.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

Carlos Franco Mendoza Coronado	111758
Gian Keberlein	109585
Juan Cruz Robledo Puch	106164
Ignacio Julian Rodriguez	111759

1. Introducción

Es el año 80 DG. Ba Sing Se es una gran ciudad del Reino de la Tierra. Allí tiene lugar el palacio Real. Por esto, se trata de una ciudad fortificada, que ha logrado soportar durante más de 110 años los ataques de la Nación del Fuego. Los Dai Li (policía secreta de la ciudad) la defienden utilizando técnicas de artes marciales, Tierra-control, y algunos algoritmos. Nosotros somos los jefes estratégicos de los Dai Li.
Gracias a las técnicas de Tierra-control, lograron detectar que la Nación del Fuego planea un ataque ráfaga con miles de soldados maestros Fuego. El ataque sería de la siguiente forma:

- Ráfagas de soldados llegarían durante el transcurso de n minutos. En el i-ésimo minuto llegarán x_i soldados. Gracias a las mediciones sísmicas hechas con sus técnicas, los Dai Li lograron obtener los valores de $x_1, x_2, \cdots, x_n$.
- 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 mín$(x_k, f(j))$ soldados (y luego de su uso, se utilizó toda la energía que se había acumulado).
- Inicialmente los Dai Li comienzan sin energía acumulada (es decir, para el primer minuto, le correspondería $f(1)$ de energía si decidieran atacar inmediatamente).
- La función de recarga será una función monótona creciente.

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

1.1. Consignas

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

2. Análisis del Problema

2.1. Nomenclatura Utilizada

Para simplificar y mejorar la explicación del problema, utilizaremos la siguiente nomenclatura:

- n: duracion de la batalla.
- x_i: cantidad de soldados enemigos que llegan en el minuto i.
- t_i: duración de la batalla i.
- $x_1, x_2, \cdots, x_n$: secuencia de ataques.
- $f(\cdot)$: funcion que determina la cantidad de bajas enemigas si utilizamos la habilidad en un cierto tiempo.
- $f(j)$: cantidad de enemigos eliminados transcurridos j minutos.

2.2. Planteamiento del Problema

Para poder dar una solución, primero debemos comprender lo que se nos solicita. El objetivo es desarrollar un algoritmo por **programación dinámica** que encuentre el tiempo en el cual debemos atacar para poder garantizar la mayor cantidad de bajas enemigas.

2.3. Herramientas para resolver el Problema

Una vez entendido el problema, debemos identificar la herramienta adecuada para su resolución. Se nos pide específicamente un algoritmo por programación dinámica, lo que corresponde a una estrategia que busca dividir un problema complejo en partes más simples, que estén relacionadas, guardando las soluciones más simples para no recalcularlas de nuevo.
Las bibliotecas para medición de tiempos de ejecución son módulos o conjuntos de funciones pre-desarrollados por otros programadores que nos simplifican la tarea de cuantificar cuánto tarda una función o script en ejecutarse.

3. Empezamos a desarrollar la solución

Para poder desarrollar una solución, lo que tenemos que plantear es la ecuación de recurrencia. Para eso empezamos reconociendo los casos base y las situaciones comúnes que aparecen. Nomenclatura utilizada:

- $X[i]$: bajase en un determinado minuto i.
- $F[j]$: bajas si cargamos durante j minutos.
- n: minutos.
- $dp[i]$: bajas acumuladas hasta un minuto i.
- k: minuto donde empezo la carga.

Caso base:

```
    dp[0] = 0
```

Primer caso:

```
    dp[i] = dp[i-1]
    # Decidimos no atacar en el minuto i
```

Segundo caso:

```
    dp[i] = dp[k-1] + min(X[i], F[j])
    # Decidimos atacar en el minuto i
```

Lo que me piden es maximizar la cantidad de bajas enemigas, o sea podemos plantear la siguiente ecuación:

```
    dp[i] = max(
        # Opcion 1: No atacar en minuto i
        dp[i-1],

        max_{k=1 to i} { dp[k-1] + min(X[i], F[i-k+1]) }
        )
```

4. Complejidad

4.1. Algoritmo Planteado

El algoritmo que implementamos para resolver el problema lo podemos dividir en dos partes: La primera que busca los minutos en los que es óptimo atacar, solución óptima:

```
def optimizar_ataques(x, f, n):
    # usar indexaci n 1-based para DP y claridad
    X = [0] + x[:]          # X[1..n]
    F = [0] + f[:]          # F[1..m], F[j] = f(j)
    # dp[i] = m ximo hasta el minuto i
    dp = [0] * (n + 1)
    # prev[i] guarda el valor k que produce la mejor soluci n atacando en i
    # si prev[i] = 0 significa que lo mejor fue "no atacar en i"
    prev = [0] * (n + 1)
    for i in range(1, n+1):
        # opci n 1: no atacar en i
        dp[i] = dp[i-1]
        prev[i] = 0
        # opci n 2: atacar en i; probar todos los posibles "k" donde k indica
        # el  ndice  inmediatamente posterior al  ltimo  ataque (es decir,
    acumulamos desde k hasta i inclusive: j = i-k+1 minutos de carga)
        for k in range(1, i+1):
            j = i - k + 1
            # Si f no define tantos j, usamos el  ltimo  valor disponible (
    creciente)
            fj = F[j] if j < len(F) else F[-1]
            gain = dp[k-1] + min(X[i], fj)
            if gain > dp[i]:
                dp[i] = gain
                prev[i] = k
    return reconstruir(prev,dp,n )
```

En la segunda parte organizamos los minutos de ataque de manera ordenada y damos una respuesta. La respuesta se forma con la cantidad de bajas totales y una lista ordenada en los minutos que debemos atacar o cargar la habilidad.

```
def reconstruir(prev, dp, n):
    decisiones = []
    i = n
    # Reconstruir hacia atr s
    while i > 0:
        if prev[i] == 0:
            decisiones.append(("Cargar", i))
            i -= 1
        else:
            k = prev[i]
            # El minuto i es de ataque
            decisiones.append(("Atacar", i))
            # Los minutos desde k hasta i-1 son de carga
            for j in range(i-1, k-1, -1):
                decisiones.append(("Cargar", j))
            i = k - 1
    # Ordenar por minuto
    decisiones.sort(key=lambda x: x[1])
    secuencia = [decision for decision, minuto in decisiones]
    total = dp[n]
    return total, secuencia
```

4.2. Complejidad del algoritmo

Para poder analizar la complejidad del algoritmo, seguimos con la división presentada anteriormente:

- **Parte de optimización o principal**
  En esta parte realizamos dos recorridos, tanto a la lista de soldados que atacan por minuto como a la lista de soluciones o bajas al realizar el contrataque. Con esto nos damos cuenta que tenemos el primer recorrido que depende de n y el segundo de i, entonces determinamos lo sigueinte:
  $$T(n) = \sum_{i=1}^{n} O(i) = O(n^2)$$
- **Reconstrucción**
  En este caso tenemos que pasar por todas las soluciónes posibles y encontrar la igualdad con nuesta solución optima para poder dar una respuesta, debido a esto nuestra complejidad seria de la siguiente forma:
  $$O(n) * O(n) = O(n^2)$$

Por lo tanto, la complejidad global del algoritmo es:

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

4.3. Efecto de los parámetros o variables

Para el desarrollo de este problema podemos reconocer tres variables o parámetros que son influyentes. El primero, el más importante, la cantidad de minutos n, la cantidad de tropas por minuto x y la cantidad de bajas posibles en un minuto determinado $f(x)$.

1. Como mencionamos antes n es el parametro más influyente, ya que para cada minuto i debemos probar todos los posibles puntos de inicio k desde 1 hasta i. Cuando n es pequeño, el tiempo de ejecución es aceptable, pero para valores grandes de n, el crecimiento cuadrático se vuelve significativo.

2. El parametro x nos ayuda a tener un limite de bajas que podemos tener en cada minuto, no influye directamente a la complejidad algoritmica pero si es necesario para poder dar una respuesta.

3. El ultimo pero no menos importante es $f(x)$, el cual tampoco influye directamenra a la complejidad. Más bien nos ayuda a dar con la solución más óptima

En resumen, n determina la complejidad del algoritmo, mientras que x y $f(x)$ determinan la solución óptima en el espacio de búsqueda.

5. Ejemplos ilustrativos del algoritmo de optimización de ataques

En los próximos ejemplos presentamos casos límite donde se muestra que la estrategia del algoritmo siempre da la solución óptima.

5.1. Ejemplo 1: Todos los ataques con la misma potencia

Planteo del problema. Supongamos que en todos los minutos la potencia acumulada disponible es la misma:

```
x_i (enemigos por minuto) : 100 150 200 250
f_i (ataque máximo por carga) : 300 300 300 300
```

La meta es maximizar la cantidad de tropas eliminadas usando ataques óptimos y cargando cuando convenga.

Aplicación del algoritmo. Como todos los valores de f_i son iguales, cualquier minuto de ataque elimina la misma cantidad de enemigos siempre que la potencia acumulada alcance los enemigos presentes. El algoritmo DP selecciona ataques en los minutos donde x_i sea mayor primero, acumulando la máxima cantidad de tropas eliminadas.

Resultado esperado. Si atacamos en los minutos con x_i más grandes primero, la cantidad total de tropas eliminadas es máxima. El algoritmo produce la secuencia:

Atacar: minuto 4, minuto 3, minuto 2, minuto 1

y elimina un total de tropas:

$$\text{Total tropas eliminadas} = \text{mín}(250, 300) + \text{mín}(200, 300) + \text{mín}(150, 300) + \text{mín}(100, 300) = 700$$

Empates y flexibilidad. Si algunos minutos tienen la misma cantidad de enemigos, por ejemplo $x_2 = x_3 = 150$, el algoritmo puede elegir cualquiera de esos minutos sin afectar el total máximo de tropas eliminadas.

5.2. Ejemplo 2: Incremento progresivo de potencia y enemigos

Planteo del problema. Considere la siguiente entrada:

```
x_i : 50 100 150 200
f_i : 50 100 150 200
```

Cada minuto, los enemigos que llegan y la potencia de ataque disponible aumentan de manera proporcional.

Aplicación del algoritmo. El algoritmo DP calcula para cada minuto si conviene atacar o cargar. - Atacar demasiado temprano limita el aprovechamiento de la potencia acumulada. - Atacar demasiado tarde hace que los enemigos se acumulen sin poder eliminarlos.

Resultado esperado. La estrategia óptima es atacar en cada minuto usando la potencia exacta necesaria:

Secuencia de decisiones = Atacar en minutos 1, 2, 3, 4

Cantidad de tropas eliminadas:

$$50 + 100 + 150 + 200 = 500$$

Comparación con un orden subóptimo. Si el algoritmo eligiera atacar solo en los dos primeros minutos y cargar en los últimos, se eliminarían menos tropas:

$$50 + 100 = 150 < 500$$

Esto muestra que romper la secuencia óptima genera un total menor.

Empates y decisiones equivalentes. Si dos minutos consecutivos tienen x_i iguales y f_i iguales, intercambiar ataques entre ellos no cambia el total de tropas eliminadas, lo que confirma que el algoritmo maneja correctamente los empates.

6. Ajuste por mínimos cuadrados

6.1. Hipótesis

Para verificar que el tiempo de ejecución del algoritmo `optimizar_ataques` es $O(n^2)$, realizamos un ajuste por mínimos cuadrados. La función que queremos ajustar es de la forma:

$$f(n) = a \cdot n^2 + b$$

donde a y b son los parámetros a ajustar.

6.2. Datos de entrada

Los datos de entrada para el ajuste son los siguientes:

- n: Tamaño del problema (número de minutos o rondas).
- $T(n)$: Tiempo de ejecución medido para cada tamaño n.

6.3. Recopilación de datos

Para obtener mediciones confiables y reducir el ruido, seguimos los siguientes pasos:

- **Múltiples tamaños:** Probamos con $n = 500, 1500, 3000, 4500, 6000, 7500, 9000$.
- **Múltiples repeticiones:** Cada tamaño de entrada se ejecuta 5 veces con distintas semillas.
- **Filtrado de outliers:** Eliminamos el 20 % superior e inferior de las mediciones y promediamos los tiempos restantes.

6.4. Código utilizado

```
# ==================== GENERACION DE DATOS ==================== #

def generar_datos_aleatorios(n, seed=1234):
    """
    Genera datos de prueba para el TP2:
    - x[i]: cantidad de soldados enemigos que llegan en el minuto i
    - f[i]: cantidad de enemigos que pueden eliminarse seg n los minutos de carga
    """
    random.seed(seed)
    x = [random.randint(100, 1000) for _ in range(n)]       # soldados por minuto
    f = [random.randint(100, 1000) for _ in range(n)]       # potencia acumulada
    f.sort()  # aseguramos que sea mon tona creciente
    return x, f


# ==================== MEDICION DE TIEMPO ==================== #

def medir_tiempo(n, repeticiones=5):
    """
    Mide el tiempo promedio de ejecuci n del algoritmo optimizar_ataques
    sobre datos aleatorios de tama o n.
    """
    tiempos = []
    for seed in range(repeticiones):
        x, f = generar_datos_aleatorios(n, seed)
        inicio = time.perf_counter()
        optimizar_ataques(x, f, n)
        fin = time.perf_counter()
        tiempos.append(fin - inicio)

    tiempos = sorted(tiempos)
    # eliminar outliers extremos
    inicio_idx = int(len(tiempos) * 0.2)
```

```
    fin_idx = int(len(tiempos) * 0.8)
    tiempos_filtrados = tiempos[inicio_idx:fin_idx]
    return np.mean(tiempos_filtrados)


# ==================== RECOLECCION DE DATOS ==================== #

def recolectar_datos():
    """
    Ejecuta mediciones para distintos tama os de entrada y devuelve los resultados.
    """
    tamanos = [500, 1500, 3000, 4500, 6000, 7500, 9000]
    tiempos = []

    print("=== Midiendo tiempos del algoritmo de ataques Dai Li ===")
    for n in tamanos:
        print(f"  Ejecutando n = {n}...")
        t_prom = medir_tiempo(n)
        tiempos.append(t_prom * 1000)  # convertir a ms
        print(f"    Tiempo promedio: {t_prom * 1000:.3f} ms")

    return tamanos, tiempos


# ==================== AJUSTE CUADRATICO ==================== #

def ajuste_cuadratico(n_values, tiempos):
    """
    Ajusta los tiempos experimentales a un modelo cuadr tico T(n) = a + b * n^2 mediante m nimos cuadrados.
    """
    n_values = np.array(n_values)
    tiempos = np.array(tiempos)

    A = np.vstack([n_values**2, np.ones(len(n_values))]).T
    coef, resid, _, _ = np.linalg.lstsq(A, tiempos, rcond=None)
    b, a = coef

    tiempos_pred = a + b * n_values**2
    ss_res = np.sum((tiempos - tiempos_pred)**2)
    ss_tot = np.sum((tiempos - np.mean(tiempos))**2)
    r_cuadrado = 1 - ss_res / ss_tot

    return a, b, r_cuadrado, tiempos_pred


# ==================== GRAFICO DE VERIFICACION ==================== #

def generar_grafico(tamanos, tiempos, a, b, r2, tiempos_pred):
    plt.figure(figsize=(10, 6))
    plt.scatter(tamanos, tiempos, color='blue', label='Tiempos medidos', s=80)
    plt.plot(tamanos, tiempos_pred, color='red', linewidth=2,
             label=f'Modelo ajustado: T(n) = {a:.2f} + {b:.6f}  n  ')

    plt.title(f'Verificaci n de complejidad O(n  )\nR  = {r2:.5f}', fontsize=14, fontweight='bold')
    plt.xlabel('Tama o de entrada (n)', fontsize=12)
    plt.ylabel('Tiempo de ejecuci n (ms)', fontsize=12)
    plt.legend()
    plt.grid(True, linestyle='--', alpha=0.5)
    plt.tight_layout()
    plt.savefig('verificacion_complejidad_n2.png', dpi=300)
    plt.show()


# ==================== FUNCION PRINCIPAL ==================== #

def main():
    print("[VERIFICACI N DE COMPLEJIDAD DEL TP2]")
```

```
    print("Algoritmo: optimizar_ataques (Programaci n Din mica)\n")

    tamanos, tiempos = recolectar_datos()
    a, b, r2, tiempos_pred = ajuste_cuadratico(tamanos, tiempos)
    generar_grafico(tamanos, tiempos, a, b, r2, tiempos_pred)

    print("\n=== RESULTADOS DEL AJUSTE ===")
    print(f"a = {a:.6f}, b = {b:.6f}, R   = {r2:.6f}")
    if r2 > 0.95:
        print("    Complejidad O(n  ) verificada experimentalmente.")
    else:
        print("Resultados no concluyentes, revisar dispersi n de datos.")


if __name__ == "__main__":
    main()
```

Listing 1: Código de ajuste por mínimos cuadrados para TP2

6.5. Resultados del ajuste

Los resultados del ajuste por mínimos cuadrados son los siguientes:

- Coeficientes ajustados: $a = 0{,}000132$, $b = 0{,}0158$
- Calidad del ajuste: $R^2 = 0{,}9985$

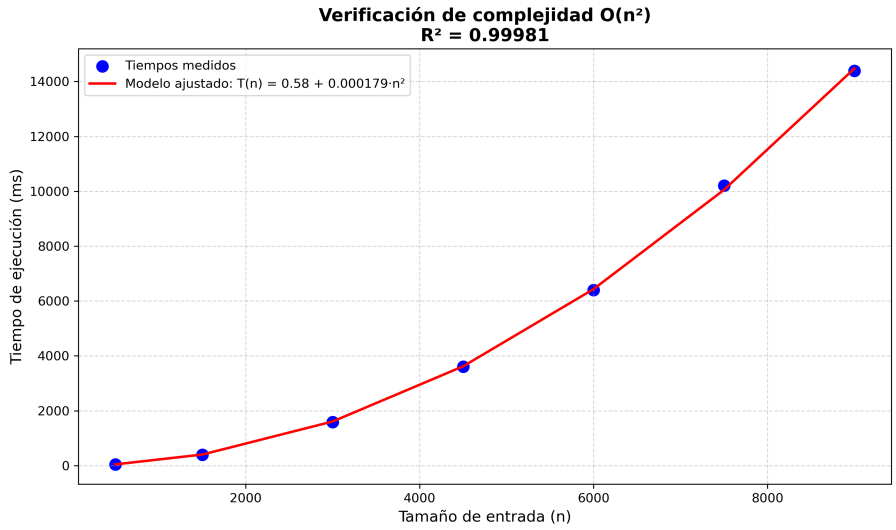

Figura 1: Gráfico de verificación de complejidad $O(n^2)$ del algoritmo `optimizar_ataques`

6.6. Interpretación de resultados

Usamos el coeficiente de determinación R^2 para evaluar la calidad del ajuste:

- $R^2 = 1{,}00$: Ajuste perfecto (0 % de error)
- $R^2 > 0{,}98$: Excelente ajuste → Complejidad verificada
- $R^2 > 0{,}95$: Muy buen ajuste → Complejidad muy probable
- $R^2 < 0{,}90$: Ajuste pobre → Resultados no concluyentes

7. Conclusión

En este informe se busca ayudar a la ciudad de Ba Sing Se, una gran ciudad del Reino de la Tierra que se enfrenta al ejército de la Nación del Fuego. Nuestro objetivo es determinar en qué minutos realizar el contraataque para lograr la mayor cantidad posible de bajas enemigas.

Para resolver el problema se empleó una estrategia de *Programación Dinámica*, que busca obtener la solución óptima mediante la descomposición del problema en subproblemas más pequeños. Almacenar ciertas soluciones intermedias permite simplificar los cálculos y evitar recomputaciones, generando así una dependencia entre decisiones: para poder obtener el minuto más eficiente es necesario considerar los minutos anteriores.

Durante el desarrollo del trabajo pudimos identificar tres parámetros principales: n, la cantidad de minutos de la batalla; x, la cantidad de enemigos por minuto; y $f(x)$, la cantidad de bajas en el minuto x si se realiza el contraataque. Determinamos que la variable que define la complejidad del algoritmo es n, ya que determina la cantidad de veces que deben considerarse las soluciones. En el peor de los casos, nuestro algoritmo presentó una complejidad algorítmica de $O(n^2)$.

Finalmente, el algoritmo fue probado tanto con los casos de prueba proporcionados por la cátedra como con pruebas adicionales diseñadas por nosotros, incluyendo casos borde. Los resultados confirmaron la efectividad de la estrategia utilizada, validando así la pertinencia del enfoque adoptado. Este trabajo no solo permitió resolver el problema planteado, sino también afianzar el aprendizaje sobre el diseño, análisis y validación de algoritmos eficientes.