minuto 0 es decir 4 bajas, menor al optimo actual. Entonces el optimo del 3er minuto es 9 bajas.

5- el optimo para el 4to minuto empieza siendo en la 1er iteracion el optimo del 3er minuto mas atacar sin recargar, por lo tanto inicialmente el optimo es 10 bajas. En la 2da iteracion las bajas actuales son el ataque con 1 minuto de recarga mas el optimo del minuto 2, lo cual son 18 bajas, lo cual es mayor al optimo actual por lo tanto pasa a ser el nuevo optimo actual 18. En la 3er iteracion las bajas actuales son 15 bajas por lo tanto no reemplaza al optimo actual y en la 4ta iteracion las bajas actuales son atacar con 3 minutos de recarga por lo tanto las bajas actuales son 23, mayor al optimo actual por lo tanto el optimo para el minuto 4 termina siendo 23.

6- siguiendo el algoritmo el optimo para el minuto 5 empieza siendo 24 en la 1era iteracion, en la 2da iteracion las bajas actuales son 18, para la 3er iteracion son 17, para la 4ta 9 y para la 5ta y ultima 9 por lo tanto el optimo para el ultimo minuto es 24.

7- finalmente el optimo para el problema termina siendo 24 que coincide con lo calculado con anterioridad.

También el algoritmo reconstruiría la estrategia pero como en este ejemplo se quiere demostrar únicamente que se llega a la solución óptima, no se creyó necesario mostrar como el algoritmo reconstruye la estrategia.

7.5. Complejidad

Como dentro del ciclo hay otro ciclo que itera todos los valores de la función hasta el valor actual de la iteración y como ya vimos que eso es cuadrático, luego el algoritmo es por lo menos $O(n^2)$. Además como dentro de cada uno de los ciclos solamente se realizan acciones $O(1)$, luego la complejidad del algoritmo hasta ahí es cuadrática, finalmente se reconstruye la solución, para lo cual usando el diccionario de padres, se agregan en un set todos los minutos en los que se ataca, esto es a lo sumo $O(n)$ si se fuese a atacar en todos los minutos por lo tanto no afecta a la complejidad final, luego crear res que es el resultado es $O(n)$ ya que se realiza mediante un ciclo de 1 a $n + 1$, luego el `.join` final es también $O(n)$ por lo tanto esta parte del código es $O(n)$, y luego leer el archivo también es lineal ya que es $O(2n)$ lo cual equivale a $O(n)$ así que finalmente, la complejidad del algoritmo es $O(n^2)$.

7.6. Reconstrucción de soluciones

A la hora de reconstruir las soluciones, para saber en qué minuto se deben producir los cada uno de los ataques, planteamos el siguiente algoritmo:

```
1 def reconstruir_estrategia(padres, minuto):
2     res = []
3     minutos_en_que_atacan = set()
4     minutos_en_que_atacan.add(minuto)
5     actual = padres[minuto]
6     while actual > 0:
7         minutos_en_que_atacan.add(actual)
8         actual = padres[actual]
9     for i in range(1, minuto + 1):
10        if i in minutos_en_que_atacan:
11            res.append("Atacar")
12            continue:
13            res.append("Cargar")
14    return ", ".join(res)
```

Así es como funciona el algoritmo:

1. Se inicializa una lista llamada 'res' que se utilizará para almacenar la estrategia reconstruida.
2. Se crea un conjunto para almacenar los minutos en los que es atacado.
3. El minuto indicado se suma al conjunto de 'minutos en los que atacan'.
4. Una variable llamada 'actual' se inicializa con el valor del padre del minuto dado.

En cada iteración del `while`, el valor de 'actual' se agrega al conjunto de 'minutos en los que atacan'. El valor de 'actual' se actualiza con el padre del valor actual. En cada iteración del `for`, se comprueba si el minuto actual está en el conjunto de 'minutos en los que atacan'. Si está en el conjunto, se agrega la cadena 'Ataque' a la lista del resultado. Si no está en el conjunto, se agrega la cadena 'Cargar' a la lista resultante.

En resumen, este algoritmo reconstruye la estrategia de ataque o carga en cada minuto hasta el minuto dado, utilizando la lista principal como referencia.

La complejidad es $O(n)$, tanto la complejidad espacial como la temporal, siendo 'n' la cantidad de minutos. Esto se debe a que al reconstruir las soluciones estamos obteniendo cada uno de los momentos en los que se debe atacar o no.