FACULTAD DE INGENIERIA
Universidad de Buenos Aires

TP2: Programación Dinámica para el Reino Tierra

[75.29] Teoría de Algoritmos
Primer Cuatrimestre de 2024

Integrantes

Alumno	Padrón
FUENTES, Azul	102184
GALIÁN, Tomás Ezequiel	104354
SANTANDER, Valentín	105637

Índice

1. Análisis del problema

. Para el problema planteado se cuenta con dos listas de tamaño $n \in \mathbf{N}$ tal que $n > 0$. Una lista con las ráfagas de soldados del ataque de la Nación del Fuego (denotada como X) y una lista de cuantos enemigos pueden eliminar los Dai Li dependiendo de cuantos minutos pasen desde el último ataque (denotada como E). En este contexto, n se refiere a la cantidad de minutos que dura el ataque.

X - Lista de Ráfagas de Soldados del Ataque de la Nación del Fuego

La lista de ráfagas de soldados del ataque o bien X consiste en la cantidad de soldados de la Nación del Fuego que llegarán a atacar la ciudad de Ba Sing Se en el minuto i tal que $1 \leq i \leq n$. Se denotará a la cantidad de soldados que llegan en el i-ésimo minuto como x_i y ese valor estará en la posición i de la lista X. Se asume que no hay soldados enemigos al inicio del ataque.

Para una lista de ráfagas X

$$X = [x_1, x_2, \cdots, x_n]$$

Se asume que:

$$x_i \in \mathbb{N}, \forall i : 1 \leq i \leq n, x_1 > 0$$

E - Lista de Carga de Energía de los Dai Li

La lista de carga de energía de los Dai Li o bien E, consiste en los puntos de una función $f(j)$ con dominio $D = \{j \in \mathbb{N} : 1 \leq j \leq n\}$. Esta función es monótona creciente y representa la cantidad de soldados de la Nación del Fuego que pueden eliminar los Dai Li habiendo recargado j minutos. A medida que pasan los minutos, los soldados pueden eliminar a más soldados, pero una vez que atacaron, tienen que empezar a recargar desde el principio. Se denotará como e_j a la cantidad de soldados que pueden eliminar los Dai Li habiendo esperado j minutos y ese valor estará en posición j de la lista E. Se asume que los Dai Li tienen carga nula al inicio del ataque.

Para una lista de carga de energía E:

$$E = [f(1), f(2), \cdots, f(n)] = [e_1, e_2, \cdots, e_n]$$

Se asume que:

$$e_j \in \mathbb{N}, \forall j : 1 \leq i \leq n, e_j > 0$$

Cabe destacar que el tamaño de E y X es el mismo para que se pueda permitir el caso en el que los Dai Li carguen en todos los minutos excepto el último, desatando su poder máximo en ese ataque particular.

1.1. Definición de Estrategia

Teniendo ambas listas, se debe elegir una estrategia a seguir para cada minuto del ataque definido en X. Para cada minuto, hay dos opciones: *Atacar* o *Cargar*.

Se define entonces una estrategia S para un par de listas X y E de tamaño n como una lista que contiene para cada minuto una acción: *Atacar* o *Cargar*. En la i-ésima posición se indica que se debe hacer en el i-ésimo minuto, denotándolo como s_i.

$$S = [s_1, s_2, \cdots, s_n] = [\text{Atacar}, \text{Cargar}, \cdots, \text{Atacar}]$$

Se permite utilizar cualquier combinación de acciones de tamaño n. Ahora se explicará a detalle que implica *Atacar* o *Cargar*.

Atacar

Si se decide *Atacar* en el minuto i, se eliminará una cierta cantidad de soldados. El ataque se realizará con respecto a la cantidad de soldados indicada por x_i. Ante esto, hay dos casos posibles:

- Caso $e_j \geq x_i$

  Significa que los Dai Li cargaron suficiente energía como para eliminar a todos o incluso a más soldados que los que hay en el minuto i. En este caso, se eliminan solamente a los x_i soldados de ese minuto. Se suma la cantidad x_i a la cantidad total de soldados eliminados en el ataque indicado por la lista X.

- Caso $e_j < x_i$

  Significa que los Dai Li no cargaron suficiente energía para eliminar a todos los soldados del minuto i. Pero pueden eliminar a e_j soldados de los x_i, dejando pasar a los restantes.

En otras palabras, al atacar en el minuto i se suma a la cantidad total de soldados eliminados el mínimo valor entre e_j y x_i siendo j la cantidad de minutos que pasaron desde el último ataque.

Cargar

Si se decide *Cargar* en el minuto i, no se sumará nada a la cantidad total de soldados eliminados y los x_i soldados lograrán entrar en la ciudad sin bajas. Pero a cambio, se aumentará el valor de j en uno pues pasó un minuto sin que los Dai Li ataquen. Se sabe que se podrán eliminar más soldados en el siguiente minuto porque la función $f(j)$ es monótona creciente por definición.

1.2. Cálculo de Cantidad Total de Soldados eliminados

Dadas las listas X y E utilizando una estrategia S, se calcula la cantidad total de soldados eliminados como

$$\sum_{1}^{n} g(s_i) \cdot \text{mín}(e_j, x_i)$$

Donde:

- g(s) es una función con dominio $T = \{\text{Atacar}, \text{Cargar}\}$ definida como:

$$g(s) = \begin{cases} 0 & \text{si } s = \text{Cargar} \\ 1 & \text{si } s = \text{Atacar} \end{cases}$$

- e_j es el valor en la posición j de la lista E siendo j la cantidad de minutos que pasaron desde el último ataque
- x_i la cantidad de soldados que llegarán a la ciudad en el minuto i

Por ejemplo, dadas las listas X y E usando la estrategia S

$$X = [4, 3]$$
$$E = [2, 3]$$
$$S = [\text{Cargar}, \text{Atacar}]$$

La cantidad total de soldados eliminados sería:

$$\sum_{1}^{2} g(s_i) \cdot \text{mín}(e_j, x_i) = g(s_1) \cdot \text{mín}(e_1, x_1) + g(s_2) \cdot \text{mín}(e_2, x_2) = 0 \cdot \text{mín}(2, 4) + 1 \cdot \text{mín}(3, 3) = 0 + 3 = 3$$

1.3. Decisión de Estrategia óptima

Entendiendo las variables a considerar, el problema radica en diseñar una manera de elegir una estrategia que permita eliminar la mayor cantidad total de soldados posible sabiendo cuando atacar y cuando cargar. Es válido atacar en cada minuto, pero puede ocurrir que habiendo cargado más energía, se eliminen más en un minuto siguiente. Pero si se espera demasiado, puede ocurrir que hubiera sido preferible eliminar menos soldados en más minutos que eliminar muchos soldados en un determinado minuto donde se cargó más energía.

Para ilustrar estas situaciones, se proponen los siguientes ejemplos.

1.3.1. Ejemplo con Estrategia agresiva óptima

Dadas las listas X_A ,E_A y S_A

$$X_A = [5, 10]$$
$$E_A = [3, 4]$$
$$S_A = [\text{Cargar}, \text{Atacar}]$$

La cantidad total de soldados eliminados se calcula como:

$$\sum_{1}^{2} g(s_i) \cdot \text{mín}(e_j, x_i) = 0 \cdot \text{mín}(3, 5) + 1 \cdot \text{mín}(4, 10) = 0 + 4 = 4$$

Pero si se hubiera usado una estrategia más agresiva como $S_B = [\text{Atacar}, \text{Atacar}]$, para las mismas listas X_A y E_A, el resultado sería:

$$\sum_{1}^{2} g(s_i) \cdot \text{mín}(e_j, x_i) = 1 \cdot \text{mín}(3, 5) + 1 \cdot \text{mín}(3, 10) = 3 + 3 = 6$$

En este caso conviene atacar más y cargar menos.

1.3.2. Ejemplo con Estrategia conservadora óptima

Dadas las listas X_A ,E_A y S_A

$$X_A = [5, 10]$$
$$E_A = [3, 10]$$
$$S_A = [\text{Atacar}, \text{Atacar}]$$

La cantidad total de soldados eliminados se calcula como:

$$\sum_{1}^{2} g(s_i) \cdot \text{mín}(e_j, x_i) = 1 \cdot \text{mín}(3, 5) + 1 \cdot \text{mín}(3, 10) = 3 + 3 = 6$$

Pero si se hubiera usado una estrategia más conservadora como $S_B = [\text{Cargar}, \text{Atacar}]$, para las mismas listas X_1 y E_1, el resultado sería:

$$\sum_{1}^{2} g(s_i) \cdot \text{mín}(e_j, x_i) = 0 \cdot \text{mín}(3, 5) + 1 \cdot \text{mín}(10, 10) = 0 + 10 = 10$$

En este caso conviene más cargar y eliminar más soldados después.

1.4. Estrategia óptima usando Programación Dinámica

Como se pudo apreciar en los ejemplos anteriores, no es posible establecer un criterio claro para determinar la estrategia óptima para eliminar la mayor cantidad de soldados.

En ejemplos sencillos de pocos minutos, se puede ver a simple vista cual es la solución óptima. Pero a medida que el problema escala, las combinaciones aumentan exponencialmente y es difícil saber que es lo que más conviene.

Debido a la naturaleza combinatoria del problema, una manera de encararlo es utilizar el enfoque de programación dinámica.

1.4.1. ¿Qué es Programación Dinámica?

La programación dinámica es un método de optimización utilizado para resolver problemas computacionales complejos dividiéndolos en subproblemas más simples y resolviendo cada subproblema solo una vez. Luego, se almacenan las soluciones de los subproblemas para evitar recálculos innecesarios cuando se encuentran en el camino hacia la solución final del problema.

Una vez que se entiende la forma de los subproblemas y como estos se componen en subproblemas más grandes, se podrá definir una ecuación de recurrencia que permita la solución óptima sin importar la cantidad de minutos.

Para facilitar la construcción de la ecuación de recurrencia se utilizará un enfoque ascendente (Bottom-Up) donde se determinarán las soluciones para una cantidad de minutos actual y luego para su siguiente, la siguiente a esa y así siguiendo. Además, se guardarán los resultados óptimos de cada paso para evitar recalcular mediciones ya obtenidas mejorando la eficiencia de la solución. Esta técnica se conoce como *Memorización*.

1.4.2. Construcción y composición de los subproblemas

En un ataque de la Nación del Fuego de tamaño n con sus listas X y E, los subproblemas se obtienen a partir de subconjuntos ordenados de cada lista. Un subconjunto ordenado se refiere a una cantidad i de ráfagas desde la primera hasta la i-ésima con $i \in \mathbb{N} : 1 \leq i \leq n$. Primero se analiza la solución óptima de tamaño uno, luego la de tamaño dos y así hasta llegar a n.

Siguiendo esta idea, se define una función $\text{OPT}(i)$ con dominio $H = \{i \in \mathbb{N} : 1 \leq i \leq n\}$. El valor de $\text{OPT}(i)$ representa la cantidad máxima de soldados que se pueden eliminar hasta el minuto i. A su vez se define la lista OPT que contiene todos los valores de la función de cantidades máximas. En la i-ésima posición se encuentra el valor $\text{OPT}(i)$.

$$\text{OPT} = [\text{OPT}(1), \text{OPT}(2), \cdots, \text{OPT}(n)]$$

Para un ataque de tamaño n se irán calculando los subproblemas utilizando la notación X_i y E_i con $i \in \mathbb{N} : 1 \leq i \leq n$ indicando que es el subproblema de tamaño i para el que se utilizará la estrategia óptima S_i. A cada subproblema le corresponde un valor óptimo $\text{OPT}(i) = O_i$

$$\begin{array}{l} X_1 = [x_1] \quad E_1 = [e_1], S_1 = [s_1] \quad \text{OPT}(1) = O_1 \\ X_2 = [x_1, x_2] \quad E_2 = [e_1, e_2], S_2 = [s_1, s_2] \quad \text{OPT}(2) = O_2 \\ \qquad\qquad\qquad\qquad \cdots \\ X_n = [x_1, x_2, \cdots, x_n] \quad E_2 = [e_1, e_2, \cdots, e_n], S_2 = [s_1, s_2, \cdots, s_n] \quad \text{OPT}(n) = O_n \end{array}$$

Una vez definida la forma de los subproblemas, se debe determinar como se componen en problemas más complejos. Para ello, se realizará un análisis Bottom-Up referido anteriormente.

Ataque de una sola ráfaga

En un ataque de una sola ráfaga, la respuesta óptima es sencilla, siempre conviene atacar. Esto es así porque de lo contrario, la suma sería nula y sea cual sea el valor e_1 y x_1, el mínimo entre ambos será mayor a 0. Por lo tanto, El $\text{OPT}(1)$ es $\text{mín}(e_1, x_1)$.

Con esta idea, se puede inferir que en un ataque de cualquier tamaño, los Dai Li atacarán al menos una vez pues ese valor será mayor a cero. Cargar en todos los turnos no puede nunca ser una solución óptima a menos que todas las ráfagas sean nulas, pero se asume que en cada minuto llega al menos un soldado.

Ataque de dos ráfagas

En un ataque de dos ráfagas, como se vió en los ejemplos anteriores, la mejor decisión es el máximo valor entre:

- Usar únicamente el mínimo entre una carga de dos y la cantidad de soldados del segundo minuto (Lo que se llamó Estrategia conservadora)
- Usar la cantidad de soldados que se elimarían atacando en el primer minuto (en otras palabras la solución óptima en el ataque de tamaño uno) y sumarle el mínimo entre la carga de uno y la cantidad de soldados del segundo minuto (lo que se llamó Estrategia agresiva)

No se puede saber a priori cual es mayor por lo que es necesario calcular ambos escenarios. El que sea mayor, se guardará como óptimo en la lista OPT y será el valor $\text{OPT}(2)$.

Ataque de tres ráfagas

En un ataque de tres ráfagas, las combinaciones aumentan. Sin embargo, conociendo las mejores soluciones de los tamaños anteriores se puede evitar tener que probar todas.

Si ya se conoce la mejor solución en un ataque de tamaño uno y dos, la mejor solución para la de tamaño tres será el valor máximo entre:

- Usar únicamente el mínimo entre una carga de tres y la cantidad de soldados del tercer minuto
- Usar la solución óptima en el ataque de tamaño uno y sumarle el mínimo entre una carga de dos y la cantidad de soldados del tercer minuto
- Usar la solución óptima en el ataque de tamaño dos y sumarle el mínimo entre una carga de uno y la cantidad de soldados del tercer minuto

Esto es así porque ya se ha calculado anteriormente que son las máximas al guardarlo en la lista OPT. Aqui es donde entra en juego la *Memorización*. Sin embargo habrá que calcular y comparar para todos los tamaños pues no se puede saber cual de todos es el mejor. El que sea mayor, se guardará como óptimo en la lista OPT y será el valor $\text{OPT}(3)$.

Ataque de n ráfagas

Se puede notar un cierto patrón al momento de obtener un valor óptimo a partir de otros. En el paso i tal que $i \in \mathbb{N} : 1 \leq i \leq n$ se deberá comparar las sumas de los óptimos del paso j tal que $j \in \mathbb{N} : 1 \leq j \leq i$ y la cantidad de soldados que se podrían eliminar esperando $i - j$ minutos.

Pero este patrón no contempla el caso en el que se usa únicamente el mínimo entre una carga de i y la cantidad de soldados del i-ésimo minuto que puede dar un resultado válido. Para poder contemplar este caso, es necesario modificar la definición de la función $\text{OPT}(i)$. Hay que agregar el valor $\text{OPT}(0) = 0$. Este valor implica que antes de que llegue la primer ráfaga de soldados, el valor óptimo es cero pues todavía no se eliminó ninguno.

La lista OPT queda entonces:

$$\text{OPT} = [\text{OPT}(0), \text{OPT}(1), \cdots, \text{OPT}(n)] = [0, \cdots]$$

Con esto en cuenta el patrón indica que en el paso i tal que $i \in \mathbb{N} : 1 \leq i \leq n$ se deberá comparar las sumas de los óptimos del paso k tal que $k \in \mathbb{Z} : 0 \leq k \leq i$ y la cantidad de soldados que se podrían eliminar esperando $i - k$ minutos.

1.4.3. Ecuación de Recurrencia

Una observación que se puede hacer en este punto es que siempre se ataca en el último minuto considerado. Esto ocurre porque aunque que se haya atacado por última vez en el minuto anterior (Provocando que los Dai Li ataquen con su ataque menos poderoso) ese valor siempre hará que la cantidad total de soldados eliminados sea mayor. Por lo tanto para cualquier n, se sabe que la Estrategia óptima siempre elige Atacar en el último minuto. Por lo tanto, paso a paso se busca la mejor solución hasta el momento sabiendo que se atacará en el último minuto necesariamente.

Finalmente, después de analizar el problema se puede escribir la ecuación de recurrencia como:

$$\boxed{\text{OPT}(i) = \max_{0 \leq k \leq i} (\text{OPT}(k) + \text{mín}(e_{i-k}, x_i))} \tag{1}$$

1.4.4. Obtención de Estrategia óptima

El último valor de la lista de óptimos tiene la cantidad máxima de enemigos que se puede eliminar en el ataque completo de tamaño n. Pero no se sabe cuáles fueron los pasos a seguir (cuando se atacó y cuando se cargó) para obtener ese resultado. Sin embargo se puede inferir la estrategia óptima S usando X, E y los valores óptimos guardados en OPT.

Para obtener S se comienza con el último paso (o bien el n-ésimo), y se recorre la lista OPT de manera descendente hasta llegar al paso k que cumpla:

$$\boxed{\text{OPT}(i) - \text{OPT}(k) = \text{mín}(e_{i-k}, x_i)} \tag{2}$$

Hasta que esta condición no se cumpla, todos los pasos intermedios entre i y k serán *Cargar*, ya que en k se encontró el paso donde corresponde *Atacar*, en el cual la diferencia entre los valores óptimos coincide con el mínimo entre la función de carga de energía en la posición $i - k$ y la cantidad de enemigos que llegan en el i-ésimo minuto. Se realiza iterativamente este procedimiento, tomando siempre la última posición de *Atacar* encontrada como referencia, hasta obtener la secuencia completa. En otras palabras, una vez que los valores coinciden se sigue el algoritmo desde la k-ésima posición ya que se sabe que fue la manera de llegar al OPT(i). Cuando $k = 0$, se termina el algoritmo.

Es posible que se de el caso de que coincidan los valores en un paso k_1 y en otro paso k_2 siendo que $k_1 > k_2$. Por lo tanto solo se seguiría el camino dado por k_1 dejando de lado una solución alternativa donde los Dai Li cargaron energía desde k_2. Sin embargo, como ambas soluciones cumplen con la ecuación, ambas deberían dar el mismo resultado final pero con estrategias diferentes. Se toma por defecto el mayor valor de k, donde se produce la primer coincidencia.

2. Algoritmos propuestos

2.1. Obtener valores óptimos ataque

Para la implementación de la solución en código se utilizó el lenguage de programación Python. El algoritmo propuesto es el siguiente:

```
3  def obtener_valores_optimos_ataque(x, e):
4
5      if len(x) != len(e):
6          raise ListasIncompatiblesError()
7
8      n = len(x)
9
10     opt = [0] * (n+1)
11     x = [0] + x
12     e = [0] + e
```

```
13
14      for i in range(1,n+1):
15          opt_i = 0
16          for k in range(0,i):
17              opt_parcial = opt[k] + min((e[(i-k)],x[(i)]))
18              if opt_parcial > opt_i:
19                  opt_i = opt_parcial
20                  opt[i] = opt_i
21
22      return opt
```

2.1.1. Explicación en detalle: Obtener valores óptimos ataque

Primero se realiza un chequeo de compatibilidad de los datos para proveer mayor robustez y detección de errores. Se verifica que X y E sean del mismo tamaño ya que de lo contrario no tendría sentido calcular el problema. Es una precondición del algoritmo, de lo contrario no se comportaría de la manera esperada.

En caso de que los datos ingresados no sean compatibles, se lanza una excepción indicando el problema la usuario.

```
5       if len(x) != len(e):
6           raise ListasIncompatiblesError()
```

Una vez que se sabe que el tamaño de las listas es compatible, se guarda el tamaño del problema en la variable `n`. Además se inicializa el arreglo de óptimos que tiene tamaño $n + 1$. Se coloca el valor 0 en cada posición. Sería un valor óptimo para cada minuto más el valor óptimo en 0 cuyo valor ya se sabe que es 0 como se vió en la descripción del problema.

Además de eso se agrega un valor extra al inicio de las listas X y E. A priori esto no es necesario, pero facilita la legibilidad del algoritmo ya que permite usar los mismos índices en las listas de Python que los de la ecuación de recurrencia. Se decidió poner un 0 ya que es el candidato más probable como valor nulo en una lista de enteros. Pero nada impide que se coloque cualquier otro objeto mientras ocupe la primer posición de la lista, que en Python tiene índice 0.

```
8       n = len(x)
9
10      opt = [0] * (n+1)
11      x = [0] + x
12      e = [0] + e
```

Una vez inicializado todo, se procede a calcular los valores óptimos. Primero se inicia un ciclo `for` desde 1 hasta n. Esto representa la búsqueda del valor óptimo para cada minuto de la lista X. Como ya se discutió anteriormente, se busca ir construyendo la soluciones parciales minuto a minuto hasta llegar a n donde se obtiene el resultado final

Se inicializa el i-ésimo valor óptimo en cero y luego se calculan todas las combinaciones con los óptimos anteriores siguiendo la ecuación de recurrencia. Se debe comparar el resultado con todos los anteriores para asegurar que se obtuvo el valor óptimo.

Esta verificación se hace a medida que se calculan los valores. Cuando se obtiene uno mejor, se actualiza el valor y se guarda en el arreglo en la posición correspondiente.

Una vez terminados los ciclos para cada valor de i, se devuelve la lista de óptimos de tamaño $n + 1$.

```
14      for i in range(1,n+1):
15          opt_i = 0
16          for k in range(0,i):
17              opt_parcial = opt[k] + min((e[(i-k)],x[(i)]))
```

```
18              if opt_parcial > opt_i:
19                  opt_i = opt_parcial
20                  opt[i] = opt_i
21
22      return opt
```

2.1.2. Complejidad algorítmica: Obtener valores óptimos ataque

Los chequeos de largo tienen complejidad $O(n)$ siendo n la cantidad de minutos del ataque, pues para saber cuantos hay se deben contar todos. Sin embargo, dependiendo de la implementación de la lista puede llegar a ser de complejdad constante, si es que se guarda el valor a medida que se colocan elementos en la lista. De todos modos utilizar una cota en $O(n)$ es suficiente.

La inicialización de la lista de valores óptimos e insertar un valor al inicio de las listas `x` y `e` tienen complejidad $O(n+1)$ y $O(n)$ respectivamente.

La inicialización porque se debe generar un arreglo de tamaño tamaño $n+1$ colocando cada elemento en su respectiva posición (en este caso todos ceros). Al ser el uno un valor constante, la complejidad resulta en $O(n)$

La inserción también tiene complejidad $O(n)$ pues al colocar un nuevo elemento al inicio, se deben mover de posición todos los elementos restantes.

Ahora bien, la parte más compleja del algoritmo se da en el doble bucle `for` donde se calculan y actualizan los valores óptimos. Para cada valor de i entre 1 y n, se itera desde 0 hasta el respectivo $i-1$. Eso significa que en para $i=1$ se itera una vez, para $i=2$ se itera dos veces y así siguiendo. Todo lo que se realice dentro de cada llamada es de tiempo constante, por lo tanto la complejidad de esta sección está dada por la cantidad de total de llamadas. Esta se puede expresar utilizando la siguiente sumatoria L:

$$L = \sum_{i=1}^{n} n = 1 + 2 + \cdots + (n-1) + n$$

A través de la siguiente demostración, se puede ver el resultado de la suma para cualquier valor de n. Con ese resultado se podrá verificar correctamente la complejidad del algoritmo.

$$L = 1 + 2 + \cdots + (n-1) + n \tag{3}$$
$$L = n + (n-1) + \cdots + 2 + 1 \tag{4}$$

Sumando 3 y 4 miembro a miembro obtengo:

$$L + L = (1+n) + (2+(n-1)) + \cdots + ((n-1)+2) + (n+1) \tag{5}$$
$$2L = (n+1) + (n+1) + \cdots + (n+1) + (n+1) \tag{6}$$

Como la sumatoria va desde 1 hasta n, hay n términos en la sumatoria. Por lo tanto:

$$2L = n \cdot (n+1) \tag{7}$$

Entonces la sumatoria resulta en:

$$L = \frac{n \cdot (n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} \tag{8}$$

Sabiendo que L reprensenta la cantidad de llamadas, se puede apreciar que esta parte del algoritmo muestra una naturaleza cuadrática. Por lo tanto la complejidad del doble bucle `for` es $O(n^2)$.

Dado que esta complejidad es el valor más significativo, el algoritmo que obtiene los valores óptimos de un ataque de n minutos es $O(n^2)$.

2.2. Obtener estrategia óptima

Ahora bien, para la reconstrucción de la estrategia óptima, el algoritmo propuesto es el siguiente:

```
24 def obtener_estrategia_optima_ataque(opt, x, e):
25     if len(x) != len(e):
26         raise ListasIncompatiblesError()
27
28     n = len(x)
29     if n == 0:
30         return []
31
32     x = [0] + x
33     e = [0] + e
34
35     s = []
36     s = [CARGAR] * (n+1)
37     s[-1] = ATACAR
38     tope = n
39
40     for i in range(n - 1,0,-1):
41         dif_pos = tope - i
42         dif_opt = opt[tope] - opt[i]
43         e_dif = e[dif_pos]
44
45         if dif_opt == min(e_dif, x[tope]) :
46             s[i] = ATACAR
47             tope = i
48         else:
49             s[i] = CARGAR
50
51     if opt[-1] != calcular_cantidad_enemigos_eliminados(x[1:], e[1:], s[1:]):
52         raise EstrategiaInoptimaError()
53
54     return s[1:]
```

2.2.1. Explicación en detalle: Obtener estrategia óptima ataque

Primero se realiza el mismo chequeo de compatibilidad explicado en 2.1.1.

```
26     if len(x) != len(e):
27         raise ListasIncompatiblesError()
```

Luego se calcula el largo de la lista y se agrega un cero al inicio de `x` y `e` por las mismas razones que se explicaron en 2.1.1.

Pero además, se realiza un chequeo de un caso borde cuando se recibe una lista de ataques vacía. La estrategia óptima en ese caso sería no hacer nada devolviendo una lista vacía.

Además se inicializa el arreglo donde se construirá la estrategia óptima (`s`). Como se discutió antes, para que la estrategia sea óptima se debe atacar al menos una vez y siempre en el último minuto. Por lo tanto, conviene colocar en un principio la acción `ATACAR`.

Además de eso, se inicializa la variable `tope` con valor n. Esta representa la posición del último ataque que se sabe que fue realizado en la estrategia óptima. Como siempre se ataca en el último minuto, su valor inicial es n.

```
29      n = len(x)
30
31      if n == 0:
32          return []
33
34      x = [0] + x
35      e = [0] + e
36      s = []
37      s = [CARGAR] * (n+1)
38      s[-1] = ATACAR
39      tope = n
```

Luego se realiza la construcción de la estrategia haciendo un ciclo similar al que se hizo en el algoritmo anterior, pero iterando desde el final hacia el principio. De esta manera, es más fácil operar pues estando parado en una posición i del arreglo de óptimos, se sabe que todas las posiciones anteriores también lo son y fueron consideradas para el cálculo del óptimo en i.

Para verificar si en un determinado paso i se realizó un *Ataque*, se comprueba que la diferencia entre los óptimos desde el último *Ataque* "*tope*"hasta i es igual al mínimo entre la función de carga de energía en la posición $tope - i$ y la cantidad de enemigos que llegan en el minuto del *tope*, luego se actualiza el valor del *tope*. En caso contrario se considera que en el paso i se realizó una *Carga*, ya que la fuerza acumulada que se posee en el paso i no es suficiente para derribar a los enemigos.

```
29      for i in range(n - 1,0,-1):
30          dif_pos = tope - i
31          dif_opt = opt[tope] - opt[i]
32          e_dif = e[dif_pos]
33
34          if dif_opt == min(e_dif, x[tope]) :
35              s[i] = ATACAR
36              tope = i
37          else:
38              s[i] = CARGAR
```

Casi terminando, se realiza un chequeo final donde se verifica que la suma generada por la estrategia óptima obtenida es igual al úlitmo valor de la lista de óptimos. Si ocurre algún error en la construcción, se lanza una excepción. La implementación de esta función auxiliar se basa en lo discutido en la sección 1.2. Finalmente se devuelve la estrategia verificando que cumple con la condición de optimalidad.

El algoritmo auxiliar que calcula la cantidad de enemigos eliminados es:

```
71  def g(s):
72      return 1 if s == ATACAR else 0
73
74  def calcular_cantidad_enemigos_eliminados(x, e, s):
75
76      if len(x) != len(e) or len(x) != len(s) or len(e) != len(s):
77          raise ListasIncompatiblesError()
78
79      n = len(x)
80      cantidad_total_enemigos_eliminados = 0
81      j = 0
82      g_s = list(map(g, s))
83
```

```
84      for i in range(n):
85          cantidad_total_enemigos_eliminados += g_s[i] * min(x[i], e[j])
86
87          if g_s[i] < 1:
88              j += 1
89          else:
90              j = 0
91      return cantidad_total_enemigos_eliminados
```

Primero chequea que los datos sean correctos y se inicializa:

- El valor de n como el largo de la lista X
- La cantidad total de enemigos en cero
- j en cero, que es la cantidad de minutos que pasaron desde el último ataque
- g_s, que es la lista S parseada con la función $g(s)$

Luego va sumando el aporte de cada minuto ajustando el valor de j correspondiente en cada paso y devuelve la cantidad total de enemigos eliminados.

2.2.2. Complejidad algorítmica: Obtener estrategia óptima ataque

Para un ataque de n minutos, las verificaciones e inicializaciones tienen complejidad $O(n)$ de manera análoga al algoritmo de cálculo de valores óptimos. Tanto para construir la estrategia óptima como para calcular la cantidad total de enemigos eliminados, se debe iterar cada minuto una sola vez. Para la primera, se itera de forma descentente y para la segunda de forma ascendente. Ambas secciones tienen entonces una complejidad de $O(n)$.

Por lo tanto, la complejidad final de este algoritmo es $O(n)$ siendo n la cantidad de minutos que dura el ataque.

3. Variabilidad de Valores

Con respecto a la variablidad de los valores, al tratarse de un algoritmo de programación dinámica, se divide el problema en subproblemas y se utilizan esos resultados para encontrar la solución óptima, evitando recálculos. Esto significa que, una vez que los subproblemas se han calculado y almacenado, el proceso de encontrar la solución óptima no se ve afectado por la variabilidad de los datos de entrada.

Dado que la programación dinámica resuelve los subproblemas y almacena sus resultados para evitar recalcularlos, la variabilidad en los valores de llegadas y recargas no debería afectar la optimalidad del algoritmo en sí mismo. El algoritmo seguirá siendo óptimo en términos de encontrar la mejor solución posible dada una serie de arribos y energía, independientemente de cómo varíen esos valores. Minuto a minuto vamos calculándo cuál es el óptimo de enemigos a eliminar en el minuto que estoy parado, de esa forma si tenga mucha o poca variablidad en mis datos de entrada no va a camibar porque voy a estar comparando con los óptimos ya calculados.

4. Ejemplos de Ejecución

En esta sección vamos a mostrar como se comporta nuestro algoritmo implementado en Python para un ejemplo en particular.

La entrada de datos en este caso va a tener la siguiente forma

$$X = [5, 10, 5, 150, 50]$$
$$E = [5, 10, 20, 50, 80]$$

En este caso $n = 5$, por lo tanto el primer bucle recorrera desde el minuto 1 hasta el minuto 5 incluido. A continuación vamos a detallar como es la ejecución de cada iteración teniendo en cuenta que los valores iniciales de nuestros arreglos serán:

$$\texttt{opt} = [0, 0, 0, 0, 0, 0]$$
$$\texttt{x} = [0, 5, 10, 5, 150, 50]$$
$$\texttt{e} = [0, 5, 10, 20, 50, 80]$$

- Minuto 1

  Al principio de cada iteración inicializamos el optimo de ese minuto en 0 y como estamos en el minuto 1 solo gurdamos 1 minuto de energía y usamos el óptimo del minuto 0. En este caso:

  $$\texttt{opt\_parcial} = \texttt{opt}[0] + \text{mín}(\texttt{e}[1-0], \texttt{x}[1]) = 0 + \text{mín}(5, 5) = 5$$

  Como el optimo parcial es el único calculado por el momento lo guardamos como el mejor óptimo dentro de nuestra lista de óptimos $\texttt{opt}[1] = 5$. Al terminar el bucle quedará como la mayor cantidad de enemigos que se pueden eliminar en ese minuto.

- Minuto 2

  En este punto, avanzamos y calculamos el óptimo para el segundo minuto, donde podemos guardar 1 o 2 minuto(s) de energía y usamos los óptimos de los minutos 0 y 1 previamente calculados. Para la iteración del minuto 0 queda:

  $$\texttt{opt\_parcial} = \texttt{opt}[0] + \text{mín}(\texttt{e}[2-0], \texttt{x}[2]) = 0 + \text{mín}(10, 10) = 10$$

  Este calculo es lo mismo que decir: cuantos enemigos reduciría si no atacara por 2 minutos y luego atacara con la llegada de enemigos en el minuto 2.

  Como por el momento parece ser la mejor opción, la guardamos quedando que el $\texttt{opt}[2] = 10$.

  Ahora calculamos la iteración del minuto 1 quedando:

  $$\texttt{opt\_parcial} = \texttt{opt}[1] + \text{mín}(\texttt{e}[2-1], \texttt{x}[2]) = 5 + \text{mín}(5, 10) = 10$$

  Como lo calculado no es mayor al optimo parcial anterior no reescribimos el valor del óptimo y pasamos al siguiente minuto.

  Notar que hubieramos tenido el mismo óptimo si atacaramos con 1 minuto de energía pero tomando el óptimo del minuto 1. Esto nos indica que puede haber varias posibilidades para un mismo caso.

- Minuto 3

  Para el minuto 3 haremos lo mismo pero usando los óptimos de los minutos 0,1 y 2 calculados previamente y teniendo en cuenta los escenarios donde podemos guardar 1,2 o 3 minutos de energía.

  Para la iteración del minuto 0 queda:

  $$\texttt{opt\_parcial} = \texttt{opt}[0] + \text{mín}(\texttt{e}[3-0], \texttt{x}[3]) = 0 + \text{mín}(20, 5) = 5$$

  Como parece ser la mejor opción por ahora la guardamos siendo $\texttt{opt}[3] = 5$.

  Para la iteración del minuto 1 queda:

  $$\texttt{opt\_parcial} = \texttt{opt}[1] + \text{mín}(\texttt{e}[3-1], \texttt{x}[3]) = 5 + \text{mín}(10, 5) = 10$$

En este caso, atacar en el primer minuto y luego esperar 2 minutos para el siguiente ataque parece ser la mejor opción para el minuto 3, entonces guardamos $\texttt{opt}[3] = 10$

Para la iteración del minuto 2 queda:

$$\texttt{opt\_parcial} = \texttt{opt}[2] + \text{mín}(\texttt{e}[3-2], \texttt{x}[3]) = 10 + \text{mín}(5, 5) = 15$$

Vemos que en la última iteración obtenemos el mejor óptimo asi que nos guardamos ese valor quedando que el $\texttt{opt}[3] = 15$.

Como la idea ya se entendió, se motrará para los siguientes minutos, solamente las ecuciones para ver los valores que se van tomando.

- Minuto 4

$$\texttt{opt\_parcial} = \texttt{opt}[0] + \text{mín}(\texttt{e}[4-0], \texttt{x}[4]) = 0 + \text{mín}(50, 150) = 50$$

$$\texttt{opt\_parcial} = \texttt{opt}[1] + \text{mín}(\texttt{e}[4-1], \texttt{x}[4]) = 5 + \text{mín}(20, 150) = 20$$

$$\texttt{opt\_parcial} = \texttt{opt}[2] + \text{mín}(\texttt{e}[4-2], \texttt{x}[4]) = 10 + min(10, 150) = 20$$

$$\texttt{opt\_parcial} = \texttt{opt}[3] + \text{mín}(\texttt{e}[4-3], \texttt{x}[4]) = 15 + \text{mín}(5, 150) = 20$$

El óptimo que se guardará será 50 que fue el calculado en la primera iteración, cuando analizamos la estrategía de atacar con 4 minutos de energía.

- Minuto 5

$$\texttt{opt\_parcial} = \texttt{opt}[0] + \text{mín}(\texttt{e}[5-0], \texttt{x}[5]) = 0 + \text{mín}(80, 50) = 50$$

$$\texttt{opt\_parcial} = \texttt{opt}[1] + \text{mín}(\texttt{e}[5-1], \texttt{x}[5]) = 5 + \text{mín}(50, 50) = 55$$

$$\texttt{opt\_parcial} = \texttt{opt}[2] + \text{mín}(\texttt{e}[5-2], \texttt{x}[5]) = 10 + \text{mín}(20, 50) = 30$$

$$\texttt{opt\_parcial} = \texttt{opt}[3] + \text{mín}(\texttt{e}[5-3], \texttt{x}[5]) = 15 + \text{mín}(10, 50) = 25$$

$$\texttt{opt\_parcial} = \texttt{opt}[4] + \text{mín}(\texttt{e}[5-4], \texttt{x}[5]) = 50 + \text{mín}(5, 50) = 55$$

De esta forma, la máxima cantidad de enemigos que se pueden eliminar es 55.

Por último mostraremos en una matriz los valores que se fueron tomando para que quede mejor ilustrado como fueron los paso a paso:

		Arribo de enemigos en el minuto i					
	↓ e \ x →	0	1	2	3	4	5
Energía acumluada en el minuto i	0	0	0	0	0	0	0
	1	0	5	10	5	50	50
	2	0		10	10	20	55
	3	0			15	20	30
	4	0				20	30
	5	0					55

Ahora pasaremos a explicar un poco la reconstrucción de la solución de forma tal que podamos comprender y obtener cuando hay que atacar y cuando hay que cargar para llegar a la solución óptima.

En este caso las variables comenzarán inicializadas de la siguiente forma:

$$\texttt{s} = [\texttt{'Atacar'}]$$
$$\texttt{x} = [0, 5, 10, 5, 150, 50]$$
$$\texttt{e} = [0, 5, 10, 20, 50, 80]$$
$$\texttt{tope} = n = 5$$

De igual forma el primer bucle comenzará en $\texttt{n} = 5$ y recorrerá todos los minutos hasta el 1, y a su vez el `tope` comenzará en la última posición `n`. Para cada iteración se calcularán los siguientes valores:

- `dif_opt`: Diferencia entre los valores óptimos entre la posición `tope` y el minuto `i` en el cual se encuentre la iteración.
- `e_dif`: Valor de la función de carga de energía para la posición `tope-i`.
- `enemigos_tope`: Valor del ataque enemigo para el minuto correspondiente al `tope`.

Vamos a detallar como es la ejecución de cada iteración para comprender como es la reconstrucción de la solución:

- Minuto 5

  Al ser el último minuto, siempre se realizará una acción de `Atacar`, ya que no tiene sentido guardarse fuerza.

- Minuto 4

  - `dif_opt`: 5

  - `e_dif`: 5

  - `enemigos_tope`: 5

  Como `enemigos_tope` = mín(`dif_opt`, `e_dif`), es decir, que la cantidad de enemigos eliminados coincide con el valor de enemigos eliminados entre el `tope` y el minuto analizado, el minuto 4 se corresponde con una acción de `Atacar`. Se cambia el valor de $\texttt{tope} = 4$

- Minuto 3

  - `dif_opt`: 35

  - `e_dif`: 5

  - `enemigos_tope`: 150

  `enemigos_tope !=` mín(`dif_opt`, `e_dif`), el minuto 3 corresponde a una acción de `Cargar` debido a que la diferencia de enemigos eliminados entre el minuto actual y el `tope` no coincide con la cantidad de enemigos que se pueden eliminar en el minuto de `tope`.

- Minuto 2

  - `dif_opt`: 40

  - `e_dif`: 10

  - `enemigos_tope`: 150

  `enemigos_tope !=` mín(`dif_opt`, `e_dif`), en el minuto 2 corresponde `Cargar`.

- Minuto 1

  - `dif_opt`: 45

  - `e_dif`: 20

  - `enemigos_tope`: 150

  `enemigos_tope !=` mín(`dif_opt`, `e_dif`), en el minuto 1 corresponde `Cargar`.

Finalmente la forma de s será:

$$\texttt{s} = [\texttt{'Cargar'}, \texttt{'Cargar'}, \texttt{'Cargar'}, \texttt{'Atacar'}, \texttt{'Atacar'}]$$

5. Mediciones Temporales

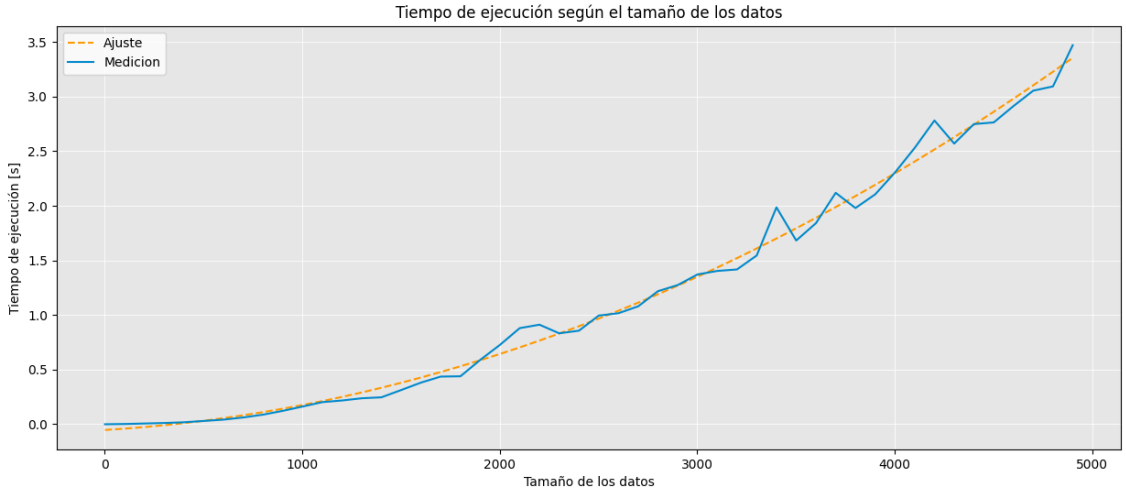
Para las mediciones temporales de este trabajo se generaron distintos set de datos de distintos tamaños y distintos pasos para poder comprobar la complejidad temporal de nuestro algorítmo.

Es clave notar que a medida que aumentemos la cantidad de arribos que hay, el tiempo que tarda en buscar la solución óptima aumenta considerablemente.

Vamos a ver como se ajusta a la complejidad de dicho algorítmo que es de $O(n^2)$ realizando para las primeras dos mediciones un ajuste por cuadrados mínimos para poder comparar.

5.1. Medición: 5000 minutos - Paso 100

Para la primera medición tomamos un set de datos que va desde 1 minuto de arribos de enemigos hasta 5000 minutos de arribos de enemigos. Cada muestra tiene un paso de 100. Como se puede notar en la imagen, el polinomio de grado 2 se ajusta bastante a nuestras mediciones.



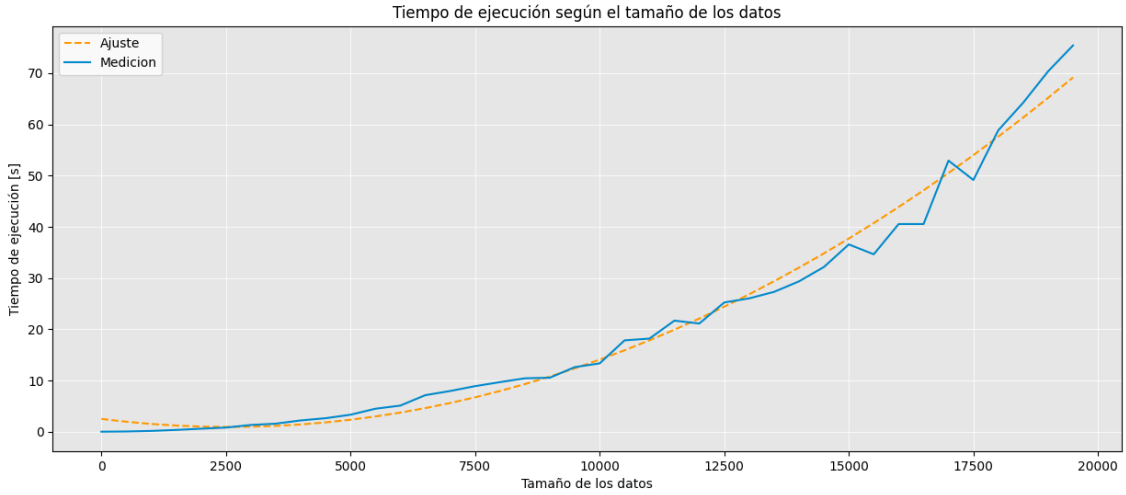
El polinomio de ajuste corresponde a la función $p(x) = 0{,}000000119x^2 + 0{,}000108x - 0{,}0524$ y el error cuadrático medio que tenemos corresponde a: ECM = 0,00774

5.2. Medición: 20000 minutos - Paso 500

Para la segundda medición tomamos un set de datos que va desde 1 minuto de arribos de enemigos hasta 20000 minutos de arribos de enemigos. Cada muestra tiene un paso de 500. A diferencia del anterior buscamos ver como se comporta nuestro algoritmo para un volumen mucho

mas grande de datos pero aumentando el paso ya que el mismo suele tardar bastante a medidad que se va aumentando el volumen de entrada de datos.

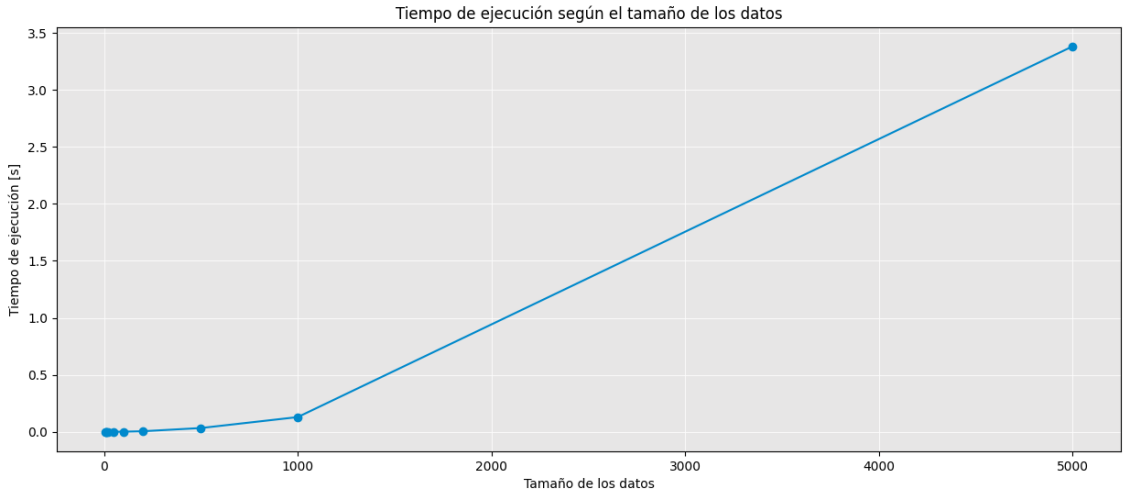
En este caso se ve de forma más clara la curva parabólica de grado 2 que toma nuestro algoritmo ajustandose bastante al polinomio que mejor se ajusta.



El polinomio de ajuste corresponde a la función $p(x) = 0{,}000000238x^2 - 0{,}00122x + 2{,}496$ y el error cuadrático medio que tenemos corresponde a: $ECM = 6{,}6698$

5.3. Medición: Datos del Curso

Para la última medición tomamos las muestras que nos proporcionó el curso donde el tamaño de los datos correponden a: 5, 10, 20, 50, 100, 200, 500, 1000, 5000 minutos.



6. Conclusiones

Como conclusión de este trabajo práctico se puede notar la gran utilidad que tiene la programación dinámica al momento de plantear un problema. En principio parecía un problema combinatorio computacionalmente complejo, pero con un análisis previo se pueden notar ciertos patrones que simplifican mucho la solución haciéndola más eficiente.

La parte más desafiante sin dudas es el planteo de la ecuación de recurrencia. Pero una vez que se comprende como se construyen y se componen los subproblemas, la implementación no se dificulta tanto.

En caso de que se puedan utilizar algoritmos de este estilo, conviene mucho implementarlos. Proporcionan una solución eficiente y elegante a un problema que a priori tiene una complejidad exponencial.