.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Programación Dinámica para el Reino de la Tierra

13 de octubre de 2025

Nombre y Apellido	Padrón
Víctor Zacarías	107080
Carolina Aramayo	106260
Francisco Nahuel Tapia	107128

1. Índice

Índice

2. Análisis del problema

Se tiene como objetivo analizar y resolver de forma óptima el siguiente problema:

Se sabe que la ciudad de *Ba Sing Se* va a ser atacada por los soldados de la Nación del Fuego durante un intervalo de N minutos.

Gracias a un análisis de inteligencia se descubrió que la Nación del Fuego va a enviar soldados en ráfagas cada minuto durante los N minutos que dure el ataque. Dichas ráfagas son conocidas de antemano y forman parte de la entrada del problema.

Por ejemplo, si un ataque dura tres minutos una posible secuencia de ataques es la siguiente:

$$Atqs = [230, 450, 120]$$

El arreglo *Atqs* indica cuantos soldados atacaran *Ba Sing Se* en el minuto i-ésimo. Véase, en el primer minuto llegan 230 soldados, en el segundo minuto 450, y en el tercero, 120.

NOTA: Cuando se hablaba de ráfagas no se hacía más que hacer referencia a la cantidad de soldados que llegan a las puertas de *Ba Sing Se* en un minuto cualquiera de los N minutos del ataque.

Por otra parte, los defensores de la ciudad disponen de un poder de ataque acumulativo, $F(j)$. $F(j)$ indica cuantos soldados enemigos se pueden asesinar en un minuto determinado si los defensores de la ciudad cargan su poder durante j minutos.

NOTA: $F(j)$ es una función monótona creciente.

Para que se entienda mejor, si se tiene una batalla que tiene una duración de $N = 15$ minutos, y desde primer minuto de la batalla se estuvo acumulando el poder de ataque hasta el minuto 5 se van a poder matar $F(5)$ enemigos.

Durante el transcurso de una batalla, por cada minuto que pasa se puede liberar el poder de ataque, matando a $F(j)$ enemigos, o puede seguir acumulándose. Si en un minuto cualquiera se libera el poder acumulado, entonces se debe acumular el poder desde cero para el próximo ataque.

Es de interés diseñar un algoritmo que, mediante programación dinámica de termine la secuencia de cargas y ataques de modo de maximizar la cantidad de enemigos/atacantes asesinados.

A modo de analizar mejor el problema a resolver, se hará uso de un ejemplo con una combinación de ataques y función de poder acumulativa válida.

Ejemplo 1

Tomando como ejemplo al archivo de ataques provisto por la cátedra, **5.txt,** se tiene que:

$$Atqs = [271, 533, 916, 656, 664]$$
$$F(x) = [21, 671, 749, 833, 1543]$$

NOTA: A modo de simplificación para este ejemplo los arreglos empiezan en la posición 1 (y terminan en la 5).

- El ataque dura 5 minutos, en el minuto i-ésimo de esos 5 minutos llegan ráfagas de k enemigos, por ejemplo, en el minuto $i = 2$ llegan $k = 533$ enemigos y en el minuto $i = 3$ llegan $k = 916$.
- $F(j)$ indica cuantos enemigos se pueden asesinar si se carga el poder durante j minutos, por ejemplo, para $j = 4$ se pueden matar a 833 enemigos. Nótese que $F(j)$ es monótona creciente, tal y como se menciono anteriormente.
- $F(j)$ empieza sin carga alguna, y al llegar la primera ráfaga de enemigos, es decir cuando $i = 1$, $j = 1$ por lo cual se podrían matar 21 enemigos de los 271 que llegan.
- Si se decide liberar el poder acumulado en el minuto i la cantidad de enemigos que se pueden

asesinar es $min(Atqs[i], F[j])$, lo que quiere decir, que se pueden asesinar a todos los enemigos que arriban en i (solo si $F(j) \geq Atqs[i]$) o los que permita matar el poder acumulado (es decir, cuando $F(j) < Atqs[i]$).

- Si se está en el minuto i, como mucho se puede cargar el poder hasta $N - i$ minutos (se puede cargar el poder un minuto más o hasta el último ataque).

Una solución posible para este problema puede ser:

$Cargar, Cargar, Cargar, Atacar, Atacar$

1. **Minuto 1:** Llegan 271 enemigos, pero como se carga el poder no se mata a ninguno.
2. **Minuto 2:** Llegan 533 enemigos, nuevamente, el poder se carga y no se mata a ninguno.
3. **Minuto 3:** Llegan 916 enemigos, ídem con los casos anteriores.
4. **Minuto 4:** Llegan 656 enemigos, se acumulo poder durante 4 minutos ($F(4) = 833$), por lo que la cantidad de enemigos asesinados es $min(656, 833) = 656$.
5. **Minuto 5:** Llegan 664, como el poder de ataque se libero en el minuto anterior, para este punto transcurrió un minuto desde que se uso el poder, lo que quiere decir que $j = 1$ y $F(1) = 21$, por lo cual la cantidad de enemigos asesinados es $min(664, 21) = 21$.

La cantidad total de enemigos asesinados es 677. Nótese que si en un minuto i se acumula poder de ataque, no se hace nada con los enemigos que arriban a la ciudad. En cada minuto la cantidad de enemigos es $Atqs[i]$ y nada más, no se acumulan los enemigos no asesinados entre arribos.

Sobre este último punto, se podría pensar que los enemigos que no se asesinan entran a la ciudad y la seguridad interna es la que se encarga de estos. El objetivo principal es matar a todos los enemigos que se puedan en la entrada, minimizando a los que entran a la ciudad (y en consecuencia maximizando a los enemigos asesinados en la entrada).

Fin ejemplo 1

Yendo al problema que el presente trabajo busca resolver (utilizar programación dinámica para maximizar la cantidad total de enemigos asesinados), si se tiene un ataque de N minutos se debe escoger a la combinación de cargas y ataques adecuadas entre los N minutos tales que la cantidad total de enemigos asesinados sea máxima.

Para un ataque de N minutos con una función $F(j)$:

$Atqs = [i_0, ..., i_{n-1}]$
$F(j) = [j_0, ..., j_{n_1}]$

Se quiere saber la cantidad máxima de enemigos que se pueden matar para un ataque de N minutos. Es lógico decir que en el minuto N siempre conviene atacar (sin importar la carga acumulada), al ser el último minuto se sabe que lo mejor que se puede hacer es atacar y liberar el poder acumulado (no hay otra cosa que hacer).

De esto último, se puede entender que la lógica inductiva del algoritmo se basa en decir "estoy en el minuto i, desde cuando me convino haber cargado para atacar ahora?". Esto es, en cada paso del algoritmo se parte del espacio de soluciones previamente construido para tomar la mejor decisión actual, y este espacio de soluciones se representa cuanto convino haber cargado el poder para cada i.

En base a la mencionado anteriormente, una posible ecuación de recurrencia es:

$Opt[i] = max(\{opt[i - j] + min(Atqs[i], F[j])\}) \forall j / 0 \leq j \leq i$

$$0 \leq i \leq N - 1$$

La ecuación escrita quiere decir, si el ataque dura N minutos, para el minuto i el óptimo viene determinado por cual de todos los óptimos anteriores maximiza la cantidad total de enemigos asesinados hasta el minuto i, teniendo en cuenta también que dependiendo del óptimo anterior escogido se puede acumular más o menos poder de ataque.

3. Algoritmo propuesto

A continuación se explicará y analizará el algoritmo propuesto para la defensa de *Ba Sing Se*.

El algoritmo consta de dos funciones:

- **planear estrategia(soldados, ataques)**: Esta función recibe como entrada un arreglo de longitud N con los soldados que atacan en cada minuto de los N minutos que dura el ataque a la ciudad, se recibe también la función de poder acumulativa (es un arreglo).

  La función devuelve a la cantidad máxima de enemigos asesinados durante los N minutos y hace uso de la función **reconstruccion(opt, soldados, ataques)** para reconstruir la solución y devolver la secuencia de cargas y ataques que llevan al óptimo.

  El algoritmo implementado no hace nada más que hacer uso de la ecuación de recurrencia para construir el espacio de soluciones con un enfoque *bottom up*.

  El arreglo de óptimos es de longitud $N + 1$ a modo de poder considerar el caso nulo de 0 ataques.

  Por otra parte, el ciclo *for* no hace nada más que iterar por cada ataque y encontrar en cual de los óptimos anteriores convino haber cargado de modo de maximizar a los enemigos asesinados (durante la iteración actual).

- **reconstruccion(opt, soldados, ataques)**: Recibe como entrada al arreglo de óptimos generado por el algoritmo de programación dinámica, al arreglo de ataques y a la función de poder acumulativa.

  El objetivo de esta función es reconstruir y devolver la secuencia de ataques y cargas requerida para la salida de **planear estrategia(soldados, ataques)**.

  En este caso se hace uso una especie de lógica "top down"para reconstruir la secuencia de acciones adoptadas durante la defensa de la ciudad.

  La solución se encuentra contenida de forma implícita en el arreglo de óptimos.

  El arreglo de óptimos se recorre de fin a inicio. En cada posición se calcula a cuantas cargas de poder corresponde el óptimo actual (es decir, cuanto se tuvo que haber cargado el poder para matar a M enemigos en la posición actual), de esta forma se obtiene la solución reconstruida.

  NOTA: Por top down solo se hace referencia a reconstruir la solución desde arriba hacia abajo, no a una lógica recursiva.

```
1  def planear_estrategia(soldados, ataques):
2      opt = [0] * (len(soldados) + 1)
3      for i in range(1, len(soldados) + 1): # O(N)
4          for carga in range(1, i + 1): # O(i+1) -> Para la  ltima  batalla se tienen
       N iteraciones!
5              ultimo_ataque = i - carga
6              rafaga_actual = soldados[i-1]
7              ataque_actual = ataques[carga-1]
8
9              opt[i] = max(opt[i], opt[ultimo_ataque] + min(rafaga_actual,
       ataque_actual))
10
11     return opt[-1], reconstruccion(opt, soldados, ataques)
12
13
14 def reconstruccion(opt, soldados, ataques):
15     estrategia = [''] * len(soldados)
16     i = len(soldados)
17     while i > 0: # O(N)
18         for carga in range(1, i+1):
19             ultimo_ataque = i - carga
20             rafaga_actual = soldados[i-1]
21             ataque_actual = ataques[carga-1]
22
23             if opt[i] == opt[ultimo_ataque] + min(rafaga_actual, ataque_actual):
```

```
                estrategia[i-1] = 'Atacar'
                i = i - carga
                break
            estrategia[ultimo_ataque-1] = 'Cargar'
    return estrategia
```

Listing 1: Algoritmo de planificación de la defensa de Ba Sing Se

3.1. Análisis de la complejidad del algoritmo

El algoritmo cuenta con una entrada de tamaño N, correspondiente a la duración del ataque a *Ba Sing Se*, reciben dos arreglos, uno con la cantidad de soldados que atacan en cada minuto, y otro representa a la función $F(x)$, ambos arreglos tienen longitud N.

El tamaño del problema solo impacta al algoritmo en la cantidad de iteraciones que se realizan para construir los óptimos y en reconstruir la solución. La variabilidad de los datos puede impactar en el tiempo de reconstrucción del algoritmo, sin embargo, la complejidad en el peor de los casos sigue siendo la misma.

En la sección **3.2** se realizará un análisis del impacto que puede tener en el algoritmo la variabilidad de los datos de entrada.

1. Durante la planificación de la defensa se itera el arreglo de óptimos desde el índice 1 hasta la última posición, el arreglo de óptimos es de longitud $N+1$ por lo que realizar esta operación tiene complejidad temporal $O(N)$ ya que se itera desde el índice 1 hasta el final y no desde el índice 0.

   Referencia: `Python: for i in range (1 , len ( soldados ) + 1) :`

2. Por cada iteración del arreglo de óptimos se revisa cual de los óptimos anteriores (últimos ataques) maximiza a la cantidad de enemigos que se pueden asesinar (para el óptimo actual). Esto se realiza mediante una iteración anidada.

   Referencia: `Python: for carga in range (1 , i + 1) :`

   A simple vista se puede decir que se itera i veces, teniendo una complejidad temporal $O(i)$. Sin embargo, si se desarrollan todas las iteraciones se puede notar que durante el calculo de $opt[N]$ se realizan N subiteraciones al final.

3. Si se desarrolla lo dicho en los puntos 2 y 3, la complejidad temporal de las iteraciones es:

   $Opt(N) * Opt(1 + ... + i + ... + n)$

   $Opt(N * 1 + ... + N * i + N^2)$

   $Opt(N^2)$

   Por lo cual, la complejidad de planear las estrategias es $C = O(n^2)$ (notar que como la complejidad es una cota asintótica, a modo de simplificación solo se toman en cuenta los términos de "mayor tamaño", es decir, N^2).

4. La complejidad de reconstruir la solución depende del tamaño del problema, ya que mientras más larga sea la batalla, mayor será la solución a reconstruir.

   Para reconstruir la solución se itera el arreglo de óptimos de fin a inicio. En el peor de los casos tiene como complejidad temporal $O(n)$, que corresponde al caso de iterar el vector por completo de fin a inicio.

   Nota: Se habla del peor de los casos, ya que dada la naturaleza de la reconstrucción se pueden hacer saltos a posiciones anteriores del arreglo de óptimos en lugar de siempre iterar de a un paso desde fin a inicio.

5. El resto de operaciones en ambos algoritmos, no son más que simples sumas, asignaciones y comparaciones, por lo que su complejidad temporal es $O(1)$.

   Respecto a este punto, en los puntos **2** y **3** las complejidades resultantes, tienen el valor que tienen debido a que no son más que una sucesión de operaciones constantes (asignaciones, sumas, etc.).

   Ejemplo: Si se itera un arreglo de longitud N, y por cada iteración se suma al valor actual del arreglo a una variable auxiliar, se está realizando una operación de complejidad temporal $O(N)$.

   Lo mismo aplica para las iteraciones del algoritmo propuesto.

6. La complejidad resultante es:

   $C(planeoestrategias) + C(reconstruccion) = O(n^2) + O(n) = O(n^2)$

3.2. Análisis de la variabilidad de los tiempos de llegada y las decisiones adoptadas (atacar o cargar)

Efecto sobre la optimalidad:

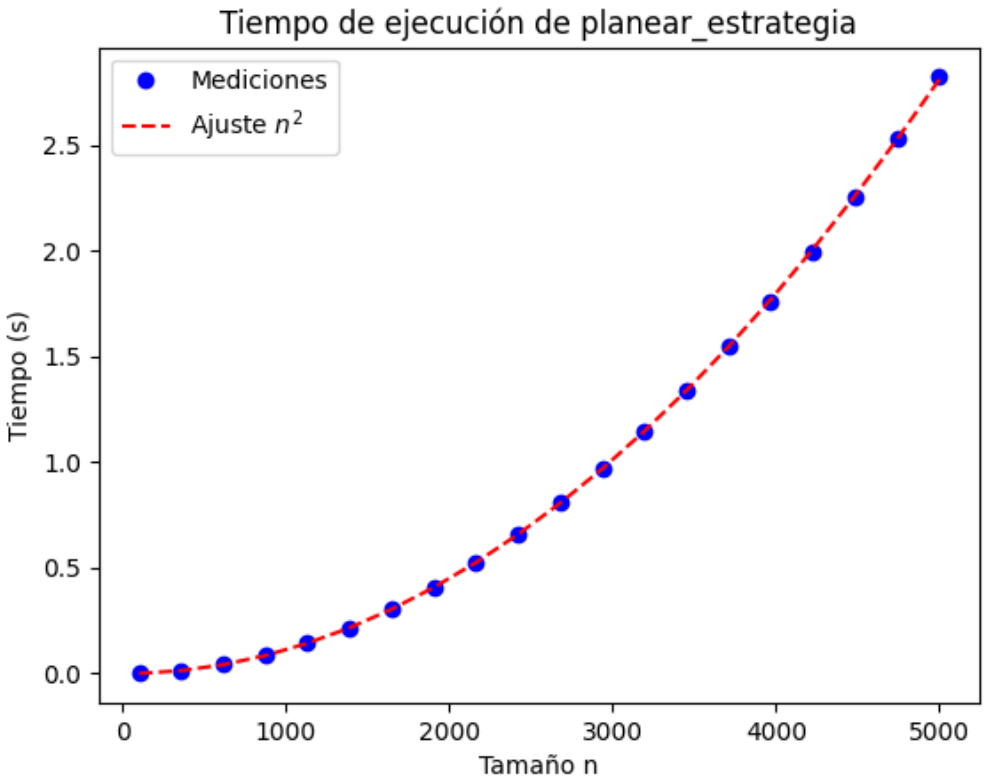
La variabilidad de los datos sí afecta a la estrategia óptima encontrada (a la secuencia de ataques y cargas), pero no a la optimalidad del algoritmo. El algoritmo de programación dinámica garantiza encontrar la mejor secuencia de cargas y ataques para cualquier combinación de valores de Atqs[i] y $F(j)$, ya que explora todas las posibilidades y maximiza el total de enemigos asesinados en la entrada. Si la función de recarga $F(j)$ crece rápido, puede convenir esperar más minutos antes de atacar; si las oleadas Atqs[i] son muy grandes y la recarga crece lento, puede convenir atacar más seguido. En todos los casos, el algoritmo se adapta y encuentra la solución óptima para los datos concretos.

4. Medición de tiempos

Para corroborar la complejidad teórica del algoritmo, se realizaron mediciones de tiempo utilizando arreglos de entrada de diferentes tamaños, generadas con valores pseudoaleatorios mediante `numpy`. Para garantizar la reproducibilidad de los experimentos, se fijó la semilla de aleatoriedad antes de generar los datos.

Para cada tamaño n, se ejecutó el algoritmo 10 veces y se promedió el tiempo de ejecución para reducir el ruido de medición. Los tamaños de entrada variaron de 100 a 5 000 elementos.

A continuación se muestra el gráfico obtenido, junto con el ajuste por cuadrados mínimos a la función teórica $O(n^2)$ (la complejidad teórica del algoritmo implementado):

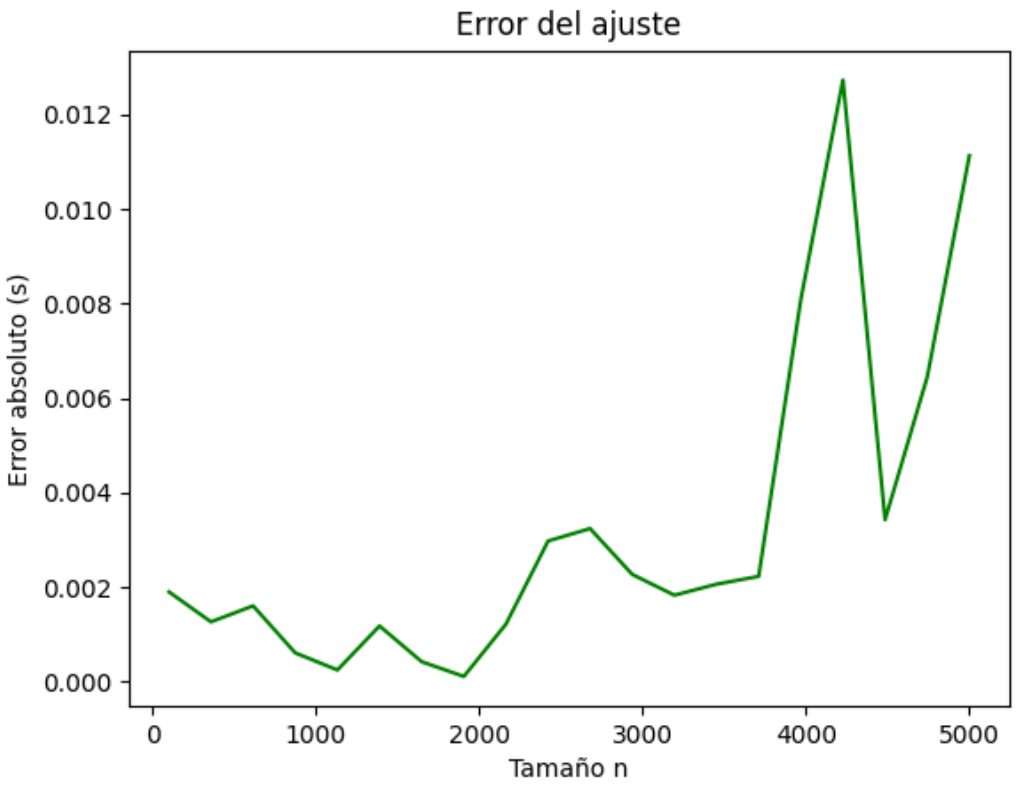


Como se observa, los datos experimentales se ajustan correctamente a la curva teórica esperada, validando la complejidad $O(n^2)$. El ajuste se realizó utilizando la técnica de cuadrados mínimos con `curve_fit` de `scipy`.

- Se promediaron 10 mediciones por tamaño para cada punto.
- Los tamaños de entrada variaron de 100 a 5 000 elementos.

4.1. Análisis del error del ajuste

Además del gráfico de tiempos, se presenta el gráfico del error absoluto entre los valores medidos y la curva ajustada. Este gráfico permite visualizar la calidad del ajuste realizado mediante cuadrados mínimos.



Como se observa, el error absoluto es bajo en la mayoría de los casos, lo que indica un buen ajuste del modelo teórico a los datos experimentales.

5. Ejemplos de ejecucion

Para ejemplificar que el algoritmo de programación dinámica propuesto obtiene la solución óptima, se presentan casos pequeños donde se puede calcular manualmente el resultado óptimo. Es importante destacar que, en algunos casos, pueden existir varias estrategias óptimas (distintos momentos de ataque y recarga) que logran el mismo total de soldados eliminados. Lo relevante es que el algoritmo encuentra el valor óptimo, aunque la secuencia de ataques pueda diferir.

Ejemplo 1: cinco minutos

Entrada:

Minuto	Soldados x_i	Fuerza de ataque $f(j)$
1	15	10
2	40	20
3	25	30
4	50	40
5	35	50

Cálculo manual de la estrategia óptima:

Supongamos que se atacan a los enemigos en los minutos 2, 4 y 5:

- **Ataque en minuto 2:** han pasado 2 minutos desde el inicio, carga $f(2) = 20$, oleada $x_2 = 40$. Se eliminan $\text{mín}(40, 20) = 20$ soldados.
- **Ataque en minuto 4:** han pasado $4-2 = 2$ minutos desde el último ataque, carga $f(2) = 20$, oleada $x_4 = 50$. Se eliminan $\text{mín}(50, 20) = 20$ soldados.
- **Ataque en minuto 5:** han pasado $5-4 = 1$ minuto desde el último ataque, carga $f(1) = 10$, oleada $x_5 = 35$. Se eliminan $\text{mín}(35, 10) = 10$ soldados.

[Cargar, Atacar, Cargar, Atacar, Atacar]

Total eliminados: $20 + 20 + 10 = 50$

Otra estrategia posible es atacar en cada minuto, el total sería:

- Minuto 1: $f(1) = 10$, $x_1 = 15 \rightarrow 10$
- Minuto 2: $f(1) = 10$, $x_2 = 40 \rightarrow 10$
- Minuto 3: $f(1) = 10$, $x_3 = 25 \rightarrow 10$
- Minuto 4: $f(1) = 10$, $x_4 = 50 \rightarrow 10$
- Minuto 5: $f(1) = 10$, $x_5 = 35 \rightarrow 10$

[Atacar, Atacar, Atacar, Atacar, Atacar]

Total eliminados: $10 + 10 + 10 + 10 + 10 = 50$

Ejecución del programa:

```
$ python3 main.py 3 ejemplo1.txt
--------------------------------------------------------------------------------
Estrategia para: ejemplo1.txt
Enemigos liquidados: 50
Acciones:
1. Atacar
```

```
2. Atacar
3. Atacar
4. Atacar
5. Atacar
--------------------------------------------------------------------------------
```

En este caso, el algoritmo encontró una de las estrategias óptimas posibles. Lo importante es que el total coincide con el máximo calculado manualmente.

—

Ejemplo 2: cuatro minutos

Entrada:

Minuto	Soldados x_i	Fuerza de ataque $f(j)$
1	12	5
2	18	10
3	30	30
4	25	32

Cálculo manual:

Supongamos que se ataca en los minutos 3 y 4:

- **Ataque en minuto 3:** han pasado 3 minutos desde el inicio, carga $f(3) = 30$, oleada $x_3 = 30$. Se eliminan $\min(30, 30) = 30$ soldados.
- **Ataque en minuto 4:** han pasado $4-3 = 1$ minutos desde el último ataque, carga $f(1) = 5$, oleada $x_4 = 25$. Se eliminan $\min(25, 5) = 5$ soldados.

[Cargar, Cargar, Atacar, Atacar]

Total eliminados: $30 + 5 = 35$

Ejecución del programa:

```
$ python3 main.py 3 ejemplo2.txt
--------------------------------------------------------------------------------
Estrategia para: ejemplo2.txt
Enemigos liquidados: 35
Acciones:
1. Cargar
2. Cargar
3. Atacar
4. Atacar
--------------------------------------------------------------------------------
```

—

Ejemplo 3: tres minutos

Entrada:

Minuto	Soldados x_i	Fuerza de ataque $f(j)$
1	3	15
2	40	30
3	70	50

Cálculo manual:

Supongamos que se ataca solo en el minuto 3:

- **Ataque en minuto 3:** han pasado 3 minutos desde el inicio, carga $f(3) = 50$, oleada $x_3 = 70$. Se eliminan $\text{mín}(70, 50) = 50$ soldados.

[Cargar, Cargar, Atacar]

Total eliminados: 50

Ejecución del programa:

```
$ python3 main.py 3 ejemplo3.txt
--------------------------------------------------------------------------------
Estrategia para: ejemplo3.txt
Enemigos liquidados: 50
Acciones:
1. Cargar
2. Cargar
3. Atacar
--------------------------------------------------------------------------------
```

6. Conclusiones

El análisis y la implementación del algoritmo de programación dinámica para la defensa de *Ba Sing Se* permitió resolver de manera óptima el problema de maximizar la cantidad de enemigos eliminados en la entrada de la ciudad. A diferencia de enfoques Greedy, la programación dinámica explora todas las combinaciones posibles de cargas y ataques (implícitamente), garantizando encontrar la mejor estrategia para cualquier configuración de oleadas y función de recarga.

Las mediciones experimentales de tiempo, realizadas sobre conjuntos de datos de distintos tamaños y variabilidad, confirmaron la complejidad $O(N^2)$ del algoritmo. El ajuste por cuadrados mínimos mostró un error absoluto bajo, validando la eficiencia y la correcta implementación del método.

Los ejemplos de ejecución presentados demuestran que el algoritmo encuentra siempre una estrategia óptima, incluso en casos donde existen varias secuencias de ataques que logran el mismo resultado máximo. En todos los casos, los resultados obtenidos manualmente coincidieron con los generados por el programa, lo que valida la optimalidad y robustez de la solución.

En conclusión, el algoritmo propuesto es eficiente, óptimo y se adapta correctamente a la variabilidad de los datos, resolviendo el problema de manera confiable tanto en teoría como en la práctica.