

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:

- Matías Vázquez Morales (111083)
- ~~Scarlet Mendoza (108524)~~
- Nestor Fabian Palavecino Arnold (108244)

## Introducción

En este informe, presentaremos un algoritmo de programación dinámica para resolver el problema de la atacar a la máxima cantidad de enemigos, y en qué tiempos corresponden dichos ataques, teniendo como datos de entrada una lista de valores que representan la cantidad de enemigos que atacan en cada minuto y una lista que representa una función de recarga de energía.

Adicionalmente, analizaremos:

- La complejidad temporal y espacial de la solución
- Variabilidad de algunos valores en el algoritmo planteado
- Tiempos de ejecución para corroborar la complejidad teórica indicada

## La ecuación de recurrencia

La ecuación de recurrencia de nuestro algoritmo es la siguiente:

$$OPT[i] = \max(k(j))$$

$$j \in \{1, \dots, i - 1\}$$

$$k(j) = \min(X[i - 1], f(j)) + \begin{cases} 0 & \text{si } j = 0 \\ OPT[i - 1] & \text{si } j > 0 \end{cases}$$

$$OPT[0] = 0$$

Explicación:

$OPT[i]$  = máximo de enemigos eliminados en el minuto  $i$

$X[i]$  = cantidad de enemigos en el minuto  $i$

$f(j)$  = valor de la función de recarga de energía en el minuto  $j$

## El Algoritmo

### Encontrar solución

Encontrar la solución al problema, tal como lo sugiere la ecuación de recurrencia, tiene complejidad temporal cuadrática. En el código se puede observar que hay dos bloques **for** que van desde 0 hasta n, con i la cantidad de minutos de la batalla. Dentro del **for** anidado todas las operaciones se ejecutan en  $O(1)$ . Hay una llamada a la función **min**, pero es siempre entre dos elementos, por lo cual su ejecución se realiza en tiempo constante.

```
def tp2_dp(lista_xi, lista_fj):
    n = len(lista_xi)

    # Maximos batallas anteriores
    OPT = []

    for minuto_actual in range(1, n+1):
        maximo_batalla_actual = -math.inf
        minuto_maximo_batalla_actual = 0

        for minuto_origen in range(n):
            if minuto_actual <= minuto_origen:
                continue

            if minuto_origen == 0:
                maximo_batallas_anteriores = 0
            else:
                maximo_batallas_anteriores = OPT[minuto_origen - 1][0]

            j = (minuto_actual - 1) - minuto_origen
            ataque_actual = min(lista_xi[minuto_actual - 1], lista_fj[j])

            abatidos_batalla_actual = maximo_batallas_anteriores + ataque_actual

            if abatidos_batalla_actual > maximo_batalla_actual:
                maximo_batalla_actual = abatidos_batalla_actual
                minuto_maximo_batalla_actual = minuto_origen

        OPT.append((maximo_batalla_actual, minuto_maximo_batalla_actual))

    return OPT
```

## Reconstruir solución

Para reconstruir la solución al problema se recorre recursivamente el arreglo **OPT** hasta que el segundo elemento de la tupla guardada en la posición actual contenga un 0; eso significa que la recarga de energía se hizo desde el minuto 0, por lo que no hay más pasos intermedios para retroceder en el arreglo. En el peor de los casos, se recorren todos los elementos del arreglo de memoización, por lo que la complejidad es  $O(n)$ .

Luego se devuelve la solución invertida, ya que al momento de la reconstrucción se recorre el arreglo desde el final hacia el principio, pero a nosotros nos sirve la sucesión de ataque-recarga desde el minuto 0 en adelante. Esto nos cuesta  $O(n)$  también.

```
def reconstruir_solucion(memo):
    enemigos_eliminados = memo[len(memo) - 1][0]
    solucion = [ len(memo), ]

    indice = len(memo)
    while indice > 0:
        indice = memo[indice - 1][1]
        if indice > 0:
            solucion.append(indice)

    return enemigos_eliminados, list(reversed(solucion))
```

## El resto del programa

El resto del algoritmo consiste en:

- La lectura del archivo de batallas, lo cual es  $O(1 + n + n) = O(n)$ , con  $n$  la cantidad de minutos de batalla
- La escritura del archivo de solución, lo cual es  $O(n)$ , con  $n$  la cantidad de minutos de batalla

## Optimalidad del algoritmo y variabilidad de los valores de entrada

Al observar la ecuación de recurrencia de nuestro algoritmo, se puede ver que no hay restricción en principio sobre qué valores puede o no tomar cada casillero de la “matriz de memorización” (va entre comillas porque no se almacena, representa los cálculos intermedios). Si embargo, empíricamente podemos demostrar que, si existen valores negativos en la lista de oleadas de enemigos, la solución obtenida no es la óptima. A continuación, mostramos un ejemplo:

```
x = [-271, -533, -916, 656, -664]
f = [ 21, 671, 749, 833, 1543]
```

Los valores *calculados* durante el procesamiento del algoritmo (no los memoizados) son los siguientes:

```
[
[-271,    0,    0,    0,    0],
[-533, -804,    0,    0,    0],
[-916, -1187, -1449,    0,    0],
[ 656,   385,   123, -895,    0],
[-664,  -935, -1197, -1580, -8]
]
```

Luego, se muestra por pantalla:

**Enemigos eliminados: -8**

**Orden de recarga/ataque: ['Cargar', 'Cargar', 'Cargar', 'Atacar', 'Atacar']**

La cantidad de enemigos figura como **-8** porque, al momento de la reconstrucción de la solución se está tomando el elemento máximo de la última fila y a partir de ese punto aplicamos iterativamente el proceso de descubrimiento de la rama de la solución.

Esto con elementos únicamente positivos siempre funciona, pero en el caso de que hubiese valores negativos posibles para indicar la cantidad de enemigos (cosa que no tiene sentido

para el dominio del problema planteado), este programa no da la solución óptima, ya que para valores de enemigos negativos, el crecimiento de la cantidad acumulada de los enemigos abatidos no es monótono, por lo cual podría suceder lo del ejemplo, que se tome como minuto de partida el máximo de la última fila (+8), en vez de tomar el valor máximo de la anteúltima fila (656).

Luego, para cantidades de enemigos 0 o positivas, el algoritmo siempre ofrece la solución óptima, ya que la función de la cantidad de enemigos abatidos es monótona creciente, con lo cual siempre podemos arrancar desde el máximo de la última fila de la “matriz de memoización”. En otras palabras, siempre vamos a tener que “atacar” en la última batalla.

## Ejemplo de ejecución

Los ejemplos dados por la catedra, se pueden obtener los resultados pasándolos por el algoritmo, 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 óptimo, para eliminar a la mayor cantidad de enemigos posibles.

Para esto vamos a calcular todos los resultados óptimos, partiendo como si tuviéramos 1 batalla hasta  $N = 5$ .

Para 1 batalla:

Calculamos el mínimo entre  $X_1$  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 oleadas de enemigos que vienen, como se ve en el cuadro nos conviene atacar en todos los minutos, dando como resultado el máximo que es 1347 enemigos que se eliminaron

```
5.txt
Estrategia: Atacar, Atacar, Atacar, Atacar, Atacar
Cantidad de tropas eliminadas: 1347
```

Dando como resultado el mismo que da el algoritmo planteado.

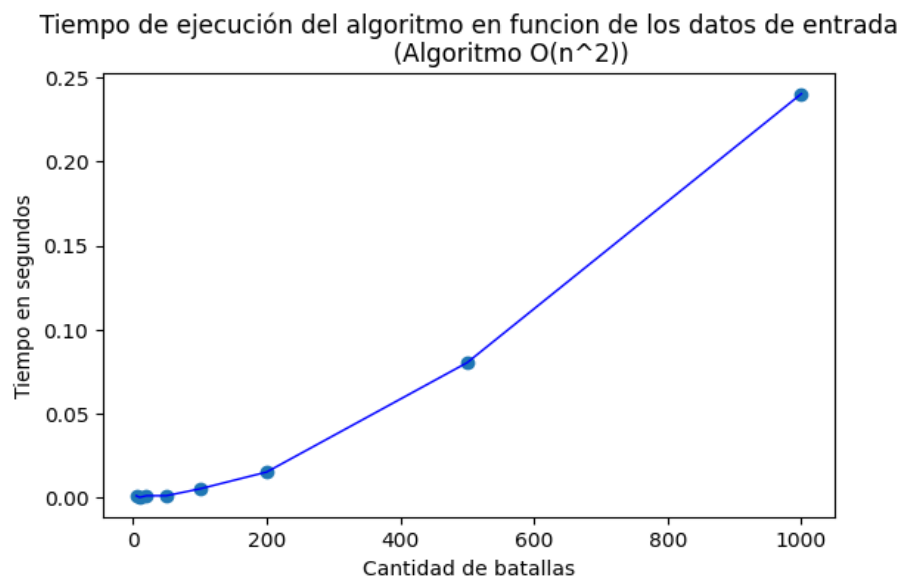


## Mediciones

Los gráficos se realizaron utilizando **matplotlib**, disponible en Python. En éstos se puede observar la complejidad temporal de nuestro algoritmo por PD.

### Pruebas de la cátedra

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

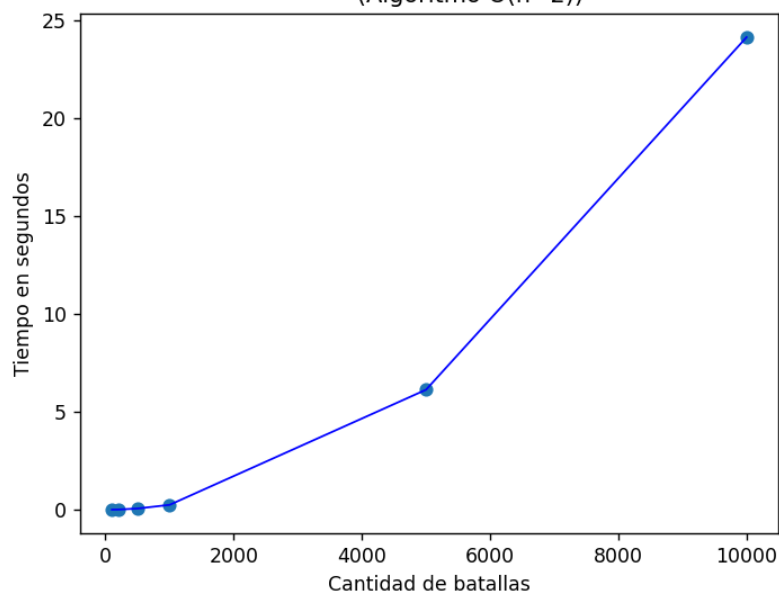


## Pruebas propias

Los datasets fueron generados utilizando el código del archivo **generator.py** del proyecto

Cantidad de batallas	Tiempo en segundos
100	0.003124237060546875
200	0.008704662322998047
500	0.060161590576171875
1000	0.247664213180542
5000	6.130439043045044
10000	24.141527891159058

Tiempo de ejecución del algoritmo en función de los datos de entrada  
(Algoritmo  $O(n^2)$ )



## Conclusiones

En conclusión, este informe ha presentado un enfoque basado en programación dinámica para resolver el problema de maximización de ataque para sucesiones de batallas.

Vimos que nuestro algoritmo resuelve el problema en tiempo cuadrático, con complejidad espacial lineal.

Explicamos en qué casos nuestro algoritmo no funciona correctamente, mostrando una prueba empírica al respecto.

Hicimos un seguimiento del algoritmo para demostrar que el mismo funciona correctamente y devuelve el resultado esperado en la complejidad indicada.

Además, revisamos los tiempos de ejecución para varios sets de prueba de batallas, tanto los de la cátedra como sets personalizados generados por nosotros mismos.