Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Programación Dinámica para el Reino de la Tierra

06-05-2024

Álvarez Julián 110825

Ascencio Felipe Santino 110675

Olivera Nicolás 110797

Introducción

1. Análisis del enunciado

La Nación del Fuego ha planeado un ataque con ráfagas de "x"soldados enemigos durante "n minutos", lo que requiere una respuesta eficiente y dinámica por parte de los Dai Li para maximizar la eliminación de enemigos y defender la ciudad de manera efectiva. Para esto, se debe optimizar la estrategia de ataques y recargas de energía de los Dai Li, aprovechando al máximo sus habilidades de Tierra-control y la función de recarga f(j).

Tenemos una secuencia de llegadas de enemigos representada por "X1, X2,. . . ,Xn".

La cantidad de soldados enemigos que llegan en cada minuto varía, lo que implica la necesidad de adaptar la estrategia de ataques. Es decir, realizar una planificación dinámica para aprovechar al máximo las oportunidades de eliminación de enemigos.

Contamos tambien con una funcion de recarga "f(j)", la cual indica cuántos enemigos se pueden eliminar con un ataque después de "j"minutos de recarga acumulada.

Cuanto más tiempo se acumule la energía, mayor será la capacidad de eliminar enemigos con un ataque.

Los Dai Li deben decidir cuándo utilizar ataques de fisuras para maximizar la eliminación de enemigos, siendo que cada ataque consume la energía acumulada.

2. Análisis problema

El problema presenta un escenario estratégico donde se deben optimizar las decisiones de ataque y recarga para defender la ciudad de Ba Sing Se de manera eficiente contra el ataque de la Nación del Fuego.

Es decir, el problema se basa en determinar en que momentos es necesario realizar los ataques con la recarga de energia para maximizar la cantidad de soldados abatidos.

En este caso nos piden usar un algoritmo de "programación dinámica", el cual va a ir resolviendo problemas pequeños, y guardando los valores obtenidos, para luego ir solucionando problemas mas grandes.

En este informe se pasa a detallar el paso a paso seguido para encontrar el "Algoritmo de Programación Dinámica" que resuelva el enunciado de forma "óptima".

3. Pequeña consideración

La explicación de la distribución de los archivos de "GITHUB", y la explicación de ejecución de los programas, se encuentra en el "README" del repositorio.

Desarrollo

4. Primeros pasos

Cuando se analiza el problema de los Dai Li en Ba Sing Se, se nota que hay dos factores críticos que determinan la estrategia óptima en cada momento:

El momento actual (representado por ï"): Cada minuto representa un momento distinto en el tiempo durante el cual llegan diferentes cantidades de enemigos.

La estrategia óptima puede variar de un minuto a otro dependiendo de cuántos enemigos lleguen en ese momento.

La energía acumulada (representada por "j"): La cantidad de energía acumulada determina cuántos enemigos se pueden eliminar con un ataque de fisuras en ese momento. Cuanto mayor sea la energía acumulada, mayor será la capacidad de eliminar enemigos.

Para obtener el "Algoritmo de Programación Dinámica" óptimo que resuelve el problema, en un principio, guiándose de los problemas resueltos en el material (como Scheduling por PD o Juan el Vago), y el análisis previo de las "2 variables de interés"se planteó una resolución del problema mediante el uso de una matriz.

En la misma se iban llenando las posiciones de la siguiente forma:

- Nos paramos en la posición "x ; x"(Al comenzar es la "0 ; 0") donde claramente la mejor opción es atacar (ya que no tenemos en cuenta la existencia de "posibles minutos futuros").
- Ya sabiendo que en ese minuto en específico se va a atacar, queda ir viendo en todas las posiciones "(x - i) ; (x - i)", siendo ï"los minutos previos transcurridos y "x.^{el} minuto actual, a partir de cual minuto nos conviene empezar a çargar"para luego ."^{a}tacar."^{en} el minuto actual, teniendo en cuenta el "f(j)"para cada caso y la cantidad máxima de enemigos a abatir.
- Luego de recorrer todas las opciones posibles nos quedamos con la que maximice la cantidad de enemigos abatidos totales, y guardamos en la posición "x ; x"de la matriz la cantidad total de enemigos derrotados hasta el momento.
- Luego se completa (hacia la derecha) toda la hilera horizontal con ese valor.
- Volvemos a repetir el proceso.

Al hacer este proceso con el ejemplo de "10.txt"se llego a la siguiente matriz:

		poder de daño por minuto											
			170	312	1000	2131	2975	3026	3035	3402	3463	3496	
	↓ i \ j →	0	1	2	3	4	5	6	7	8	9	10	
	0	0	0	0	0	0	0	0	0	0	0	0	
	254	0	170	170	170	170	170	170	170	170	170	170	matar
	515	0		340	340	340	340	340	340	340	340	340	matar + matar
	647	0			647	647	647	647	647	647	647	647	esperar + esperar + matar
ORDAS	454	0				817	817	817	817	817	817	817	e + e + m + m
	126	0					943	987	987	987	987	987	e + e + m + m + m
	406	0						1129	1129	1129	1129	1129	e + e + m + m + e + m
	69	0							1198	1198	1198	1198	e + e + m + m + e + m + m
	48	0								1246	1246	1246	e + e + m + m + e + m + m + m
	781	0									1910	1910	e + e + m + m + e + m + e + e + m
	920	0										2118	e + e + m + m + e + m + m + e + e + m

Figura 1: Matriz a analizar.

Esta matriz nos aseguraba que en la posición "(n-1) ; (n-1)"siendo "n"la cantidad de minutos totales, obtendríamos el valor máximo de rivales que se podían derrotar, pero todavía faltaba obtener el orden seguido.

El primer planteo era, ver como se debían rellenar los espacios restantes, de forma tal que luego con un recorrido específico (y detectando ciertos patrones en los valores, similar a otros problemas vistos) se logre obtener el orden de ."taques/cargas"seguido para obtener el óptimo que nos brindó el valor máximo de rivales eliminados.

Pero es en este punto que se detectó una forma mas fácil de obtener el orden sin tener que recorrer la matriz con algún patrón en específico.

5. ¿Por qué un vector y no una matriz?)

La forma más fácil de obtener tanto el máximo de enemigos eliminados, como el orden seguido, es utilizar un vector que contenga una variable numérica que guarde el máximo de enemigos eliminados por minuto y, al llegar al final de todos los minutos, conseguir mediante a todos los óptimos el orden seguido para la solución final.

Esto genera que solo se necesite (en memoria) una lista con todos los "óptimos parcialesçonseguidos, para luego en base al "óptimo globalçonseguir el camino recorrido y crear la otra lista con la estrategia seguida.

Consiguiendo que estas 2 listas conseguidas ocupen mucho menos espacio en memoria que una matriz entera de valores númericos.

6. Complejidad espacial del algoritmo

Este algoritmo planteado tiene una gran ventaja, y esta es la optimización en cuanto al espacio en memoria que vamos a necesitar para obtener la solución óptima.

Esto se da porque en definitiva lo único que vamos a utilizar para obtener el orden seguido va a ser un recorrido que crea una lista con el camino seguido en relación a los "óptimos parcialesçonseguidos, y no en relación al recorrido específico por una matriz.

Esto reduce mucho la complejidad espacial, ya que por cada minuto transcurrido utilizando la matriz, tendríamos "nçasillas, generando que tanto para el minuto "1çomo el minuto "n"la cantidad de casillas sea "n"siempre, cosa que es una pérdida de espacio en memoria a la hora de resolver nuestro problema de forma óptima.

7. Ecuación de recurrencia

Ahora, ya con la idea clara de la estructura necesaria para la implementación del algoritmo "óptimo", se desarrollo la ecuación de recurrencia.

La ecuación obtenida es la siguiente:

$$\text{Opt}(i) = \max_{j=1}^{i} \{\text{Opt}(j) + \text{mín}\{x_i, f(i-j)\}\}$$

Donde :

Opt(i) representa la cantidad máxima de enemigos que se pueden eliminar hasta el i-ésimo minuto.

El término "máx$_{j=1}^{i}$ïndica que estamos buscando el máximo entre todas las posibles combinaciones de ataques y cargas anteriores hasta el i-ésimo minuto.

Opt(j) representa la cantidad máxima de enemigos que se pueden eliminar hasta el j-ésimo minuto, es decir, antes del ataque en el i-ésimo minuto.

$minxi, f(i-j)$ Si el ataque se realiza en el $i-ésimo$ minuto, se considera la cantidad de enemigos que llegan en ese minuto (Xi) y la recarga acumulada hasta el $j-ésimo$ minuto $f(ij)$ que se usa. Tomamos el mínimo entre estos dos valores para asegurarnos de que el ataque no elimine más enemigos de los que llegan en ese minuto o de los que la recarga acumulada puede eliminar.

En cada minuto i, los Dai Li deben tomar una decisión sobre si usar o no un ataque de fisuras para eliminar los "Xi.^{en}emigos que llegan en ese minuto. Si deciden utilizar la función de recarga "f(j)", deben determinar cuánto tiempo han acumulado energía (representada por "j") y qué cantidad de enemigos se pueden eliminar utilizando esa energía acumulada. La decisión óptima en el minuto i implica evaluar si usar "f(j)"para eliminar los enemigos en ese minuto o si es mejor esperar y acumular más energía para futuros ataques.

Entonces, la ecuación de recurrencia nos dice que para calcular la cantidad máxima de enemigos que se pueden eliminar hasta el $i-ésimo$ minuto, consideramos todas las posibles recargas anteriores (Opt(j)) y seleccionamos aquella que maximice la cantidad de enemigos eliminados, sumando la cantidad de enemigos que se pueden eliminar con el ataque en el $i-ésimo$ minuto considerando la recarga acumulada hasta el $j-ésimo$ minuto "$f(ij)$".

8. Demostración de optimalidad

Luego de todo lo analizado es bastante fácil ver porque el algoritmo que se va a implementar es óptimo.

El mismo lo que va haciendo es resolver problemas pequeños (resolviendo de a "1"minuto a la vez), guarda los resultados obtenidos (rescataremos esto mas adelante en la justificación acerca de porque es un algoritmo de "Programación Dinámica").

Y luego, con el pasar de los turnos va tomando las decisiones acerca de cuando cargar y cuando atacar en base a la cantidad de "poder acumulado"desde el minuto que se empieza a cargar hasta el actual, y la cantidad de enemigos a abatir en ese minuto (pasando por todos los minutos previos).

De esta manera logra por cada minuto ir consiguiendo un "óptimo temporal", que va a ir variando hasta llegar al último minuto en el que se va a convertir (por no existir mas ataques del reino enemigo) en el "óptimo global", y también, la solución de nuestro problema.

Como particularidad, puede pasar que en el algoritmo se de mas de "1.^{o}rden posible de .^{a}taques/cargas"que también nos de el valor óptimo de enemigos eliminados.

Este orden variará en relación a como se tomen los casos bordes del algoritmo, pero no variará en la optimalidad de la resolución al problema en si, así que no hay por que preocuparse.

9. Código del algoritmo óptimo

```
CARGAR = "Cargar"
ATACAR = "Atacar"
PRIMER_RENGLON = 2

#PRE: - .
#POST: Hace la lectura y el parseo del archivo para devolver los datos necesarios.
def carga(archivo):
    try:
        nombre_archivo = archivo + ".txt"
        with open(nombre_archivo) as archivo:
            lineas = archivo.readlines()
            mitad = (len(lineas) - PRIMER_RENGLON)/2
            ordas = []
            poder = []
            for linea in lineas[PRIMER_RENGLON:]: #O(n)
                if mitad > 0:
                    datos = linea.strip().split(",")
                    ordas.append(int(datos[0]))
                    mitad -= 1
                else:
                    datos = linea.strip().split(",")
                    poder.append(int(datos[0]))
        return ordas, poder
    except IOError:
        print("Error al abrir el archivo")
        return None

#PRE: Ordas y poder deben estar cargados correctamente.
#POST: Devuelve un arreglo con el numero de bajas optimo para cada ataque.
def batallas(ordas, poder): #O( n  )
    bajas_optimas = [0] * len(ordas)
    bajas_optimas[0] = (min(ordas[0], poder[0]))

    for i in range(1, len(ordas)): #O(n)
        bajas_optimas[i] = (min(ordas[i], poder[i]))
        for j in range(i): #O(n)
            bajas_optimas[i] = max(bajas_optimas[i], bajas_optimas[j] + min(ordas[i
    ], poder[i -1-j]))
    estrategia = sacar_resultado(bajas_optimas, ordas, poder) #O( n  )
    return estrategia, bajas_optimas[len(bajas_optimas) - 1]

#PRE: bajas_optimas, ordas y poder deben estar cargados correctamente.
#POST: Devuelve un arreglo con la estrategia necesaria para llegar al optimo final.
def sacar_resultado(bajas_optimas, ordas, poder): #O( n  )
    estrategia = [CARGAR] * len(bajas_optimas)
    i = len(bajas_optimas) - 1

    while i > 0: #O(n)
        for j in range(i): #O(n)
            estrategia[i] = ATACAR
            bajas = bajas_optimas[i]
            if bajas == bajas_optimas[j] + min(ordas[i], poder[i - 1-j]):
                i = j
                break
        if i - 1 == j:
            break
        if i == 0:
            estrategia[i] = ATACAR
    return estrategia

def main():
    archivo = input("ingrese nombre del archivo\n")
    ordas, poder = carga(archivo)
    estrategia, bajas_tot = batallas(ordas, poder) #O( n  )
    print("Cantidad de tropas eliminadas:", bajas_tot)
    print("Estrategia:", estrategia)
```

10. Explicación del algoritmo óptimo

Se pasa a explayar todo el algoritmo, paso a paso, para luego analizar si efectivamente cumple con las condiciones de un "Algoritmo de Programación Dinámica".

Primero: Se procede a leer el archivo, parsear todos los datos y guardarlos para su posterior uso.

Segundo: Se inicializa todo el vector en "0".

Tercero: Se va recorriendo cada posición del vector y se van obteniendo los "óptimos parciales.[en] relación a los valores correspondientes al minuto actual y los anteriores (excepto en el primer turno donde solo se tiene en cuenta el minuto actual).

Cuarto: Al terminar todos los minutos, se obtiene el orden seguido hasta alcanzar el "óptimo global.[en] relación a todos los "óptimos parcialesçonseguidos.

Quinto: Se le imprime por pantalla al usuario la cantidad de enemigos eliminados y el orden seguido de .ªtaques/cargas".

11. Análisis de cada problema y cada subproblema

Antes de pasar a la justificación de porque la solución brindada es un .ªlgoritmo de Programación Dinámica", se pasa a detallar el funcionamiento del "problemaz los "subproblemas.[en] relación a el algoritmo implementado:

Una estrategia propia de la "Programación Dinámica.[es] la resolución de "subproblemas pequeños"para luego ir construyendo la respuesta a los problemas mas grandes a partir de este.

Asi que primero vamos a analizar los subproblemas:

Forma de los subproblemas: Cada subproblema se relaciona con un minuto específico durante el ataque enemigo y está determinado por la cantidad de energía acumulada para realizar el ataque de fisuras, así también por cómo dicho ataque puede aumentar la eliminación de soldados enemigos. **Esto se resume en:**

Minuto en que ocurre: Se debe evaluar la situación en ese instante específico y tomar decisiones estratégicas en consecuencia.

Cantidad de energía acumulada: La cantidad de energía acumulada disponible en cada momento es fundamental para determinar la efectividad del ataque de fisuras.

Estrategia para aumentar la eliminación de soldados enemigos: El objetivo principal de cada subproblema es encontrar la mejor manera de emplear la energía acumulada para aumentar la eliminación de soldados enemigos en cada minuto, protegiendo así la ciudad.

Por ejemplo, un subproblema podría ser decidir si utilizar la energía acumulada en el minuto 5 para eliminar una gran cantidad de soldados enemigos que llegan en ese momento, o esperar y acumular más energía para un ataque con más bajas posteriormente.

Manera en la que se componen los subproblemas para solucionar subproblemas más grandes:

La manera en la que se componen los subproblemas implica evaluar todas las posibles combinaciones de ataques de fisuras en momentos anteriores y determinar la combinación óptima que maximice la eliminación de soldados enemigos hasta el momento actual. **Incluyendo:**

Análisis de combinaciones previas: Se consideran todas las decisiones tomadas en momentos anteriores, incluyendo el uso o no del ataque de fisuras y la cantidad de energía acumulada empleada en cada caso.

Determinación de la combinación óptima: Se busca encontrar la combinación de ataques de fisuras y uso de energía acumulada que maximice la eliminación de soldados enemigos hasta el momento actual.

Por ejemplo, si en el minuto actual tenemos la opción de usar el ataque de fisuras o no, y al final decidimos usarlo, debemos determinar cuánta energía acumulada utilizar para maximizar la eliminación de soldados enemigos hasta el momento actual.

12. Justificación de Programación Dinámica

Con todo lo analizado previamente, ya se puede pasar a listar las condiciones que cumple el algoritmo implementado, y así demostrar que efectivamente estamos hablando de un .ªlgoritmo de Programación Dinámica":

- **Exploración implícita del espacio de posibles soluciones**: El algoritmo se encarga de verificar todas las posibles combinaciones de .ªtaques/cargas"que pueden brindar una mejor solución al problema.
- **Descomposición del problema en subproblemas que permitan construir las soluciones de problemas más grandes**: Como se vió en el apartado previo, cumple totalmente con esta condición.
- **Nos basamos en la intuición que nos da la Memorización para reconocer los subproblemas y utilizarlos para construir la solución**: Al ir construyendo en el vector los "óptimos parciales"vamos consiguiendo todos los posibles "óptimos útiles"para seguir construyendo mas adelante con el pasar de los minutos.
- **Una vez que tenemos todas las soluciones memorizadas, el problema está resuelto**: Al llegar al final de los minutos, ya se puede aplicar la función que obtiene el camino seguido para obtener el "óptimo globalz resolver el problema de forma óptima.
- **Nos ayuda evitando explorar un espacio exponencial de soluciones por fuerza bruta**: EL algoritmo implementado no solo cumple eso sino que va mas allá mejorando la complejidad espacial que nos podía dar una matriz para la resolución del mismo.

13. Análisis de complejidad computacional

Primero que nada, para evaluar la complejidad computacional se va a utilizar la notación "BIG O".

Como se puede observar en el "algoritmo óptimo", vemos que principalmente hay 2 funciones con complejidades computacionales a analizar (se obvia el "main" porque simplemente lo que hace es llamar a las 2 siguientes funciones a analizar):

- **batallas(ordas, poder)**: La función "batallas"lo que hace es crear un arreglo con los "óptimos parciales"de cada minuto mediante a 2 iteraciones, dependientes de "n", siendo "n"la cantidad de minutos. Su complejidad es "$O(n^2)$".
- **sacar resultado(bajas optimas, ordas, poder)**: La función "sacar resultado"lo que hace es obtener el orden seguido de .ªtaques/cargas"para obtener el "óptimo global"del algoritmo según los "óptimos parcialesçonseguidos. Esto lo hace también mediante a 2 iteraciones, dependientes de "n", siendo "n"la cantidad de minutos. Su complejidad es "$O(n^2)$".

Luego de este análisis, Podemos definir que la complejidad computacional de nuestro algoritmo óptimo es $O(n^2+n^2)$, nos quedaría $O(2n^2)$ que, en definitiva tendiendo al infinito, terminaría quedando en $O(n^2)$.

14. Testeo del algoritmo

El algoritmo fue probado tanto con las pruebas brindadas por la cátedra, como por las propias creadas y siempre cumplió con la condición pedida para el valor óptimo.

Los resultados de las pruebas propias fueron verificados mediante a tablas de Excel "Mismo método que el explicado al inicio del informez dieron el resultado óptimo.

Las pruebas propias son:

OrdaChica-PoderChico: Turnos: 10. Hordas: 2, 5, 14, 4, 12, 6, 11, 15, 21. 20. Poder: 1, 2, 6, 7, 9, 12, 20, 22, 24, 30.

Resultado esperado:

Estrategia: Cargar, Cargar, Atacar, Cargar, Cargar, Cargar, Cargar, Cargar, Cargar, Atacar Cantidad de tropas eliminadas: 26

OrdaChica-PoderGrande: Turnos: 10. Hordas: 2, 5, 7, 4, 12, 6, 9, 8, 71, 20. Poder: 880, 3182, 10600, 213771, 2975555, 30255544, 40356667, 58888884, 74465666, 888888496.

Resultado esperado:

Estrategia: Atacar, Atacar, Atacar, Atacar, Atacar, Atacar, Atacar, Atacar, Atacar, Atacar Cantidad de tropas eliminadas: 144

OrdaGrande-PoderChico: Turnos: 10. Hordas: 246567, 543263, 144643, 4422, 1244, 62345, 11333, 15222, 21325, 2202554. Poder: 1, 2, 6, 7, 9, 12, 20, 22, 24, 30.

Resultado esperado:

Estrategia: Cargar, Cargar, Cargar, Cargar, Cargar, Cargar, Cargar, Cargar, Cargar, Atacar Cantidad de tropas eliminadas: 30

OrdaGrande-PoderGrande: Turnos: 10. Hordas: 223334, 555674, 14334566, 433465, 12354, 646333, 1133456, 1536, 21333666, 20333. Poder: 45565, 58777, 90676, 755567, 955555, 1266878, 2088888, 22644436, 4455532354, 30352523353.

Resultado esperado:

Estrategia: Atacar, Cargar, Cargar, Cargar, Cargar, Cargar, Cargar, Cargar, Atacar, Atacar Cantidad de tropas eliminadas: 21399564

Con las pruebas propias se busco llegar a los casos bordes de la complejidad computacional del algoritmo, consiguiendo que con el test de .ºrdaGrande-PoderChico"se consiguiera hacer que solo se ataque en un turno y se cargue en todo el resto. Por otro lado, con el test de .ºrdaChica-PoderGrande"se consiguió que se ataque en todos los turnos.

Teniendo asi, con estas 2 pruebas en específico, bordes opuestos de comportamiento del algoritmo implementado. Para luego, en la parte de los gráficos, ver si se perciben (o no) cambios en la complejidad computacional.

15. Código del algoritmo de los gráficos

```
import time
import math
import matplotlib.pyplot as plt

CARGAR = "Cargar"
ATACAR = "Atacar"
PRIMER_RENGLON = 2

def carga(archivo):
    try:
        nombre_archivo = archivo + ".txt"
        with open(nombre_archivo) as archivo:
            lineas = archivo.readlines()
            mitad = (len(lineas) -2 )//2
            minutos = lineas[1]
            ordas = []
            poder = []
            for linea in lineas[2:]: #O(n)
                if mitad > 0:
                    datos = linea.strip().split(",")
                    ordas.append(int(datos[0]))
                    mitad -= 1
                else:
                    datos = linea.strip().split(",")
                    poder.append(int(datos[0]))


        return ordas, poder, minutos
    except IOError:
        print("Error al abrir el archivo")
        return None

#PRE: Ordas y poder deben estar cargados correctamente.
#POST: Devuelve un arreglo con el numero de bajas optimo para cada ataque.
def batallas(ordas, poder): #O(n )
    bajas_optimas = [0] * len(ordas)
    bajas_optimas[0] = (min(ordas[0], poder[0]))

    for i in range(1, len(ordas)): #O(n)
        bajas_optimas[i] = (min(ordas[i], poder[i]))
        for j in range(i): #O(n)
            bajas_optimas[i] = max(bajas_optimas[i], bajas_optimas[j] + min(ordas[i
    ], poder[i -1-j]))
    estrategia = sacar_resultado(bajas_optimas, ordas, poder) #O(n )
    return estrategia, bajas_optimas[len(bajas_optimas) - 1]


#PRE: bajas_optimas, ordas y poder deben estar cargados correctamente.
#POST: Devuelve un arreglo con la estrategia necesaria para llegar al optimo final.
def sacar_resultado(bajas_optimas, ordas, poder): #O(n )
    estrategia = [CARGAR] * len(bajas_optimas)
    i = len(bajas_optimas) - 1

    while i > 0: #O(n)
        for j in range(i): #O(n)
            estrategia[i] = ATACAR
            bajas = bajas_optimas[i]
            if bajas == bajas_optimas[j] + min(ordas[i], poder[i - 1-j]):
                i = j
                break
        if i - 1 == j:
            break
        if i == 0:
            estrategia[i] = ATACAR
    return estrategia


```

```
archivo = input("ingrese nombre del archivo\n")
ordas, poder, minutos = carga(archivo)
# Realizar mediciones de tiempos
n = len(ordas)
tiempos = []
for i in range(1, n+1):
    inicio = time.time()
    cantidad,estrategia =  batallas(ordas[:i], poder[:i])
    fin = time.time()
    tiempos.append(fin - inicio)

#Graficar los tiempos de ejecuci n
escala = 10**7
tiempos_escalados = [t * escala for t in tiempos]
n_cuadrado = [2*(i**2) for i in range(1, n + 1)]


plt.figure(figsize=(12, 8))
plt.plot(range(1, n+1), tiempos_escalados, marker='o')
plt.plot(range(1, n + 1), n_cuadrado, linestyle='--', label='O(2*n^2)')
plt.xlabel('N mero de ordas')
plt.ylabel('Tiempo de ejecuci n (s)')
plt.title('Tiempo de ejecuci n del algoritmo')
plt.grid(True)
plt.show()
```

16. Explicación del programa de gráficos

Se utilizó la biblioteca "matplotlib"para realizar unos gráficos que nos muestren las curvas de tendencia de complejidad computacional que siguen las pruebas realizadas.

En este caso, además de las curvas de tendencia de cada prueba en particular (color azul), se trazó una curva punteada referencial (color rojo) de ecuación "$2n^2$" (Curva calculada teóricamente en el inciso anterior), para poder corroborar si efectivamente la complejidad computacional estudiada responde a la detectada con la ejecución del programa.

17. Explicación de gráficos

A continuación se pasa a exhibir los gráficos obtenidos con el programa antes mencionado, para luego hacer el análisis práctico de la complejidad computacional y ver si los datos se corroboran con lo analizado previamente.

18. Gráficos

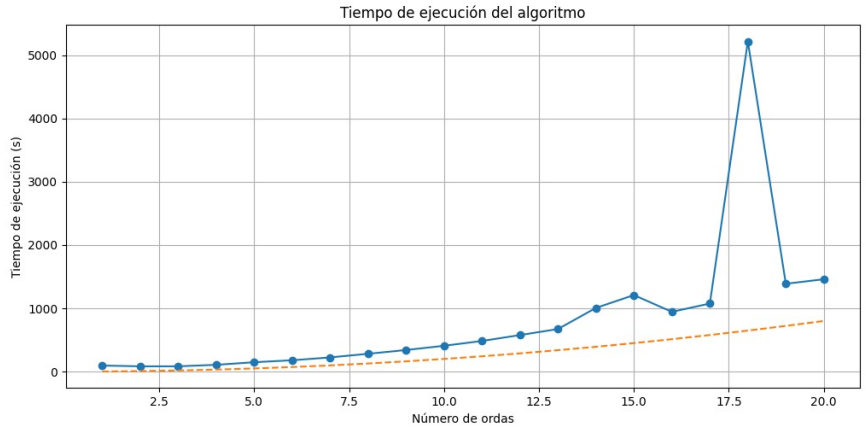

Figura 2: Gráfico de prueba de tiempo de ejecución de los 20 elementos.

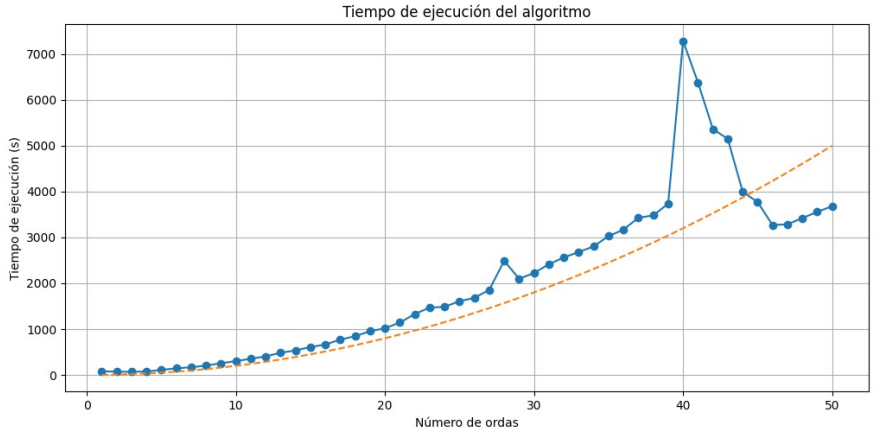

Figura 3: Gráfico de prueba de tiempo de ejecución de los 50 elementos.

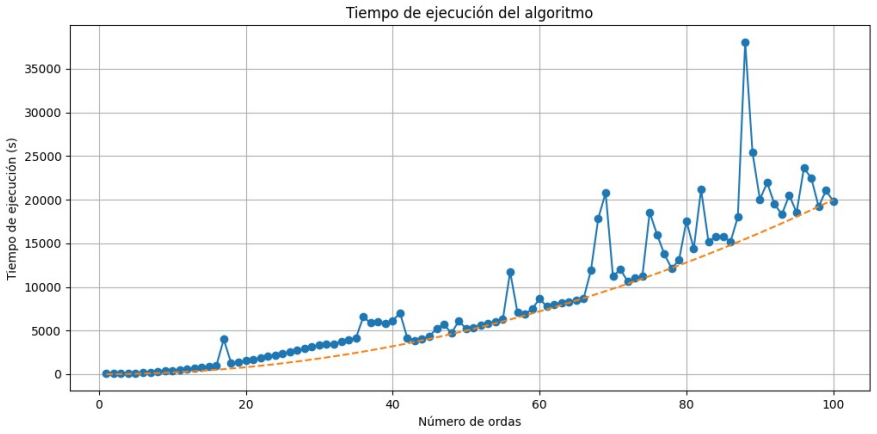

Figura 4: Gráfico de prueba de tiempo de ejecución de los 100 elementos.

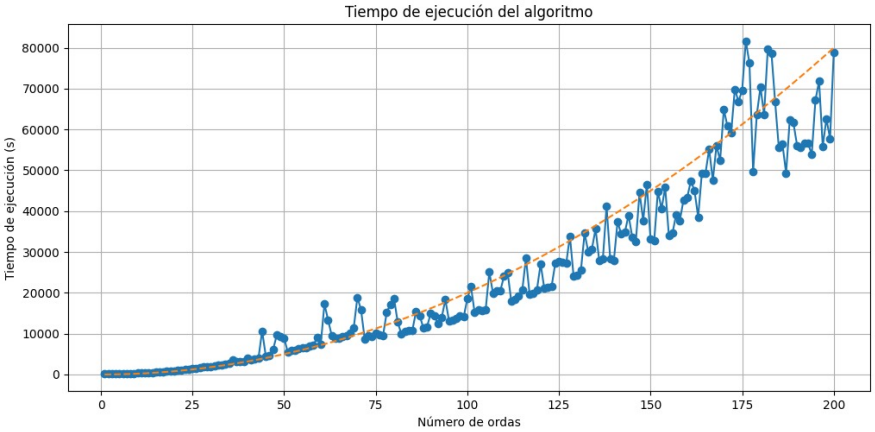

Figura 5: Gráfico de prueba de tiempo de ejecución de los 200 elementos.

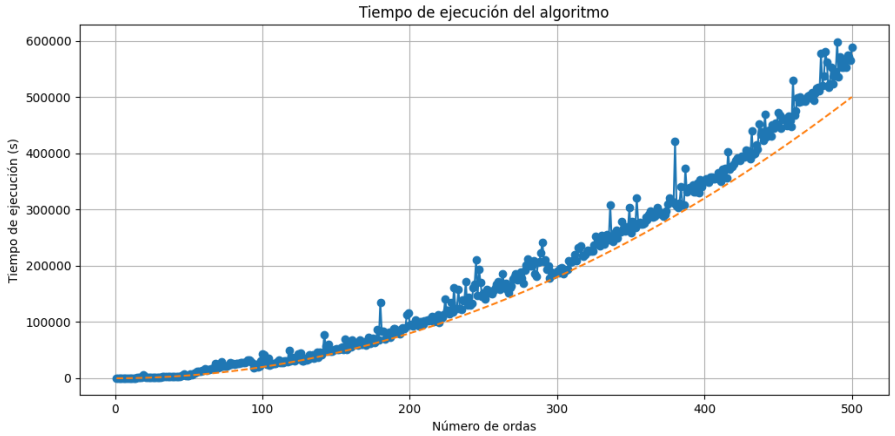

Figura 6: Gráfico de prueba de tiempo de ejecución de los 500 elementos.

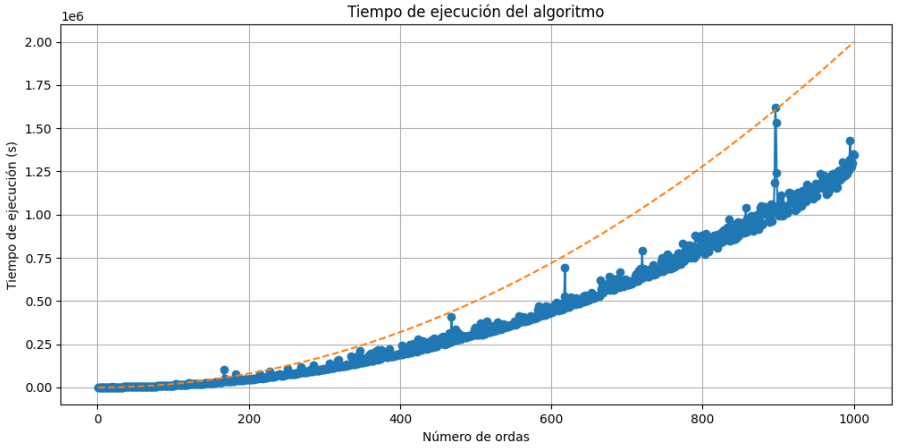

Figura 7: Gráfico de prueba de tiempo de ejecución de los 1000 elementos.

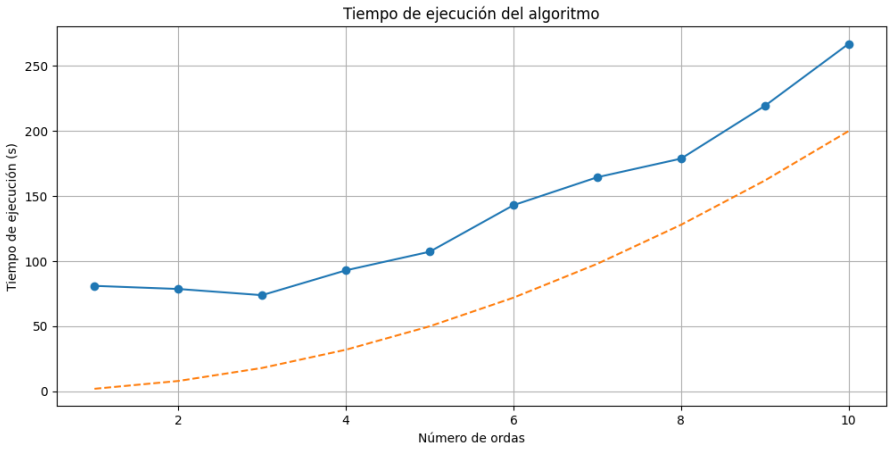

Figura 8: Gráfico de prueba de tiempo de ejecución del caso .ºrdaChica-PoderChico".

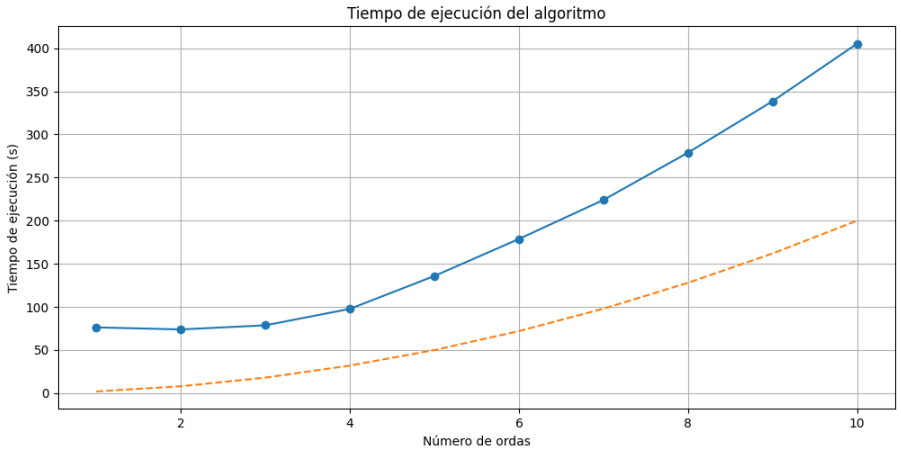

Figura 9: Gráfico de prueba de tiempo de ejecución del caso .ºrdaChica-PoderGrande".

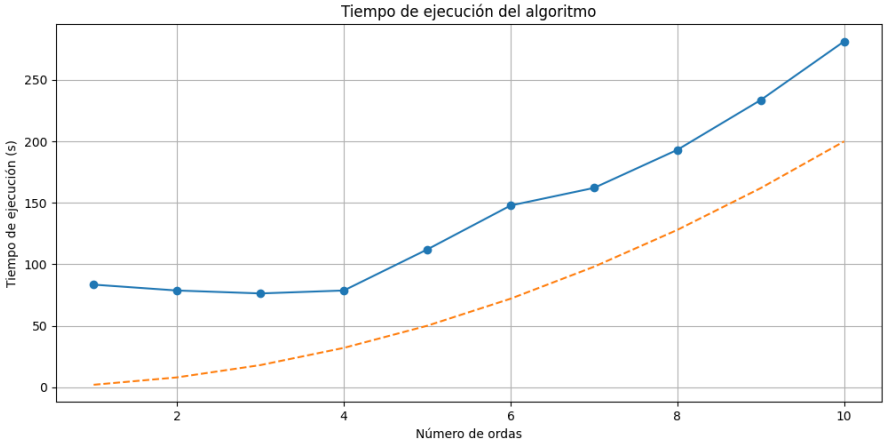

Figura 10: Gráfico de prueba de tiempo de ejecución del caso .ºrdaGrande-PoderChico".

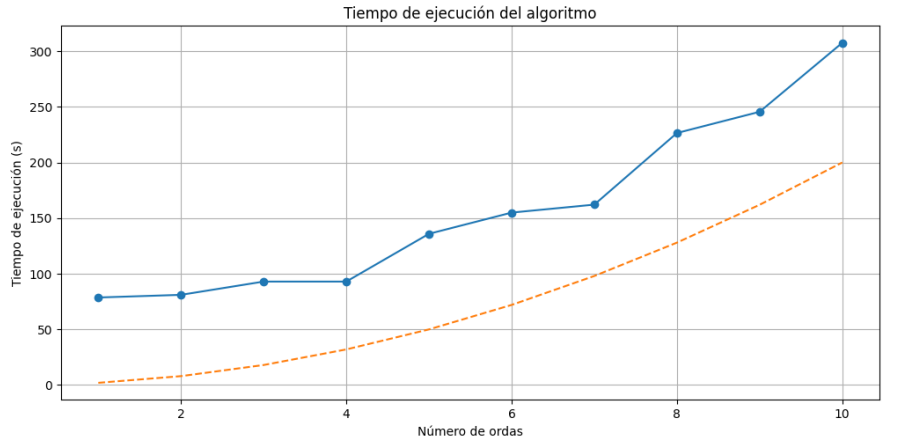

Figura 11: Gráfico de prueba de tiempo de ejecución del caso .ºrdaGrande-PoderGrande".

19. Interpretación de los gráficos

Luego de ver todos los gráficos podemos sacar varias conclusiones, pero primero vamos a aclarar algo.

La diferencia de altura que podemos apreciar entre la curva teórica de referencia (roja) y la real del programa (azul) es por una cuestión neta de escala que se utilizó para obtener mayor claridad en los gráficos, así que lo que vamos a tener en cuenta en si es la relación y la "pendiente"(derivada de la curva) que van adquiriendo los gráficos.

Apreciaciones de los gráficos:

- **Correlación con la curva teórica**: Como podemos apreciar en todos los gráficos la curva teórica aproxima muy bien los valores que toma el programa con su funcionamiento. Si bien se ve que en programa con pocos elementos (20 - 50) la tendencia es mas errática y no tan parecida a la teórica, el resto de pruebas se comportan muy fieles a la misma (excepto saltos puntuales que se dan por cuestiones de "HARDWARE"mas que de funcionamiento del programa).
- **Continuidad de la curva**: Si bien la curva suele pegar algunos saltos repentinos (probablemente por culpa del "HARDWARE") se mantiene bastante pareja y con un crecimiento constante de forma armónica.
- **¿Es fiel a nuestro análisis teórico?**: Una vez obtenidos los gráficos podemos afirmar que "SI", si bien los gráficos tienen sus pequeñas variaciones (mas perceptible en las pruebas pequeñas) se ve que la tendencia de la curva calculada teóricamente es respetada por lo que podemos afirmar que tanto de manera teórica como práctica la Çomplejidad Computacional Temporal"del algoritmo es "$O(n^2)$".

20. ¿Los valores de entrada afectan a la optimalidad y la complejidad computacional esperada?

En cuanto a complejidad computacional:

Luego de un análisis, con pruebas propias, donde se ponían valores específicos de entrada, buscando tener casos bordes de tamaños muy grandes y muy pequeños de cada uno, se dio lo siguiente:

Si bien desde la parte analítica se puede apreciar que si se forza el caso donde siempre toque cargar el ataque, hasta el minuto actual donde si o si se debe atacar la complejidad computacional debería de variar al caso en donde todos los turnos se ataca.

Al ejecutarlo de forma práctica y ver su rendimiento con los gráficos no se pudo observar algún cambio aparente y/o relevante para la complejidad o la optimalidad del algoritmo planteado.

En todos los casos siempre se llego al mejor resultado posible, y con la complejidad teórica esperada.

Así que, si bien de forma teórica (como se vió en el análisis) está bastante claro que los valores de entrada pueden modificar la complejidad computacional esperada, a la hora de rendir no se llego a percibir dicho cambio.

En cuanto a la optimalidad:

La optimalidad no se ve para nada afectada por ningún tipo de dato de entrada, por lo que da exactamente lo mismo que le ingresemos, ya que siempre va a devolver el valor óptimo al problema (siempre y cuando se respeten los tipos de los datos y el orden establecido de antemano).

Conclusión

21. Conclusiones

Al final del trabajo se consiguió completar todos los requisitos del enunciado, y se pudo obtener resultados exitosos de las hipótesis y propuestas que se fueron desarrollando a lo largo del trabajo.

Algunas de las ideas principales que se lograron completar son:

- Se llegó a la resolución del problema mediante a un “Algoritmo de Programación Dinámica” óptimo.
- Se pudo verificar la optimalidad del algoritmo de forma analítica.
- Se realizó un código e implementación total del algoritmo pedido en formato funcional.
- Se probó que los valores de entrada no influyen en la complejidad computacional y tampoco en la optimalidad del algoritmo.
- - Se verificó mediante a gráficos la complejidad teórica, y la tendencia del gráfico a respetar el original de una función “$O(2n^2)$”.

Para concluir, se cumplió con todas las expectativas iniciales y se pudo resolver el problema de forma óptima.