Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2: Programación Dinámica para el Reino de la Tierra

13 de octubre de 2025

Tomas Goncalves Rei
111405

Valentín Abaca
112088

Lucia De Sequera Nebreda
INT01

Alexis Maximiliano Torrez Vargas
111449

Índice

1. Introducción

1.1. Enunciado

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 $x_1, x_2, \cdots, x_n$.
- 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 $\text{mín}(x_k, f(j))$ 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.

1.2. Análisis

El problema describe una situación en la que somos los jefes estratégicos de los los *Dai Li*, la policía secreta de la ciudad, y debemos planificar ataques óptimos para eliminar la mayor cantidad posible de enemigos durante un ataque de múltiples oleadas.

Cada minuto i llegan x_i enemigos. El equipo puede decidir **atacar** o **no atacar** en cada minuto, pero cada ataque consume toda la energía acumulada y reinicia el proceso de carga.

1. Datos de entrada

- n: cantidad total de minutos o rondas de ataque.
- x_i: cantidad de enemigos que llegan en el minuto i.
- $f(j)$: cantidad de enemigos que puede eliminar un ataque luego de haber cargado energía durante j minutos consecutivos.
- La función $f(j)$ es monótona creciente (a mayor tiempo de carga, mayor potencia del ataque).

2. Restricciones y reglas

En cada minuto se puede:

- **Atacar:** se destruyen mín$(x_i, f(j))$ enemigos, donde j es el tiempo transcurrido desde el último ataque.
- **No atacar:** se acumula energía (aumenta j para el siguiente minuto).

Al atacar, el contador de carga j se reinicia a 1.

El objetivo es **maximizar la cantidad total de enemigos eliminados** durante los n minutos.

1.3. Ecuación de recurrencia

Definimos $E(i)$ como la máxima cantidad de enemigos eliminados durante los primeros i minutos.

Para cada minuto i se consideran dos opciones:

- **No atacar en el minuto i:** En ese caso, la cantidad eliminada se mantiene igual que hasta el minuto anterior.
  $\Rightarrow E(i-1)$
- **Atacar en el minuto i:** Supongamos que el último ataque previo fue en el minuto j (con $j < i$). En ese caso, el ataque del minuto i podrá eliminar hasta mín$(x_i, f(i-j))$ enemigos, y al total se le suma lo que ya se había eliminado hasta el minuto j.
  $\Rightarrow E(j) + \text{mín}(x_i, f(i-j))$

Por lo tanto, la ecuación de recurrencia queda expresada como:

$$E(i) = \text{máx}\left(E(i-1),\ \max_{0 \le j < i}\left[E(j) + \text{mín}(x_i, f(i-j))\right]\right)$$

con condición inicial:

$$E(0) = 0$$

y el resultado final se obtiene evaluando:

$$E(n)$$

2. Desarrollo

2.1. Algoritmo propuesto

A continuación se presenta la implementación del algoritmo que resuelve el problema mediante **programación dinámica**. Se utiliza un enfoque iterativo basado en la ecuación de recurrencia previamente planteada.

```
def algoritmo(n, x, f):
    # eliminados[i] = máxima cantidad de enemigos eliminados hasta el minuto i
    eliminados = [0] * (n + 1)

    for i in range(1, n + 1):
        eliminados_sin_atacar = eliminados[i-1]   # Caso 1: no atacar en el minuto i
        eliminados_atacando = 0                   # Inicializa el mejor valor al atacar

        for j in range(0, i):
            # Caso 2: atacar en el minuto i
            # Si el último ataque fue en el minuto j, la carga fue (i - j) minutos.
            eliminados_atacando = max(
                eliminados_atacando,
                eliminados[j] + min(x[i], f[i-j])
            )

        # Se elige la mejor opción entre atacar o no atacar
        eliminados[i] = max(eliminados_sin_atacar, eliminados_atacando)

    return eliminados
```

—

Descripción paso a paso:

- Se inicializa el arreglo `eliminados` con ceros. Cada posición $eliminados[i]$ representa la máxima cantidad de enemigos eliminados hasta el minuto i.
- Para cada minuto i, se evalúan dos posibilidades:
  - **No atacar:** se conserva el valor anterior $eliminados[i-1]$.
  - **Atacar:** se prueban todos los posibles últimos ataques $j < i$ y se calcula $eliminados[j] + \text{mín}(x_i, f(i-j))$, eligiendo el máximo.
- Finalmente, se guarda en $eliminados[i]$ el mejor resultado entre atacar o no atacar.

—

3. Mediciones

3.1. Complejidad

Para poder analizar la complejidad del algoritmo, desglosaremos el código en distintas secciones.

La función principal llamada *algoritmo(n, x, f)* calcula el arreglo *eliminados*:

```
    for i in range(1, n + 1):
        eliminados_sin_atacar = eliminados[i-1]
        eliminados_atacando = 0
        for j in range(0,i):
            eliminados_atacando = max(eliminados_atacando, eliminados[j] + min(x[i
    ], f[i - j]))
        eliminados[i] = max(eliminados_sin_atacar, eliminados_atacando)
```

En esta sección del código observamos un bucle externo, el cual se ejecuta n veces. Dentro de cada iteración hay un bucle interno que realiza i iteraciones, donde las cuales tienen una complejidad constante, por lo que la complejidad temporal del algoritmo sera $O(n^2)$

Como segunda parte, tenemos la función *construir estrategia(n, eliminados,x,f)*, la cual se encarga de la reconstrucción de la solución de la programación dinámica.

```
    i = n
    while i > 0:
        if eliminados[i] == eliminados[i-1]:
            i -= 1
        else:
            for j in range(0,i):
                if eliminados[i]==eliminados[j]+min(x[i],f[i-j]):
                    ataques.append(i)
                    i = j
                    break
```

En esta primera parte de la función, vemos un *while* el cual, en el peor caso, puede iterar hasta n veces (una por cada minuto). Dentro del *else* hay un bucle interno que puede recorrer hasta i elementos. Por lo tanto, la complejidad de esta sección sera $O(n^2)$.

Luego, la función vuelve a realizar un ciclo de la siguiente manera:

```
    for i in range(1, n + 1):
        if i in set_ataques:
            estrategia.append("Atacar")
        else:
            estrategia.append("Cargar")
```

Este bucle recorre n elementos y consulta en un *set*, lo cual es $O(1)$. Teniendo una complejidad final de $O(n)$.

Como complejidad final de esta función, obtenemos un $O(n^2)$ debido al primer doble bucle que se haya en la misma.

3.2. Complejidad Final del Algoritmo

- Función *algoritmo(n, x, f)*: $O(n^2)$.
- Función *construir estrategia(n, eliminados,x,f)*: $O(n^2)$.

El algoritmo tiene complejidad final $O(n^2)$ en el peor caso, debido a lo explicado previamente, donde el tiempo crece cuadraticamente con el tamaño n de la entrada. En cambio, en la practica, si los valores de f y x hacen que los máximos se alcancen rápido, el tiempo puede comportarse casi linealmente $O(n)$.

3.3. Demostración empírica

Para demostrar, empíricamente, la complejidad planteada teóricamente tomamos mediciones de tiempo de ejecución del algoritmo con distintos tamaños de muestra.

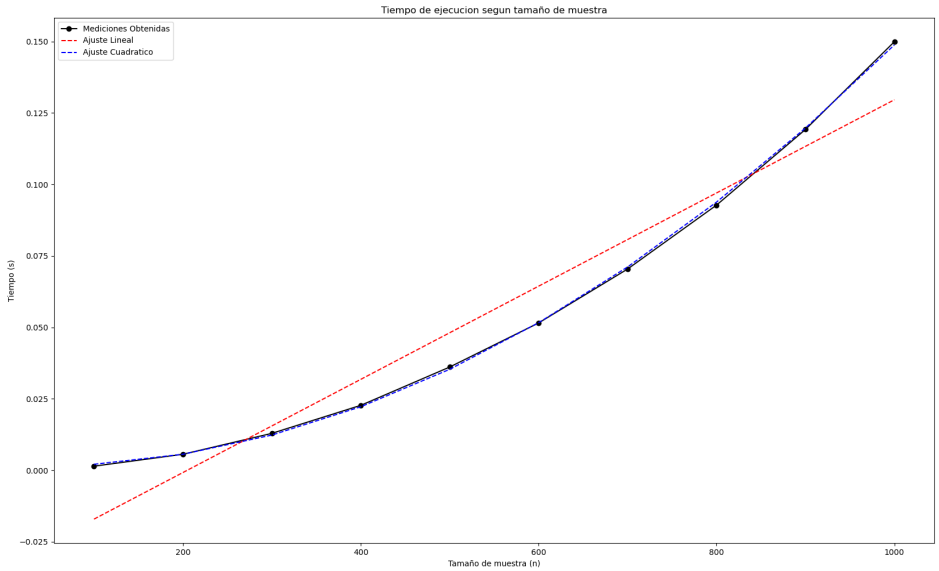

Figura 1: Gráfico del tiempo de ejecución según tamaño de muestra.

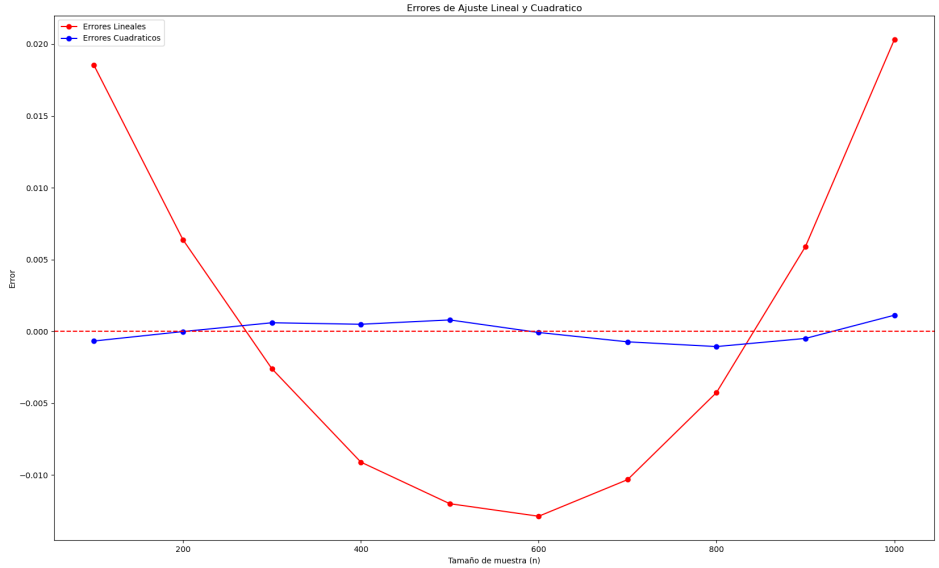

Figura 2: Gráfico de errores de los ajustes obtenidos.

Para demostrar que el algoritmo planteado es efectivamente $O(n^2)$ realizamos ajustes por cuadrados mínimos a las mediciones obtenidas, como podemos ver en la figura 1 el ajuste cuadrático ajusta casi a la perfección a las mediciones obtenidas, en contraparte el ajuste lineal no ajusta con gran precisión. Para reafirmar esto podemos ver en la figura 2 la comparativa de errores absolutos tanto del ajuste lineal como del cuadrático, podemos ver los errores cuadráticos prácticamente nulos, muy cercanos al 0 mientras que los errores lineales van de un rango $-0,015 < e < 0,020$ reafirmando la complejidad teórica previamente planteada.

3.4. Variabilidad de los datos

Variamos los datos y tomamos mediciones en busca de optimizaciones temporales del algoritmo.

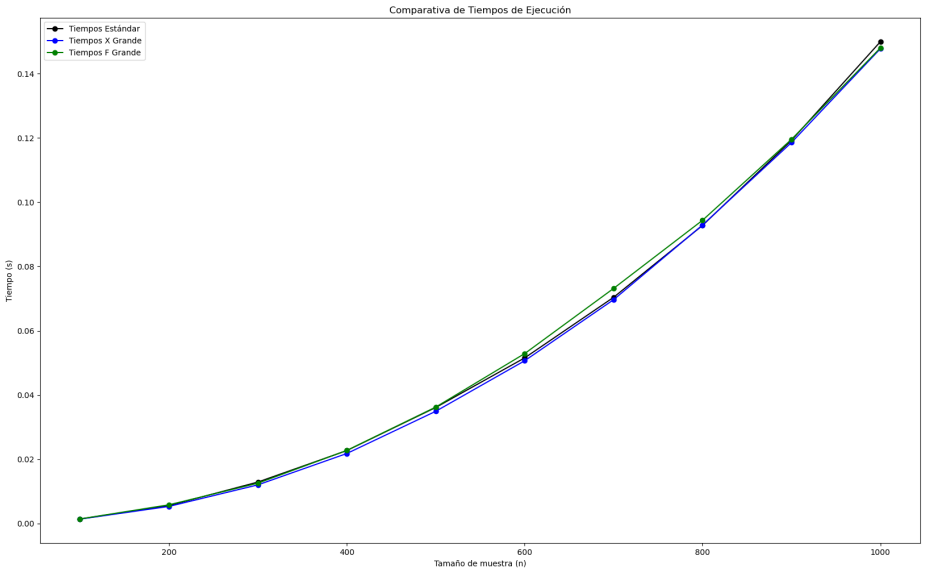

Figura 3: Gráfico de tiempos con datos variados.

Tomamos dos variaciones, dejamos f fijo y aumentamos los valores de x y el caso contrario dejando x fijo y aumentando los valores de x, como se puede ver en la figura 3 no hay variación en los tiempos obtenidos, esto nos demuestra que la variabilidad de los datos no afecta a los tiempos de ejecución del algoritmo, vamos a obtener tiempos similares independientemente de los datos recibidos.

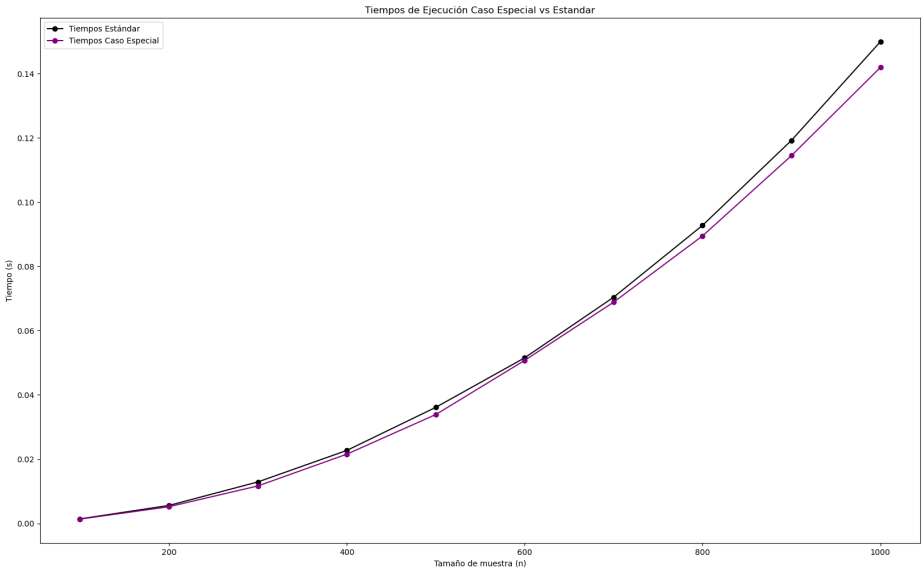

Figura 4: Gráfico de tiempos obtenidos en caso especial.

A pesar de esto encontramos un caso especial donde el algoritmo podría funcionar en tiempo lineal, si $f(1) \geq x_i \forall i$ podríamos tomar la decisión de atacar siempre y no hace falta verificar lo que sucede anteriormente ya que atacando siempre obtenemos la mayor cantidad de soldados abatidos, sin embargo deberíamos modificar el algoritmo, ya que como vemos en la figura 4 midiendo los tiempos de este caso con el algoritmo original no hay una mejora, y no funcionaria, la versión modificada, en el resto de los casos por lo que decidimos no tomar este caso como una mejora posible.

4. Conclusiones

En este trabajo se abordó la planificación óptima de ataques de los **Dai Li** mediante técnicas de *programación dinámica*, con el objetivo de **maximizar** la cantidad total de enemigos eliminados durante una serie de n minutos de ataque.

A partir del análisis del problema y la ecuación de recurrencia planteada, se diseñó un algoritmo iterativo capaz de evaluar, para cada instante, las dos decisiones posibles: atacar o cargar energía. Este enfoque permitió obtener una solución óptima global considerando todas las combinaciones de decisiones posibles hasta cada momento.

El estudio teórico de la complejidad del algoritmo demostró que tanto la función principal como la de reconstrucción presentan una complejidad temporal $O(n^2)$, debido a los bucles anidados necesarios para evaluar todos los posibles puntos de ataque anteriores.

Además, se comprobó que la variabilidad de los datos de entrada (modificando los valores de x y f) no afecta el tiempo de ejecución, lo que reafirma que la complejidad depende principalmente del tamaño de la entrada n y no de los valores específicos del problema.

En conclusión, el algoritmo desarrollado cumple correctamente con el objetivo propuesto, mostrando un comportamiento cuadrático en el caso general, con un rendimiento consistente y predecible frente a distintas configuraciones de datos, y garantizando la obtención de una estrategia óptima para los **Dai Li** en la defensa de **Ba Sing Se**.