

Teoría de algoritmos

(75.29) Curso Buchwald - Genender

**Trabajo Práctico 2**

**Programación Dinámica para el Reino de la Tierra**

6 de Mayo de 2024

**Integrantes:**

* Matias Vazquez Morales (111083)
* ~~Scarlet Mendoza (108524)~~
* Nestor Fabian Palavecino Arnold (108244)

# **Introducción**

En este informe, presentaremos

1. **Algoritmo de programación dinámica para encontrar el listado óptimo de las batallas a pelear**

En esta sección se encontrará un análisis completo sobre

**Resolución de las consignas**

**1)**

**2) el algoritmo planteado se encuentra en el repositorio de github** [**https://github.com/NestorPala/TDA-tp2/tree/main**](https://github.com/NestorPala/TDA-tp2/tree/main)

def main():

    if len(sys.argv) != 2:

        print("Ejemplo de uso: python3 tp2.py 500.txt")

        return

    path = sys.argv[1]

    filename = path.split(".")[0] + ".txt"

    enemigos\_eliminados, orden\_recargar\_atacar = tp2\_batallas\_solver(path)

    escribir\_resultados(filename, enemigos\_eliminados, orden\_recargar\_atacar)

    print("\nArchivo procesado con éxito!")

    print(f"Los resultados se encuentran en el archivo solved\_{filename}")

la funcion main(), comprueba si el numero de argumentos de la linea de comandos es correcto, si no da un mensaje de como debe ser su uso, extrae de estos argumentos el archivo .txt que se quiere leer, hace el llamado a la funcion tp2\_batallas\_solver y a la funcion escribir\_resultados.

La complejidad de esta funcion depende de estas 2 funciones, sin embargo si no las tenemos en cuenta la funcion main() tiene una complejidad constante.

def tp2\_batallas\_solver(file\_path):

    with open(file\_path, 'r') as file:

        lines = file.readlines()[1:]

    n = int(lines[0].strip())

    x\_values = [int(x.strip()) for x in lines[1:n+1]]

    function\_values = [int(x.strip()) for x in lines[n+1:]]

    return tp2(x\_values, function\_values)

abre el archivo en la ruta proporcionada en modo de lectura, lee todas las lienas del archivo omitiendo la primer linea, crea 2 listas una con los valores de X que son los enemigos y otra con el valor de las funciones, hace el llamado a la funcion tp2, pasando por parametro estas 2 listas y devuelve su resultado.

En cuanto a la complejidad es O(n) donde en es el numero de lineas del archivo, sin embargo al tener otra funcion dentro de esta, depende de esa funcion.

def escribir\_resultados(filename, enemigos\_eliminados, orden\_recargar\_atacar):

    with open(f"solved\_{filename}", 'w+') as resultados\_file:

        resultados\_file.write(filename)

        estrategia = ""

        for i in range(len(orden\_recargar\_atacar)):

            orden = orden\_recargar\_atacar[i]

            estrategia += orden

            if i < len(orden\_recargar\_atacar) - 1:

                estrategia += ", "

        resultados\_file.write("\nEstrategia: " + estrategia)

        resultados\_file.write("\nCantidad de tropas eliminadas: " + str(enemigos\_eliminados))

        resultados\_file.write("\n")

Crea el archivo llamado solved\_{filename} en modo de escritura, crea una cadena estrategia que contiene la secuencia de ordenes de carga y ataque, agregando cada cadena separado por una coma, escribe la cantidad de enemigos eliminados en el archivo.

La complejidad es O(n) donde n es el numero de ordenes en orden\_recargar\_atacar

def tp2(x, f):

    memo = tp2\_dp(x, f)

    cantidad\_oleadas\_enemigos = len(x)

    enemigos\_eliminados, indices\_solucion = reconstruir\_solucion(memo)

    orden\_recargar\_atacar = beautify\_solucion(indices\_solucion, cantidad\_oleadas\_enemigos)

    return enemigos\_eliminados, orden\_recargar\_atacar

Hace el llamado a la funcion tp2\_dp con las listas x y f como argumentos, calcula la cantidad de oleadas de enemigos, que es la longitud de la lista x, llama a las funciones recontruir\_solucion, beutify\_solucion y devuelve los enemigos eliminados y el orden de recargar y atacar.

La complejidad de la funcion sin tener en cuenta a las otras 2 funciones llamadas es constante, ya que no tiene bucles ni estructuras de datos que crezcan con el tamaño de entrada, pero como hace el llamado a otras 2 funciones, la complejidad depende de estas.

def tp2\_dp(lista\_xi, lista\_fj):

    n = len(lista\_xi)

    OPT = inicializar\_memo(n)

    maximos\_batallas\_anteriores = []

    for minuto\_actual in range(1, n+1):

        maximo\_batalla\_actual = -math.inf

        for minuto\_origen in range(n):

            if minuto\_actual <= minuto\_origen:

                continue

            if minuto\_origen == 0:

                maximo\_batallas\_anteriores = 0

            else:

                maximo\_batallas\_anteriores = maximos\_batallas\_anteriores[minuto\_origen - 1]

            minuto\_actual\_ = minuto\_actual - 1

            j = minuto\_actual\_ - minuto\_origen

            ataque\_actual = min(lista\_xi[minuto\_actual\_], lista\_fj[j])

            # ecuacion de recurrencia: OPT[i][j] = max(OPT[i-1][k] ∀ k ∈ {0,...,j-1}) + min(X[i], f(j))

            # X = lista de cantidad enemigos en el minuto i ("lista\_xi")

            # f = función de recarga ("lista\_fj")

            abatidos\_batalla\_actual = maximo\_batallas\_anteriores + ataque\_actual

            OPT[minuto\_actual\_][minuto\_origen] = abatidos\_batalla\_actual

            # optimización O(n^3) -> O(n^2):

            # guardo el valor de la rama de valor máximo para cada minuto

            # al mismo tiempo que proceso los minutos en la matriz

            if abatidos\_batalla\_actual > maximo\_batalla\_actual:

                maximo\_batalla\_actual = abatidos\_batalla\_actual

        maximos\_batallas\_anteriores.append(maximo\_batalla\_actual)

    return OPT

inicializa una matriz OPT y una lista maximos\_batallas\_anteriores, itera sobre cada minuto desde 1 hasta n que es la longitud de las listas xi y fj, para cada minuto itera sobre los minutos anteriores, calcula el ataque actual como el minimo entre el valor actual en lista\_xi y el correspondiente en lista\_fj, actualiza la matriz OPT con el maximo de las batallas anteriores mas el ataque actual y devuelve esta matriz OPT.

La complejidad de esta funcion en O(n^2) donde n es la longitud de lista\_xi y lista\_fj, esto porque son 2 bucles anidados que recorren hasta n elementos

def reconstruir\_solucion(memo):

    enemigos\_eliminados = max(memo[len(memo) - 1])

    solucion = [ len(memo), ]

    indice = len(memo)

    while indice > 0:

        indice = indice\_elemento\_maximo(memo[indice - 1])

        if indice > 0:

            solucion.append(indice)

    return enemigos\_eliminados, list(reversed(solucion))

encuentra el numero maximo de enemigos eliminados, que es el maximo valor en la ultima fila de la matriz memo, inicializa una lista solucion con el tamaño de la matriz memo, comienza un bucle desde el final de la matriz hasta el principio, en cada iteracion, encuentra el indice del elemento maximo en la fila actual de la matriz usando la funcion indice\_elemento\_maximo(), si el indice es mayor que 0, lo agrega a la lista solucion, y devuelve el numero maximo de enemigos eliminados y la lista solucion en orden inverso.

La complejidad de esta funcion es O(n), donde n es la longitud de memo, esto es porque la recorre desde el final hasta el principio, pero al hacer el llamado a la funcion indice\_elemento\_maximo(), la complejidad va a depender de esa funcion.

def beautify\_solucion(indices\_solucion, cantidad\_oleadas\_enemigos):

    beautified = []

    for i in range(1, cantidad\_oleadas\_enemigos + 1):

        if i in indices\_solucion:

            beautified.append("Atacar")

        else:

            beautified.append("Cargar")

    return beautified

inicializa una lista vacia beautified, itera sobre cada numero desde 1 hasta cantidad\_oleadas\_enemigos, en cada iteracion, comprueba si el numero actual esta en la lista indices\_solucion, si el numero esta, agrega a la cadena “Atacar” a la lista beutified, si no esta agrega a la cadena “Cargar”, devuelve la lista beautified.

La complejidad de esta funcion es O(n^2), donde n es la cantidad de oleadades de enemigos, esto es porque el bucle recorre desde 1 hasta cantidad\_oleadas\_enemigos y la operación in que se realiza en cada iteracion tiene una complejidad O(n) en el peor de los casos.

def indice\_elemento\_maximo(lista):

    maximo = lista[0]

    indice\_maximo = 0

    for i in range(len(lista)):

        if lista[i] > maximo:

            maximo = lista[i]

            indice\_maximo = i

    return indice\_maximo

inicializa maximo con el primer elemento de la lista y indice\_maximo con 0, itera sobre cada elemento en la lista, cada iteracion, si el elemento actual es mayor que maximo, actualiza maximo con el elemento actual y indice\_maximo con el indice actual, devuelve indice\_maximo.

La complejidad de esta funcion es O(n), donde n es la longitud de la lista.

def inicializar\_memo(n):

    memo = []

    z = -1

    for i in range(n):

        memo.append([])

        z += 1

        for j in range(n):

            memo[z].append(0)

    return memo

inicializa una lista vacia memo y una variable z con -1, itera sobre cada numero desde 0 hasta n-1, en cada iteracion, agrega una nueva lista vacia a memo e incrementa z en 1, despues en un bucle anidado, agrega 0 a la lista en la posicion z de memo n veces, y devuelve memo.

La complejidad de esta funcion es O(n^2) donde n es el argumento de entrada de la funcion, esto es por los 2 bucles anidados que recorren hasta n elementos.

En conclusion la complejidad total del algortimo es O(n^2).

La variabilidad de los valores de las llegadas de enemigos y recargas no afecta directamente a la complejidad del algoritmo, ya que esta se determina por el numero de operaciones que tiene que realizar y no por los valoores especificos de estos.

La complejidad del algoritmo es O(n^2) donde n es la cantidad de oleadas de enemigos, esto se mantiene independientemente de los valores de las llegadas de enemigos y recargas. Sin embargo si los valores son muy grandes, las operaciones aritmeticas pueden llevar mas tiempo, ademas si son valores muy variados, el algortimo tambien aumenta el tiempo de ejecucion.

**3)**

La complejidad teórica del algoritmo planteado se calculó con la siguiente lógica de análisis:

**4)**

Los ejemplos dados por la catedra, se pueden obtener los resultados pasandolos por el algortimo, todos estos, nos dieron los resultados esperados.

Tomando como ejemplo los siguientes valores

|  |  |  |
| --- | --- | --- |
| N | X | F( ) |
| 5 | 125 | 316 |
|  | 378 | 429 |
|  | 492 | 563 |
|  | 689 | 907 |
|  | 274 | 831 |

Vamos a calcular cual es el optimo, para eliminar a la mayor cantidad de enemigos posibles.

Para esto vamos a calcular todos los resultados optimos, partiendo como si tuvieramos 1 batalla hasta N = 5.

Para 1 batalla:

Calculamos el minimo entre X1 y F(1), en este caso 125.

|  |
| --- |
| 125 |

Para 2 batallas:

|  |  |
| --- | --- |
| 125 | 441 |

Para 3 batallas:

|  |  |  |
| --- | --- | --- |
| 125 | 441 | 757 |

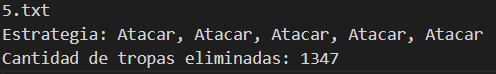
Para 4 batallas:

|  |  |  |  |
| --- | --- | --- | --- |
| 125 | 441 | 757 | 1073 |

Para 5 batallas:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 125 | 441 | 757 | 1073 | 1347 |

Para las 5 oleadeas de enemigos que vienen, como se ve en el cuadro nos combiene atacar en todos los minutos, dando como resultado el maximo que es 1347 enemigos que se eliminaron



Dando como resultado el mismo que da el algoritmo planteado.

**5)**

En el informe se encuentra un apartado dedicado a las mediciones de tiempo realizadas para corroborar la complejidad teorica del algortimo. En este apartado hay 4 conjuntos distintos de mediciones.

Las 2 primeras corresponden a la ejecucion del algortimo sin la optimizacion del valor maximo por minuto. En este caso la complejidad teeorica espeerada es de O(n^3). Donde en es la cantidad de oleadas de enemigos. Estas mediciones dan uan visiond de como se comporta el algortimo sin ninguna optimizacion.

En las otras 2, son de la ejecucion del algoritmo con la optimizacion del valor meximo por minuto implementada. Con esta la complejidad espeerada se reduce a O(n^2).

Para cada conjunto de mediciones, se generaron diversos sets de datos para probar el algoritmo bajo diferentes condiciones.

Estas mediciones son solo una aproximacion y pueden variar dependiendo de varios factores, como la capacidad de la maquina en la que se ejecutan y la carga del sistema en el momento de la prueba.

Estos datos proporcionan una buena indicacion de la eficiencia del algortimo y como escala dependiendo del tamaño de la entrada.

**6)**

# **Mediciones**

**Complejidad del algoritmo sin la optimización de máximo valor por minuto**

Orden temporal: O(n3)

|  |  |
| --- | --- |
| **Cantidad de batallas** | **Tiempo en segundos** |
| 5 | 0.00099945068359375000 |
| 10 | 0.00000000000000000000 |
| 20 | 0.00150609016418457031 |
| 50 | 0.00203466415405273438 |
| 100 | 0.02012777328491210938 |
| 200 | 0.06371688842773437500 |
| 500 | 0.74523496627807617188 |
| 1000 | 5.32637214660644531250 |
| 5000 | 621.24419236183166503906 |

A graph with a line

Description automatically generated

**Complejidad del algoritmo con la optimización de máximo valor por minuto**

Orden temporal: O(n2)

|  |  |
| --- | --- |
| **Cantidad de batallas** | **Tiempo en segundos** |
| 5 | 0.00099754333496093750 |
| 10 | 0.00000000000000000000 |
| 20 | 0.00099873542785644531 |
| 50 | 0.00100183486938476562 |
| 100 | 0.00507354736328125000 |
| 200 | 0.01508188247680664062 |
| 500 | 0.08023452758789062500 |
| 1000 | 0.24034714698791503906 |
| 5000 | 6.27057290077209472656 |

# 

# **Conclusiones**

En conclusión, este informe ha presentado un enfoque basado en programación dinámica para optimizar