.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Programación Dinámica para el Reino de la Tierra

21 de mayo de 2024

Ana Gabriela Gutson	Lucas Franciulli	Mariana Juarez Goldemberg
105853	107059	108441

1. Introducción

Es el año 80 DG. Ba Sing Se es una gran ciudad del Reino de la Tierra. Allí tiene lugar el palacio Real. Por esto, se trata de una ciudad fortificada, que ha logrado soportar durante más de 110 años los ataques de la Nación del Fuego. Los Dai Li (policía secreta de la ciudad) la defienden utilizando técnicas de artes marciales, Tierra-control, y algunos algoritmos. Nosotros somos los jefes estratégicos de los Dai Li.

Gracias a las técnicas de Tierra-control, lograron detectar que la Nación del Fuego planea un ataque ráfaga con miles de soldados maestros Fuego. El ataque sería de la siguiente forma:

- Ráfagas de soldados llegarían durante el transcurso de n minutos. En el i-ésimo minuto llegarán x_i soldados. Gracias a las mediciones sísmicas hechas con sus técnicas, los Dai Li lograron obtener los valores de $x_1, x_2, \ldots, x_n$.
- Cuando los integrantes del equipo juntan sus fuerzas, pueden generar fisuras que permiten destruir parte de las armadas enemigas. La fuerza de este ataque depende cuánto tiempo se utilizó para cargar energía. Más específicamente, podemos decir que hay una función $f(\cdot)$ que indica que si transcurrieron j minutos desde que se utilizó este ataque, entonces es capaz de eliminar hasta $f(j)$ soldados enemigos.
- Si se utiliza este ataque en el k-ésimo minuto, y transcurrieron j minutos desde su último uso, entonces se eliminará a mín$(x_k, f(j))$ soldados (y luego de su uso, se utilizó toda la energía que se había acumulado).
- Inicialmente los Dai Li comienzan sin energía acumulada (es decir, para el primer minuto, le correspondería $f(1)$ de energía si decidieran atacar inmediatamente).
- La función de recarga será una función monótona creciente.

Como jefes estratégicos de los Dai Li, es nuestro deber determinar en qué momentos debemos realizar estos ataques de fisuras para eliminar a tantos enemigos en total como sea posible.

1.1. Consigna

1. Hacer un análisis del problema, plantear la ecuación de recurrencia correspondiente y proponer un algoritmo por programación dinámica que obtenga la solución óptima al problema planteado: Dada la secuencia de llegadas de enemigos $x_1, x_2, \ldots, x_n$ y la función de recarga $f(\cdot)$ (dada por una tabla, con lo cual puede considerarse directamente como una secuencia de valores), determinar la cantidad máxima de enemigos que se pueden atacar, y en qué momentos se harían los correspondientes ataques.
2. Escribir el algoritmo planteado. Describir y justificar la complejidad de dicho algoritmo. Analizar si (y cómo) afecta a los tiempos del algoritmo planteado la variabilidad de los valores de las llegadas de enemigos y recargas.
3. Analizar si (y cómo) afecta a la optimalidad del algoritmo planteado la variabilidad de los valores de las llegadas de enemigos y recargas.
4. Realizar ejemplos de ejecución para encontrar soluciones y corroborar lo encontrado. Adicionalmente, el curso proveerá con algunos casos particulares que deben cumplirse su optimalidad también.
5. De las pruebas anteriores, hacer también mediciones de tiempos para corroborar la complejidad teórica indicada. Realizar gráficos correspondientes. Generar todo set de datos necesarios para estas pruebas.
6. Agregar cualquier conclusión que parezca relevante.

1.2. Ejemplo

Dada los siguientes datos, deberíamos atacar en el minuto 3 y 5 y obtener una cantidad de 1413 tropas eliminadas.

n	x_i	$f(.)$
1	271	21
2	533	671
3	916	749
4	656	833
5	664	1543

Cuadro 1: Ejemplo

Resolución

2. Análisis del problema

Definimos las siguientes variables a partir del problema:

- n: número total de minutos en los que se produce el ataque enemigo.
- x_i: cantidad de enemigos que llegan en el minuto i.

Y tomamos en cuenta que contamos con:

- $x_1, x_2, \ldots, x_n$: la secuencia de llegadas de enemigos en cada minuto i, con x_i representando el número de enemigos que llegan en el minuto i.
- $f(\cdot)$: la función de recarga, que indica cuántos enemigos podemos eliminar en función del tiempo transcurrido desde el último ataque, la cual es una función monótona creciente.

3. Análisis para encontrar la solución por Programación Dinámica

El objetivo es pensar cómo distribuir nuestros ataques a lo largo del tiempo para maximizar el número total de enemigos eliminados.

Por cada minuto tengo 2 opciones: ¿ataco o no ataco?

Si ataco en un minuto en específico, la cantidad máxima que podré eliminar de enemigos estaría condicionada entre la energía que tengo disponible para atacar y la cantidad de enemigos que se encuentren en ese minuto. Si no ataco, mantengo la cantidad de enemigos heridos desde la última vez que ataqué y acumulo la energía para futuros ataques (mantengo la solución del subproblema anterior).

3.1. ¿Cómo podría saber cuándo conviene atacar?

Utilizando "*memoization*", para almacenar los resultados intermedios, construyendo la solución global de forma iterativa considerando cada ataque individualmente y evaluando la mejor opción entre atacar o no atacar en cada minuto, teniendo en cuenta los tiempos de recarga.

3.2. Componentes

Para nuestro caso en particular tenemos:

1. El caso base:
   - Cuando no han atacado enemigos todavía ($n = 0$), por lo tanto no resultará afectado ningún enemigo.
2. La forma que tienen los subproblemas.
   - Para el minuto i será la cantidad de enemigos atacados hasta ese tiempo que maximice el ataque, teniendo en cuenta la energía acumulada por la función de recarga.
3. La forma en que dichos subproblemas se componen para solucionar subproblemas más grandes.

- Comenzamos resolviendo los subproblemas más simples y avanzamos hacia subproblemas más grandes. Comenzamos con el subproblema de utilizar el primer ataque, luego el segundo, y así sucesivamente, hasta llegar al subproblema de utilizar todos los ataques disponibles (iteración de tipo $bottom - up$).

4. Ecuación de recurrencia

La ecuación de recurrencia que hemos seguido para resolver el problema por programación dinámica es:

$$OPT(i) = \text{máx}_{0 \leq j \leq i} \left(OPT(j) + \text{mín}(x(i), f(i-j)) \right)$$

Donde:

- i: ataque actual.
- j: último ataque antes del ataque actual.
- $OPT(i)$: cantidad óptima (máxima) de enemigos eliminados hasta el minuto i.
- $OPT(j)$: cantidad óptima (máxima) de enemigos eliminados hasta el minuto j (siendo $0 \leq j \leq i$).

Lo que se busca es encontrar el máximo de la cantidad de enemigos que se pueden eliminar en el ataque actual i. Será el máximo valor de los minutos anteriores a i (los j) (teniendo en cuenta los valores del ataque actual) determinado por:

- El mínimo entre el valor de $x(i)$ (cantidad de enemigos en el minuto i), que son los posibles a eliminar en este ataque actual y $f(i-j)$ que será la cantidad de enemigos a eliminar desde el último ataque (con el recargo de i "$resto$" lo utilizado en j).
- Sumado al óptimo de enemigos eliminados hasta el minuto j.

Como se observa, esta representación toma en cuenta que el primer ataque se realizará en el minuto 0, que sería más acorde al índice de un vector. A la hora de implementar nuestro código, decidimos comenzar desde el minuto 1 ya que nos parecía más cercano a la naturaleza del problema.

5. Implementación en código funcional del algoritmo

Para encontrar la mejor combinación posible, la cual maximice el daño, utilizamos la siguiente función `max_enemies_eliminated()`, la cual devuelve el máximo número de enemigos eliminados (`dp[n]`) y la lista de momentos en los que se realizaron los ataques correspondientes (`attack_times[n]`).

Esta función internamente implementa dos bucles anidados, donde:

- El bucle que itera sobre i (el exterior) desde 1 hasta n inclusive, evalúa todos los ataques anteriores a cada i y encuentra el que de como resultado mayor cantidad de enemigos atacados a partir del bucle interno.
- El segundo bucle, el cual itera sobre j desde 1 hasta $i+1$, representa todos los momentos anteriores al momento actual i en los que se podría haber realizado un ataque. Dentro, se calcula el número de enemigos que se pueden eliminar si se realiza un ataque en el momento i. Esto se hace tomando el mínimo entre el número de enemigos en el momento i y la cantidad de energía recargada en el momento $i-j$, donde j es el momento anterior más cercano en el que se realizó un ataque. Luego, se compara ésta cantidad con la solución óptima anterior. Si es mayor, se actualiza la solución óptima y se registra el momento en el que se realizó el ataque en la lista `attack_times`.

Para almacenar resultados intermedios se utilizan las variables `dp` y `attack_times`.

```python
def max_enemies_eliminated(x, f):
    n = len(x)
    dp = [0] * (n + 1)
    attack_times = [[] for _ in range(n + 1)]

    for i in range(1, n + 1):
        for j in range(1, i + 1):
            new_enemies = min(x[i - 1], f[i - j])
            if dp[j - 1] + new_enemies > dp[i]:
                dp[i] = dp[j - 1] + new_enemies
                attack_times[i] = attack_times[j - 1] + [i]

    return dp[n], attack_times[n]
```

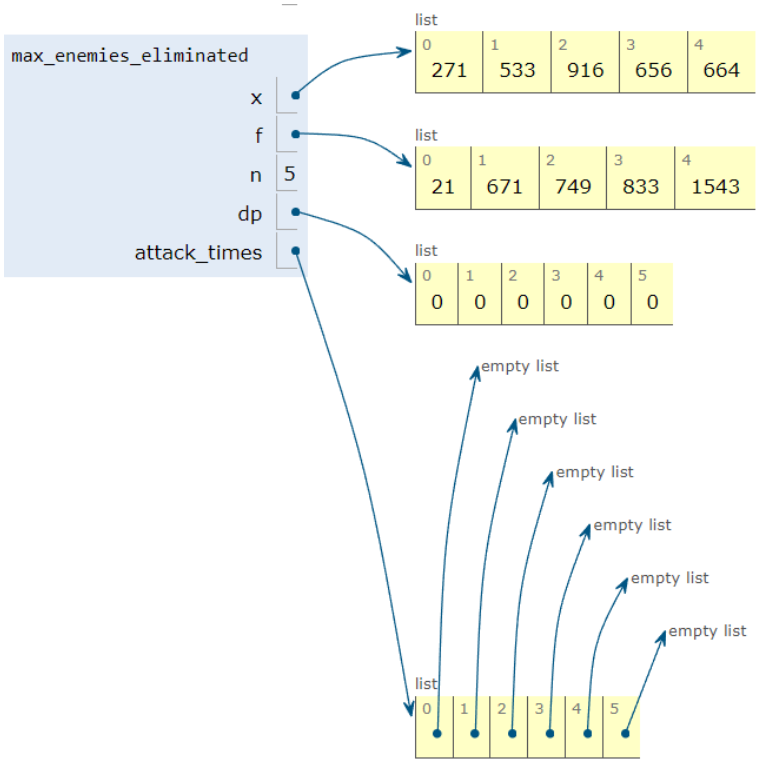
5.1. Explicación visual del algoritmo

A continuación vamos a mostrar paso por paso cómo se resuelve el caso de **5.txt** con nuestro algoritmo.

Recordemos que los valores de **5.txt** son:

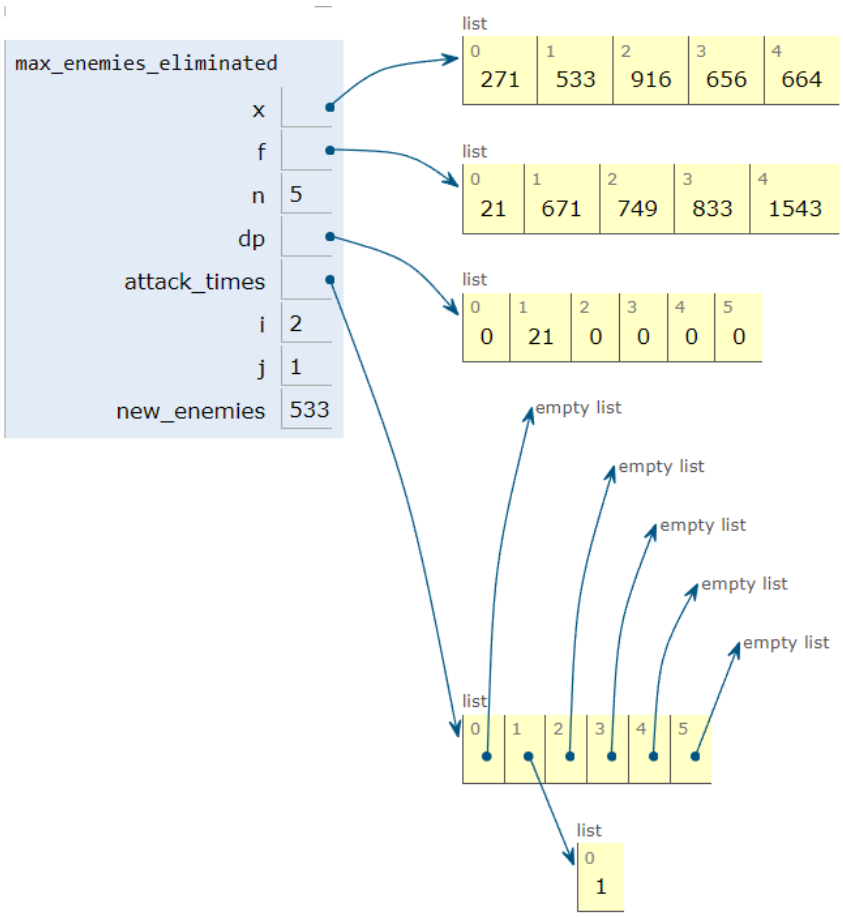
```
x = [271, 533, 916, 656, 664]
f = [21, 671, 749, 833, 1543]
```

En primer lugar se inicializa `n` según el largo de los array de ataques (es decir, `x`). Una vez se tiene `n` se crea un array de lleno de ceros con un largo de $n + 1$ llamado `dp`. Luego se crea una matriz $(n + 1) \times (n + 1)$ vacía.

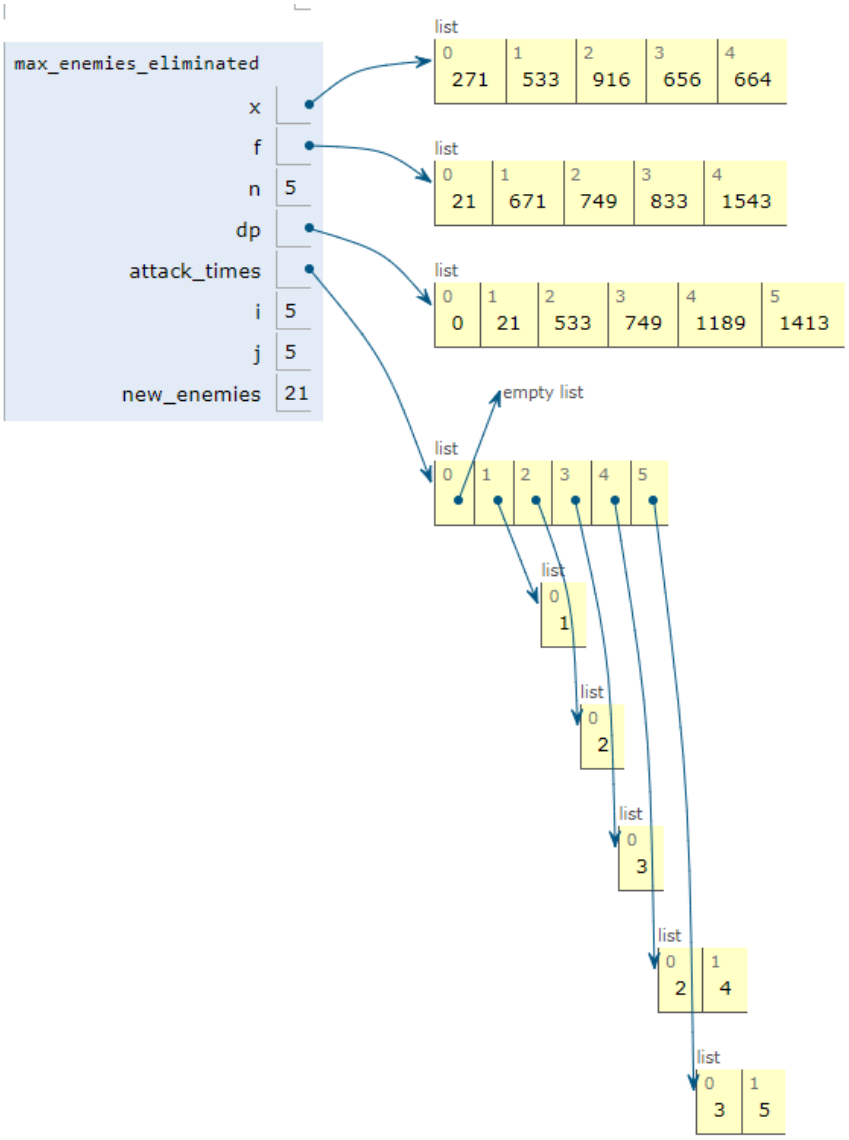


Al tener todo inicializado correctamente se empieza iterar sobre `dp`. Primero se encuentra el menor numero entre `x[i - 1]` y `f[i - j]`. En la primer iteración como `i=1` y `j=1` entonces

`x[0]= 271` y `f[0] = 21`, por lo tanto la variable `newEnemies = 21`. Una vez se tiene calculada la variable `newEnemies` entonces se compara `dp[j - 1] + newEnemies` si es mayor que `dp[i]`. En este caso `dp[0] + newEnemies = 21` y `dp[1] = 0`, entonces al cumplirse que es mayor se reemplaza `dp[i]` por este valor, en este caso `dp[1] = 21`. Y attackTimes va a ser reemplazado por `attackTimes[i] = attackTimes[j - 1] + [i]`



Si iteramos esto varias veces podemos ver como en `attackTimes` obtenemos en el último elemento del array, otro array con el orden de cuándo atacar y en el último elemento de `dp` la cantidad total de tropas enemigas eliminadas.



5.2. Prueba del correcto funcionamiento del algoritmo

Para verificar el correcto funcionamiento del algoritmo, decidimos visualizar los minutos en los que se realizan ataques, obteniendo:

Ejemplo 5.txt

Minuto 1	Minuto 2	Minuto 3	Minuto 4	Minuto 5	Daño
Cargar	Cargar	Atacar	Cargar	Atacar	1413

Ejemplo 10.txt

Minuto 1	Minuto 2	Minuto 3	Minuto 4	Minuto 5	Minuto 6	Minuto 7	Minuto 8	Minuto 9	Minuto 10	Daño
Cargar	Cargar	Atacar	Atacar	Cargar	Atacar	Atacar	Cargar	Cargar	Atacar	2118

Ejemplo 10_bis.txt

Minuto 1	Minuto 2	Minuto 3	Minuto 4	Minuto 5	Minuto 6	Minuto 7	Minuto 8	Minuto 9	Minuto 10	Daño
Cargar	Cargar	Cargar	Atacar	Cargar	Atacar	Cargar	Atacar	Cargar	Atacar	1237

Ejemplo 20.txt

Minuto 1	Minuto 2	Minuto 3	Minuto 4	Minuto 5	Minuto 6	Minuto 7	Minuto 8	Minuto 9	Minuto 10	Minuto 11	Minuto 12	Minuto 13	Minuto 14	Minuto 15	Minuto 16	Minuto 17	Minuto 18	Minuto 19	Minuto 20	Daño
Atacar	Atacar	Atacar	Atacar	Atacar	Atacar	Atacar	Atacar	Atacar	Atacar	Atacar	Atacar	Atacar	Atacar	Atacar	Atacar	Atacar	Atacar	Atacar	Atacar	11603

El algoritmo utilizado puede ser encontrado en **este Google Colaboratory**.

6. Complejidad

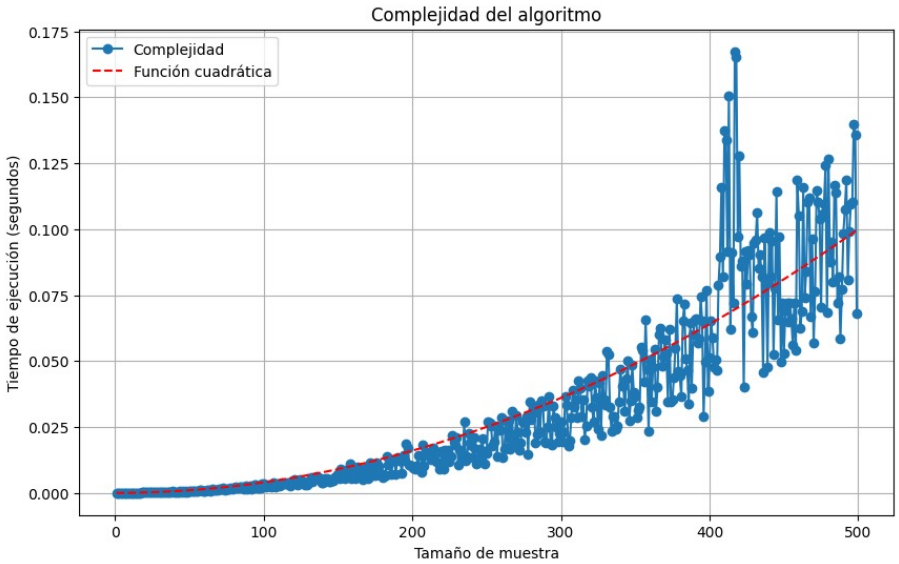
Dada la implementación de nuestro algoritmo, observamos que la complejidad depende directamente de los 2 bucles anidados que recorren para cada minuto i, todas las combinaciones posibles de ataques y la cantidad de heridos en cada uno, ya que el resto (operaciones de comparación, asignaciones) son de tiempo constante. Por lo tanto:

$$T(n) = O(n^2)$$

Esto indica que el tiempo de ejecución del algoritmo aumenta cuadráticamente en relación a la cantidad de entradas, lo que significa que la eficiencia del algoritmo disminuye a medida que el número de minutos disponibles para realizar ataques aumenta.

6.1. Estudio de variabilidad y complejidad algoritmica

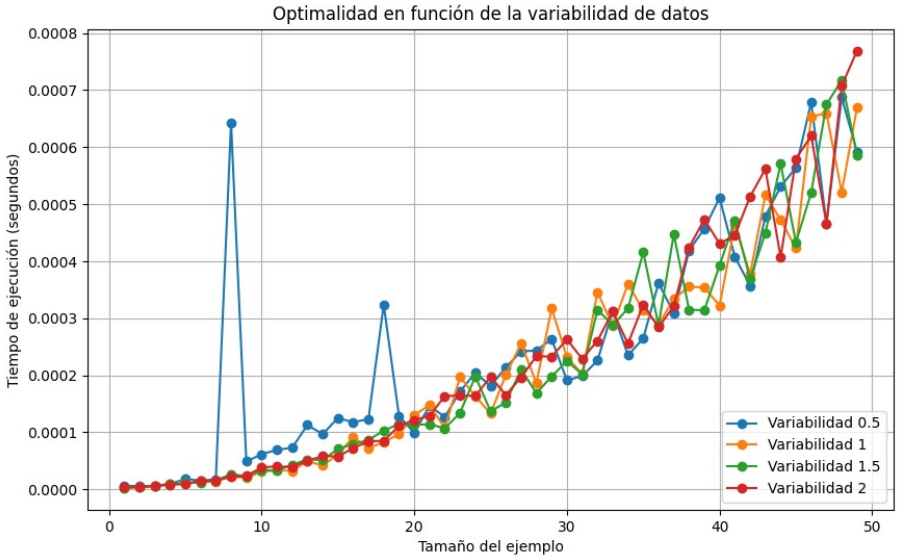
Realizamos un set de datos con el módulo *random* (respetando que la función de recarga debe ser monótona creciente) donde aumentamos la cantidad de pares $x_i, f(.)$ de manera progresiva de a 1 hasta 500 elementos, para lograr graficar cómo va variando la complejidad del algoritmo. A partir del gráfico con los datos, incorporamos una función cuadrática para demostrar que cumple la complejidad con esa función.



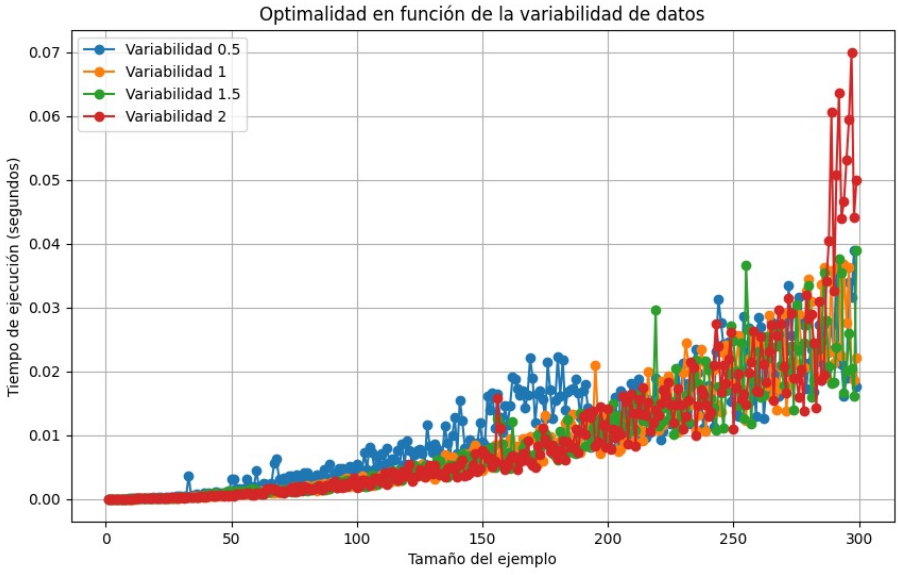
Para estudiar cómo afecta a la optimalidad del algoritmo la variabilidad de la secuencia de ataques y la función de recarga, realizamos un gráfico donde simulamos una dispersión de los datos

dado un campo de variabilidad, que controla cómo cambian los valores de los datos en función del tamaño del conjunto de datos. Un nivel bajo de variabilidad indica que los valores de los datos no cambian mucho a medida que aumenta el tamaño del conjunto, mientras que un nivel alto indica lo contrario.

El siguiente gráfico muestra la variabilidad dados 50 conjuntos de datos:



También, estudiamos con 300 conjuntos de datos:



Observamos que, salvo ciertos casos particulares (donde podría ser que el módulo random genere datos "*extremos*"), la variabilidad no afecta a la optimalidad.

7. Conclusión

Al finalizar este trabajo práctico, hemos implementado un algoritmo basado en programación dinámica para resolver el problema de determinar en qué momentos realizar ataques para eliminar

la mayor cantidad posible de enemigos en un ataque ráfaga planificado por la Nación del Fuego sobre Ba Sing Se.

Durante el desarrollo del algoritmo, observamos que:

- Utilizamos programación dinámica para resolver el problema, dividiéndolo en subproblemas más pequeños y almacenando las soluciones óptimas de estos subproblemas para evitar recálculos innecesarios.
- Definimos un caso base para cuando no ha habido ningún ataque enemigo y establecimos la forma en que los subproblemas se componen para resolver problemas más grandes.
- Encontramos una ecuación de recurrencia que nos permitió calcular la cantidad óptima de enemigos eliminados hasta el minuto i basándonos en los ataques anteriores y la cantidad de enemigos eliminados en esos ataques.

En resumen, hemos desarrollado un algoritmo eficiente que logra encontrar la solución óptima al problema planteado, proporcionando el resultado esperado en un tiempo de ejecución razonable dada la complejidad temporal que posee.

8. Correcciones

8.1. ¿Qué es la variabilidad presente en los gráficos?

El parámetro de variabilidades que se encuentra en los gráficos, determina la dispersión de los datos de entrada con respecto al tamaño del ejemplo. Con una variabilidad más alta, los valores generados serán más "*extremos*", aumentando la variabilidad de la muestra.

Así, pudimos evaluar cómo se comporta el algoritmo bajo diferentes niveles de dispersión de datos.

8.2. Reconstrucción de la solución

Para solucionar el orden cúbico que se genera a a partir de repetir la operación lineal

```
    attack_times[i] = attack_times[j - 1] + [i]
```

Añadimos una función de reconstrución de la solución:

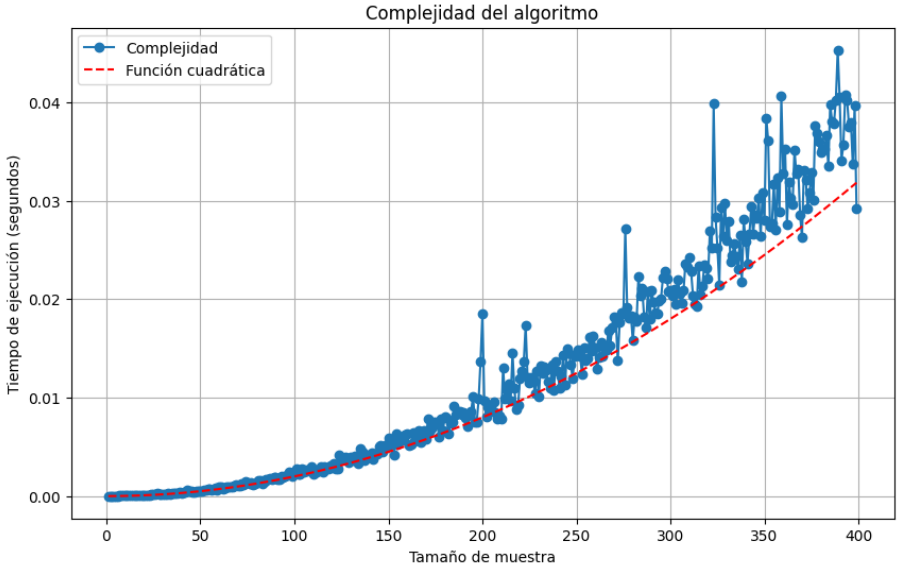
```
    def solution_reconstruction(n, prev_index):
        solution = []
        while n > 0:
            solution.append(n)
            n = prev_index[n]
        return solution
```

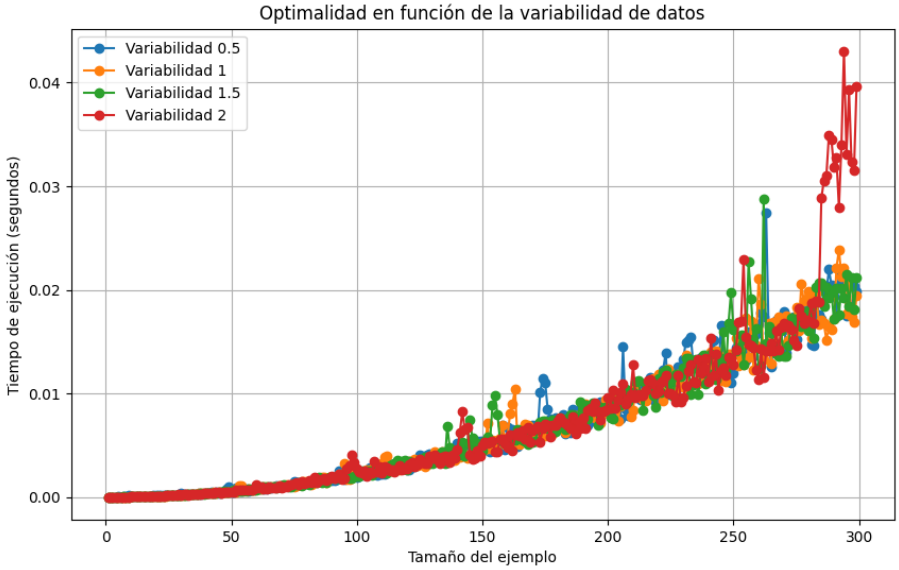
Siendo agregada al código de la siguiente manera:

```
    def max_enemies_eliminated(x, f):
        n = len(x)
        dp = [0] * (n + 1)
        prev_index = [0] * (n + 1)

        for i in range(1, n + 1):
            for j in range(1, i + 1):
                new_enemies = min(x[i - 1], f[i - j])
                if dp[j - 1] + new_enemies > dp[i]:
                    dp[i] = dp[j - 1] + new_enemies
                    prev_index[i] = j - 1

        solution = solution_reconstruction(n, prev_index)

        return dp[n], solution[::-1]
```

De esta manera, el vector `prev index` almacena el índice del último elemento que contribuyó a la solución óptima de los subproblemas anteriores que llevaron a la solución óptima para el minuto i. Reconstruimos la solución volcando los elementos de `prev index` hacia una lista y la invertimos para obtener el orden correcto de la misma.

Realizamos nuevamente los gráficos de complejidad y variabilidad sobre los datos:





Observamos que, como era lo esperado dada la corrección, el tiempo de ejecución se redujo de manera satisfactoria. Agregando el archivo compartido con 10000 conjuntos de prueba a nuestra carpeta de ejemplos, cumplimos con el tiempo estimado de ejecución (alrededor de 20 segundos), además de notar que con los ejemplos más chicos, también observamos disminución del mismo.