.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Programación Dinamica

1 de junio de 2024

Santiago Varga
110561

Morena Sandroni
110205

Nahuel Cellini Rotunno
103320

1. Introducción

En este trabajo, presentamos un algoritmo diseñado utilizando la técnica de Programación Dinámica. El propósito del mismo es ayudar a los Dai Li a encontrar la combinación óptima de decisiones (atacar o cargar) tal que se eliminen a tantos enemigos como sea posible. El problema indica que cada n minutos, en el i-ésimo llegan x_i soldados, para eliminar a estos, al unir sus fuerzas, los integrantes del equipo pueden generar fisuras, ahora bien, la fuerza de este ataque depende de cuánto tiempo se usó para cargar energía, dado que existe una función $F(j)$ que indica que si transcurrieron j minutos desde que se utilizó el ataque, entonces es capaz de eliminar hasta $F(j)$ soldados enemigos.

A lo largo de este trabajo, no solo analizaremos el funcionamiento del algoritmo, sino que también evaluaremos su complejidad temporal. Además, exploramos su capacidad de adaptación a diferentes escenarios, considerando variaciones en los datos y el tamaño de entrada. En las siguientes secciones, vamos a explorar cómo llegamos al algoritmo óptimo, teniendo en cuenta que atravesamos diferentes etapas antes de dar con la solución ideal. Analizaremos cómo funcionaban estos algoritmos, por qué llegamos a ellos y cómo nos dimos cuenta de que no eran óptimos. Evaluaremos su desempeño en varios contextos y, al final, daremos una conclusión general.

2. Análisis del problema

El algoritmo propuesto se ha desarrollado con la meta de alcanzar la solución óptima, lo que implica encontrar el ordenamiento ideal que le permita a los Dai Li concluir cuando es conveniente atacar y cuándo es conveniente recargar, teniendo en cuenta las siguientes condiciones:

- Los minutos de carga j disminuyen a 0 cuando se decide atacar.
- En el minuto i de batalla llegarán x_i soldados, cuya cantidad puede ser mayor o menor a la cantidad anterior. (los soldados anteriores no se mantienen)
- La cantidad de soldados eliminados, si se decide atacar, es de $\min(x_i, F(j))$.
- Se comienza sin energía, osea para el primer minuto le corresponde $F(1)$.
- Independientemente de la cantidad de soldados y el tiempo transcurrido desde el último ataque, en el último minuto siempre lo mejor es atacar, ya que esto sumará a la cantidad de enemigos derrotados.

Una vez planteadas estas condiciones podemos comenzar a buscar la ecuación de recurrencia que nos permita resolver el problema.

3. Algoritmos propuesto

3.1. Primer algoritmo propuesto

El primer algoritmo propuesto fue nuestra primera aproximación e intenta maximizar la cantidad total de enemigos derrotados a través de comparar las dos siguientes instancias: Por un lado, $G(i-1)$ es el óptimo del minuto anterior y $G(1)$ es el óptimo del primer minuto, en el cual siempre conviene atacar antes de no hacer nada, aunque no haya carga. Por otro lado tenemos la suma de la cantidad de enemigos que pueden ser derrotados si llevamos una carga j ($\min(x_i\ ,\ F(j))$).

$$G(i) = \max(G(i-1) + G(1),\ (\min(x_i\ ,\ F(j)))\ j = i$$

Este algoritmo tiene algunos problemas que hace que no sea para nada eficiente. El aspecto más importante es que se está dando por hecho que j sólo puede tomar el valor i. Esto es un error ya que no se está teniendo en cuenta todas las posibles cargas anteriores. Deberíamos poder acceder a lo que sucede si se carga en cada uno de los minutos anteriores, es decir, si reiniciáramos j antes del minuto i. El algoritmo, por lo tanto, se encuentra incompleto y hubo que descartarlo.

3.2. Segundo algoritmo propuesto

Después de identificar los problemas en la ecuación de recurrencia anteriormente planteada, concluimos que habría que encontrar una manera de que la variable de carga j pudiera tomar todos los valores de 0 hasta i para poder evaluar todos los casos posibles en cada iteración, que corresponde a cada minuto i.

De esta forma, llegamos a una segunda ecuación posible en la cual se comparan todas las instancias posibles hasta el minuto i. El resultado óptimo es, entonces, el valor máximo de: el óptimo del minuto j sumado a la cantidad de enemigos que podrían ser derrotados en el minuto i con la carga $i - j$; todo esto es evaluado para cada j entre (0) e $i - 1$. Con $G(i)$ siendo el valor óptimo en el minuto i entonces tenemos:

$$G(i) = \max(G(j) + \min(x_i,\ F(i-1)\ ))\ 0 =< j < i$$

Con este algoritmo logramos representar todos los casos posibles para cada minuto i y determinar el de valor máximo. Esto es debido a que, para cada iteración, tomamos cada uno de los óptimos anteriores y los sumamos a la cantidad posible de enemigos a derrotar con cada todas las cargas disponibles.

Pero, aunque hayamos encontrado el algoritmo que devuelve siempre de manera óptima la máxima cantidad de enemigos que se pueden derrotar, la otra parte de la consigna nos pide que devolvamos la secuencia de movimientos a realizar para conseguir estos resultados. El problema surge entonces cuando intentamos realizar ingeniería inversa sobre este algoritmo, ya que encontrarle una forma de retornar hacia atrás para conocer la secuencia de "ataque" y "recarga" resulta sino imposible, extremadamente complejo.

Es por esto que nos pareció lo más razonable encontrar un nuevo algoritmo con una nueva ecuación de recurrencia que nos facilite este inconveniente. La idea era plantear el problema de una forma matricial, de manera que se puedan visualizar mejor los posibles resultados para cada instancia.

3.3. Algoritmo Optimo

Buscamos entonces hacer un algoritmo que además de ser eficiente, permita reconstruir el camino de manera sencilla. Para esto optamos por una representación matricial, cuyas filas representan el minuto de batalla y las columnas el minuto de ataque (habiendo cargado los minutos anteriores).

$$G[i][j] = \begin{cases} min(x[i], F[j]) & \text{si } j = 1 \\ max(G[i-j] + min(x[i], F[j]) & \text{si } j < 1 \end{cases}$$

Dentro de la diagonal de la matriz vamos a tener "Atacar en el minuto n habiendo cargado por $n-1$ minutos", lo cual significa que no hubo ataques anteriormente. Esto se traduce básicamente en min($x[i]$, $F[j]$) siendo $j = j = n$. Para calcular el resto de los valores, además de calcular min($x[i]$, $F[j]$) necesitamos sumarle el valor de soldados derrotados anteriormente, ya que para estos casos sabemos que hubo al menos un ataque anteriormente. En este caso los valores de j ¿i no nos interesan porque no tiene sentido atacar en el minuto $n + m$ minutos si solo transcurrieron n minutos de batalla.

Para poder saber dónde encontrar el valor de soldados anteriormente tenemos que entender bien qué significa "Atacar en el minuto i habiendo cargado por $j-1$ minutos"

- Si j vale 1, entonces significa que se cargó por 0 minutos, es decir que el último ataque fue en el minuto anterior, en el minuto $i-1$.
- Si j vale 2, entonces significa que se cargó por 1 minuto, es decir que el último ataque fue hace 2 minutos, en el minuto $i-2$.
- Si j vale n, entonces significa que se cargó por $n-1$ minutos, es decir que el último ataque fue hace n minutos, en el minuto $i-n$.

Esto podemos generalizarlo entonces como que el ataque anterior se encuentra en el minuto $i-j$. Y en ese minuto tengo que buscar el máximo valor, ya que esa es la cantidad de soldados máximos derrotados para ese subproblema.

Pero esta solución en particular tiene un problema, estamos buscando el máximo para un minuto dado constantemente, cuando eso es algo que podríamos tener almacenado de antemano. Para solucionar este problema decidimos que en vez de buscar siempre el máximo, podríamos solo hacerlo cuando se termina de calcular un subproblema anterior. Entonces guardamos en la primera columna de la fila el valor del máximo de esa fila, y luego, cuando queremos saber el máximo valor para el minuto, lo podemos buscar en $G[i-j+1][0]$. La ecuación de recurrencia entonces quedaría de la siguiente forma:

$$G[i][j] = \begin{cases} max(G[i-1][k]) & \text{si } j = 0, 1 <= k <= i \\ min(x[i], F[j]) & \text{si } j = i \\ G[i-j+1][0] + min(x[i], F[j]) & \text{si } j < i \end{cases}$$

Finalmente nuestro codigo es el siguiente:

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

    for i in range(1, n + 1):
        for j in range(1, i + 1):
          if j == i:
            G[i][j] = min(x[i - 1], f[i - 1])
          else:
            G[i][j] = max(G[i - j]) + min(x[i - 1], f[j - 1])

    return max(G[n])

def matriz_soldados_elim(x, f):
    n = len(x)
    DP = [[0] * (n + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for j in range(0, i + 1):
          if j == 0:
            DP[i][j] = max(DP[i - 1])
          elif j == i:
            DP[i][j] = min(x[i - 1], f[i - 1])
          else:
            DP[i][j] = DP[i - j + 1][0] + min(x[i - 1], f[j - 1])
    return DP

def reconstruir_camino(x,f):
  G = matriz_soldados_elim(x, f)
  n = len(x)

  #Armo la lista llena de "Cargar" y pongo "Atacar" al final porque siempre se ataca al final
  estrategia = ["Cargar" for _ in range(n - 1)]
  estrategia.append("Atacar")
  i = n

  while( i > 0 ):
    j = G[i].index(max(G[i]))
    i = i - j
    estrategia[i - 1] = "Atacar"

  return estrategia

x = [271, 533, 916, 656, 664]
f = [21, 671, 749, 833, 1453]

print("Estrategia: ",reconstruir_camino(x,f))
```

4. Variabilidad de los valores de los carga y cantidad enemigos

En esta sección del informe, examinaremos la variabilidad de los datos. La variabilidad de los datos puede ejercer una influencia significativa en la complejidad temporal y espacial del algoritmo. Analizaremos cómo los distintos conjuntos de datos afectan la eficiencia del algoritmo. Para analizar la tendencia del algoritmo y como los valores de los datos afectan el mismo, probamos varios casos:

- $x_i > F(j)$: La cantidad de soldados es siempre es mayor que los valores de la función de carga.
- $F(j) > x_i$: El caso contrario al anterior, donde para la función de carga siempre retorna valores mayores a la cantidad de enemigos.
- Aleatorio: Generamos valores aleatorios cumpliendo la condición de que la función de carga es monótona creciente, como es requerido por el problema.

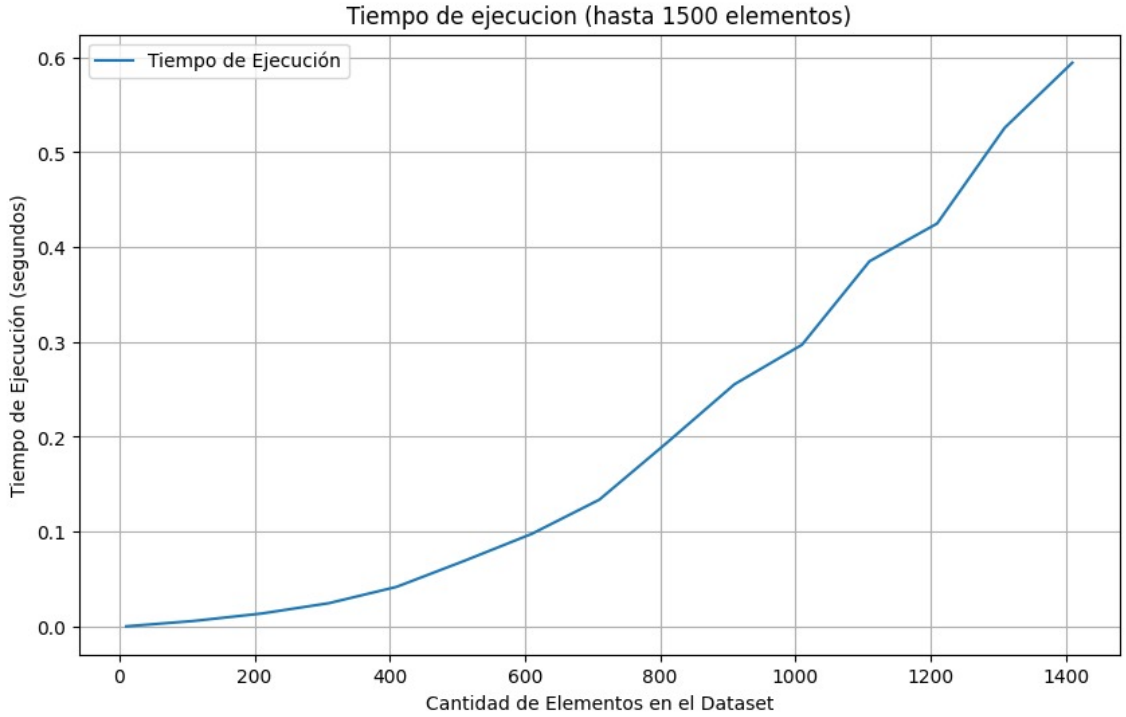


Grafico 1. Variación de Tiempo de Ejecución con Datasets de diferentes tamaños.

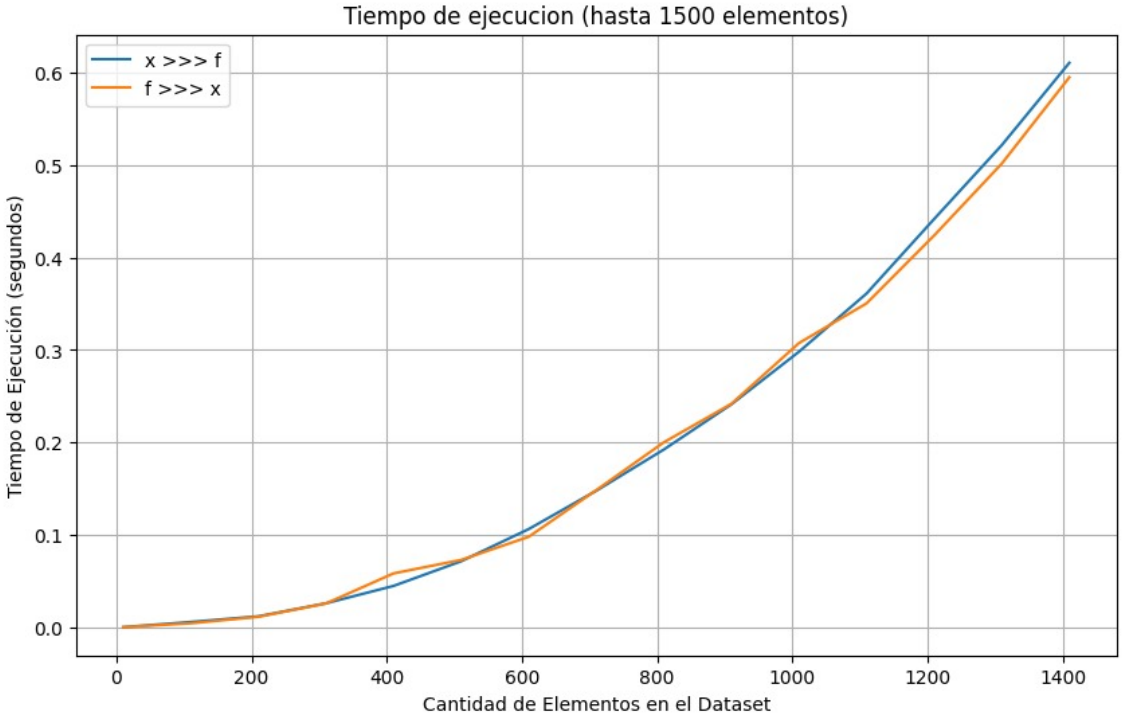


Grafico 2. Variación de Tiempo de Ejecución con Datasets de diferentes tamaños, comparando $x_i > F(j)$ y $F(j) > x_i$.

Después de analizar los datos y los gráficos, concluimos que la variabilidad de los datos no es importante para la complejidad temporal del algoritmo. Esto se debe a que en el fondo, lo que hace el algoritmo es explorar todo el espacio de soluciones de forma implícita, por lo tanto, termina probando todas las combinaciones posibles de los datos, y su valor no afecta. A continuación, dejamos los gráficos de complejidad con los distintos conjuntos.

5. Complejidad Temporal

En cuanto a la complejidad del algoritmo presentado, concluimos que este es $O(n^2)$. Esto se debe a que recorremos una matriz de n filas y n columnas, realizando operaciones de tiempo constante en el medio (acceder a cálculos ya realizados) y operaciones lineales (buscar el máximo del subproblema anterior). Si bien no se recorre en su totalidad, las constantes se desprecian. Por otro lado, el cálculo del máximo se realiza una única vez por fila ya que este resultado se guarda. Haciendo que se puedan acceder a los máximos de forma constante una vez obtenidos. Teniendo en cuenta todo esto, llegamos a una complejidad temporal de $O(n^2 + n + 1) => O(n^2)$.

Con respecto al algoritmo de reconstrucción, este tiene una complejidad también de $O(n^2)$ ya que, en el peor de los casos, deberemos recorrer toda la matriz en busca de la secuencia solicitada.

6. Conclusión

En conclusión, a raíz de lo desarrollado, pudimos observar como los algoritmos propuestos mediante Programación Dinámica pueden dar la solución óptima a un problema utilizando recálculos y explorando todo el espacio de soluciones de forma implícita. A través de diferentes enfoques, buscamos encontrar la ganancia máxima dado el conjunto de datos correspondiente. Aunque creemos que el algoritmo basado en la ecuación de recurrencia planteada es óptimo, reconocemos que podría existir algún otro enfoque al problema que también pueda proporcionar una solución óptima.

7. Anexo

En este apartado a forma de anexo realizaremos las correcciones pertinentes sobre el trabajo entregado.
Se constata que la sección de "Algoritmos Propuestos" debería ser "Ecuaciones de recurrencia propuestas" ya que se hace el análisis sobre ellas.
Por otro lado, luego de las correcciones pudimos analizar nuevamente la ecuación de recurrencia planteada en la sección 2.2, segundo algoritmo propuesto, ya que en inicialmente se planteaba que esta si llegaba al resultado esperado, pero que al intentar realizar ingeniería inversa sobre el algoritmo para hallar la secuencia de "ataque" y "recarga", nos resultaba sino imposible, extremadamente complejo. A partir de ello planteamos la siguiente:

$$G(i) = \max(G(j) + \min(x_i, F(i-1)\ ))\ 0 =< j < i$$

Donde evaluamos el óptimo en cada minuto posible de ataque, acumulando más o menos carga y viendo cómo eso repercute en la cantidad de soldados eliminados. También de esta forma se memorizan los óptimos anteriores para no tener que recalcularlos.
Luego de ello planteamos correctamente un algoritmo de reconstrucción que logra recuperar las estrategias óptimas a realizar en cada minuto de batalla sin incurrir en una complejidad mayor a $O(n^2)$.
Finalmente nuestro codigo es el siguiente:

```
def array_soldados_elim(x, f):
    n = len(x)
    DP = [0] * (n + 1)
    for i in range(1, n + 1):
        DP[i] = max(DP[j] + min(x[i - 1], f[i - j - 1]) for j in range(i))

    return DP

def max_soldados_elim(x, f):
    return array_soldados_elim(x,f)[-1]

def reconstruir_camino(x, f, DP):
    n = len(x)
    camino = []
    i = n  # Empezamos desde el final

    while i > 0:
        for j in range(i):
            if DP[i] == DP[j] + min(x[i - 1], f[i - j - 1]):
                camino.append(i - 1)  # Ajustar a  ndice  0
                i = j
                break

    camino.reverse()  # Porque construimos el camino desde el final

    # Ahora asociamos los indices conseguidos a las acciones correspondientes

    acciones = ["Cargar"] * n

    for i in camino:
        acciones[i] = "Atacar" # Si el indice est  en el camino, significa que en
    ese minuto se atac

    return acciones
```

7.1. (5) Complejidad Temporal

Ahora analizaremos la complejidad temporal del algoritmo presentado. Empezando con la función `array_soldados_elim`, su complejidad temporal se deriva de dos bucles anidados. El bucle externo itera desde 1 hasta n, donde n es la longitud de la lista de soldados. Dentro de este primer bucle, hay un segundo bucle implícito en la expresión que calcula el máximo valor, que también recorre desde 0 hasta i. Esto significa que, en el peor de los casos, el número total de operaciones es la suma de los primeros n números, lo cual corresponde a una complejidad temporal de $O(n^2)$. La operación de hallar el máximo valor en un rango también contribuye a esta complejidad, pero hace que supere $O(n^2)$.

La función `max_soldados_elim` llama a `array_soldados_elim` y luego toma el último valor de la lista resultante, ya que este valor coincide con el máximo valor resultante. Dado que `array_soldados_elim` tiene una complejidad de $O(n^2)$ y encontrar el último en una lista es una operación constante $O(1)$, entonces la complejidad total de ‘`max_soldados_elim`‘ está dominada por $O(n^2)$.

Por último, la función `reconstruir_camino` utiliza el array DP para reconstruir la secuencia de decisiones óptimas. Este algoritmo contiene un bucle while que itera hasta que se ha procesado toda la lista de soldados, lo que implica un máximo de n iteraciones. Dentro de este bucle while, hay un bucle for que, en el peor de los casos, puede realizar hasta i comparaciones en cada iteración del while. Por lo tanto, la complejidad total de este proceso también es $O(n^2)$. Operaciones adicionales como la inversión del array y la construcción de la lista de acciones son lineales, es decir,$O(n^2)$, pero nuevamente, no incrementan la complejidad.

En resumen, cada uno de los algoritmos,`array_soldados_elim` , `max_soldados_elim`, y `reconstruir_camino`, presenta una complejidad temporal de $O(n^2)$. Esta complejidad cuadrática se debe a los bucles anidados que recorren los índices y realizan cálculos para cada combinación posible de soldados y fuerzas.

7.2. Optimalidad

En el apartado de optimalidad, evaluaremos el algoritmo presentado a ver si efectivamente proporciona una solución óptima para la eliminación máxima de soldados dadas las condiciones específicas del problema.

En primer lugar, la función `array_soldados_elim` construye un array DP donde cada entrada DP[i] representa el valor máximo de soldados eliminados hasta el i-ésimo minuto. La clave de su optimalidad radica en la subestructura óptima y la superposición de subproblemas. La relación de recurrencia que utiliza asegura que cada subproblema se resuelva de manera óptima antes de usarse para resolver problemas más grandes. Además, al resolver cada subproblema una sola vez y almacenar su resultado en el array DP, se evitan cálculos redundantes y se asegura que cada subproblema contribuya de manera efectiva a la solución final. Por lo tanto, podemos afirmar que `array_soldados_elim` proporciona una solución óptima al problema.

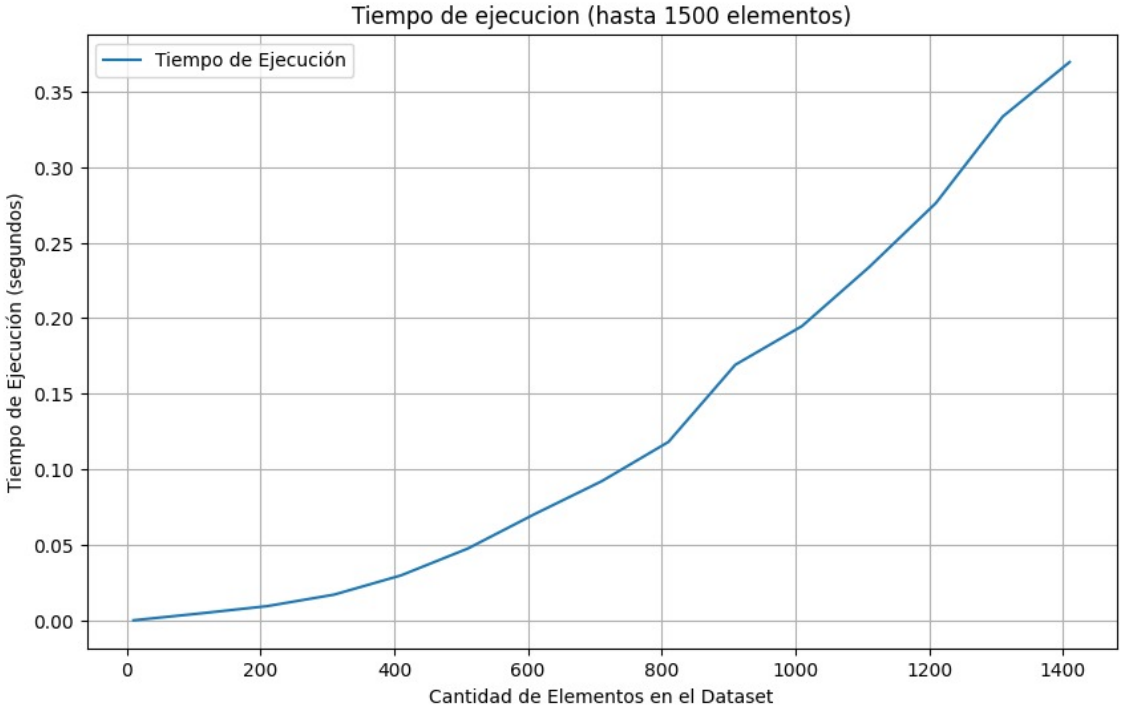
Por otro lado, la función `max_soldados_elim` simplemente llama a `array_soldados_elim` y toma el último valor del array DP generada. Dado que `array_soldados_elim` ya garantiza una solución óptima, `max_soldados_elim` hereda esta propiedad. La optimalidad de `max_soldados_elim` está directamente ligada a la optimalidad de `array_soldados_elim`. Al seleccionar el último valor del array DP, asegura que se obtenga la cantidad máxima de soldados eliminados. Por lo tanto, `max_soldados_elim` también proporciona una solución óptima.

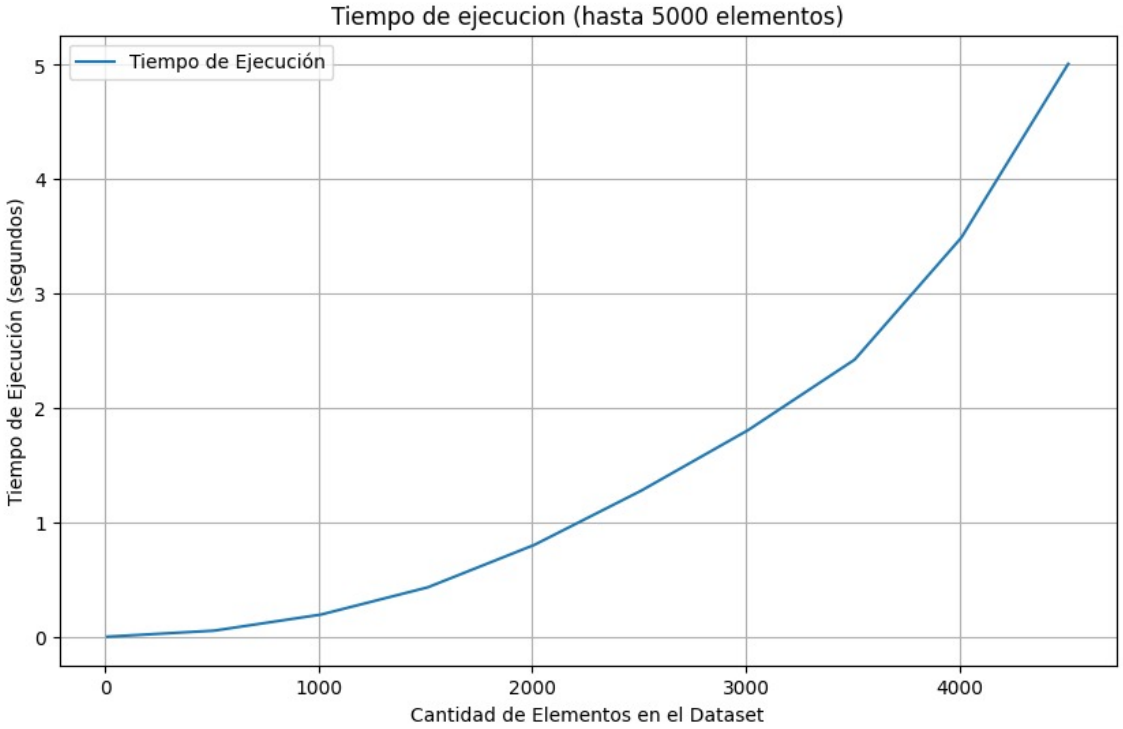
Por último, `reconstruir_camino` se encarga de reconstruir el camino de decisiones basado en el array DP previamente calculado. Utiliza un enfoque de retroceso para determinar en qué momentos específicos se tomó la decisión de .ªtacar". La reconstrucción del camino se basa en los valores óptimos almacenados en DP. Dado que DP contiene soluciones óptimas para cada subproblema, la reconstrucción del camino también será óptima. El algoritmo revisa cada decisión posible comparando valores en DP, asegurando que cada paso en el camino reconstruido corresponde a una decisión que maximiza la eliminación de soldados en ese punto específico. Al basarse en el array DP, que contiene soluciones óptimas, y al reconstruir el camino usando estas decisiones, `reconstruir_camino` también proporciona una solución óptima.

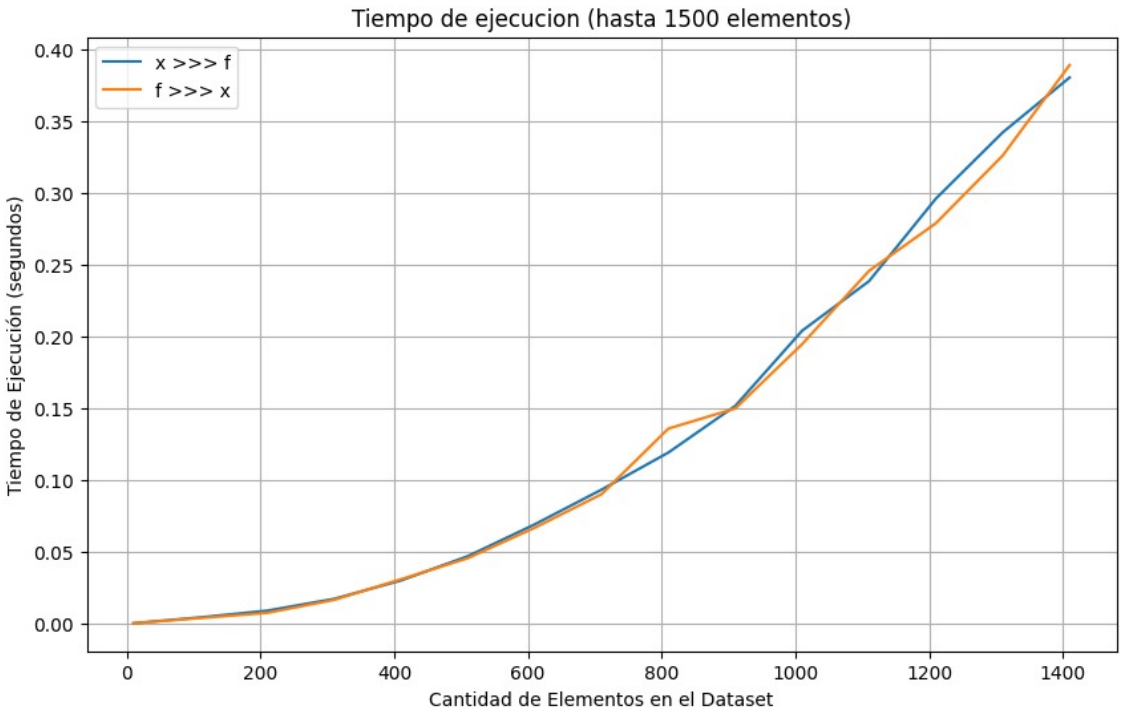
En conclusión ambos algoritmos, tanto el que utiliza la ecuación de recurrencia como el de reconstrucción, terminan por ser óptimos. Utilizando programación dinámica para descomponer el problema en subproblemas manejables y asegurar que cada subproblema se resuelva de manera óptima antes de combinarse para formar la solución final.

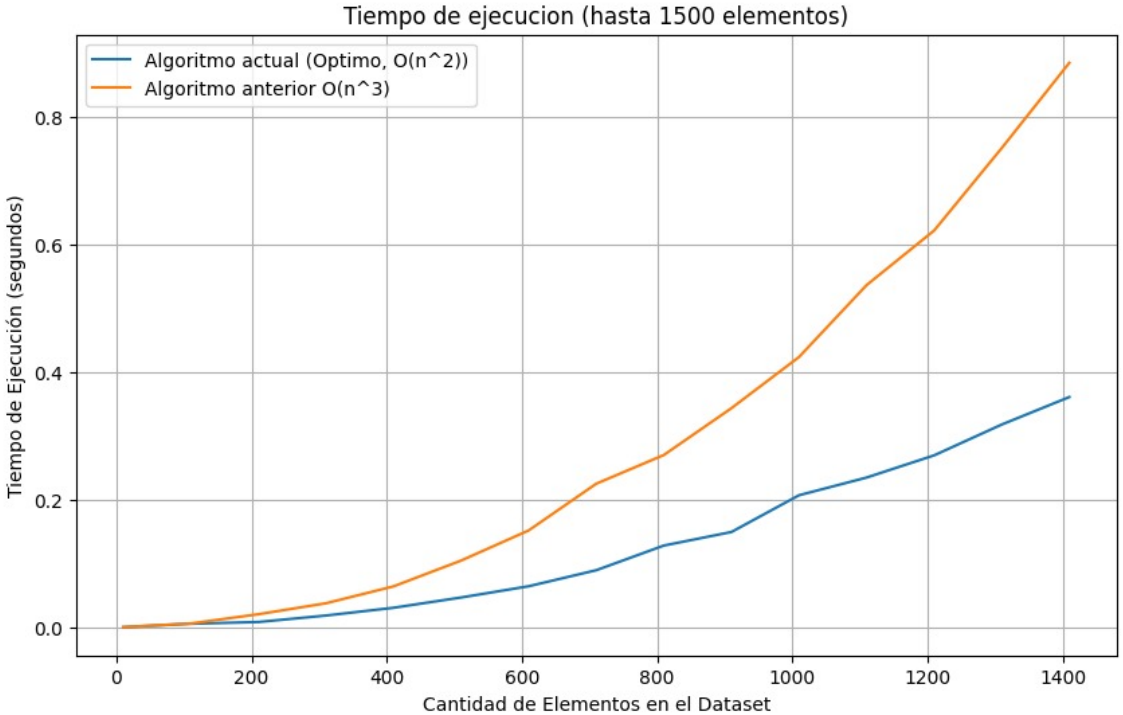
7.3. Mediciones

En esta seccion vamos a ver diferentes pruebas con el algoritmo propuesto. Se generaron de forma aleatoria sets de datos de tamaño creciente, primero hasta un maximo de 1500 elementos y luego hasta un maximo de 5000 elementos para poder probar la complejidad teorica. Tambien lo vamos a comparar con el algoritmo que habiamos propuesto como optimo anteriormente, el cual tenia una complejidad de O(n3) para poder ver la diferencia en los tiempos de ejecucion:









Para poder ver y reproducir los resultados mostrados arriba se puede utilizar el siguiente Notebook de google colab.