.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Programación Dinámica

6 de mayo de 2024

Tomás Hevia	Manuel Campoliete	Andrés Colina
110934	110479	110680

1. Análisis del Problema

Detallamos a partir de aquí cuál fue exactamente el proceso mental que tuvimos al enfrentarnos con el problema planteado en este trabajo práctico.

Comenzamos tomando como guía el caso particular más sencillo de los archivos de ejemplo proveídos por la cátedra, "5.txt"donde:

$$x_i = [0, 271, 533, 916, 656, 664]$$
$$f = [0, 21, 671, 749, 833, 1543]$$

Quisimos comenzar por los casos base y nos preguntamos: ¿cuál sería la mejor solución si tuviéramos solamente una ráfaga de soldados x_1 y un sólo ataque f(1)? La solución es obvia, y es atacar en el único minuto que tenemos, eliminando así 21 enemigos.

Ahora, ¿qué pasaría si tuviéramos 2 minutos? Es decir, tenemos:

$$x_i = [0, 271, 533]$$
$$f = [0, 21, 671]$$

Si bien no estábamos resolviendo por Programación Dinámica, quisimos plantear todos los casos posibles (incluso aquellos que no tienen sentido) para ver que ocurría. Los casos posibles son:

$$[\text{Cargar, Cargar}] \rightarrow \text{Tropas eliminadas} = 0$$
$$[\text{Cargar, Atacar}] \rightarrow \text{Tropas eliminadas} = 533$$
$$[\text{Atacar, Cargar}] \rightarrow \text{Tropas eliminadas} = 21$$
$$[\text{Atacar, Atacar}] \rightarrow \text{Tropas eliminadas} = 42$$

Acá nos dimos cuenta de 3 cosas:

- En primer lugar (algo que ya sabíamos y es totalmente obvio), no tiene sentido cargar todo el tiempo.
- En segundo lugar, que siempre deberíamos atacar en el último minuto, porque al menos tenemos un poder de ataque de f(1) y lo tenemos que aprovechar.
- Y en tercer lugar que si seguíamos esta lógica de resolución, estaríamos explorando un espacio exponencial de soluciones.

Para n = 2 es sencillo: planteamos los 4 casos y nos quedamos con el mejor, en este caso [Cargar, Atacar] que nos da un óptimo de 533 tropas eliminadas. Ahora, al plantear lo mismo para n = 3, la exponencialidad del espacio de soluciones ya se hace valer lo suficiente como para pensarlo realmente por Programación Dinámica, y aquí es cuando se hicieron presentes los conceptos de esta técnica de diseño.

Dimos vuelta el problema y pensamos: ¿por qué no mejor pensar que ya tenemos calculadas todas las respuestas a los subproblemas, y utilizamos éstas para responder al problema general?

Volvemos al ejemplo:

$$x_i = [0, 271, 533, 916, 656, 664]$$
$$f = [0, 21, 671, 749, 833, 1543]$$

Llamésmole OPT (de Óptimo) al arreglo contenedor de las soluciones a problemas más pequeños. Sabemos ya que OPT[0] = 0, que OPT[1] = 21 y que OPT[2] = 533. El resto lo desconocemos. Y queremos resolver OPT[5], ese es nuestro problema.

$$OPT = [0, 21, 533, ..., ..., ???]$$

Vayamos un poco más allá, y supongamos incluso que no calculamos manualmente las soluciones a los subproblemas para n = 1 y n = 2. Lo que tenemos es lo siguiente:

$$OPT = [0, ..., ..., ..., ..., ???]$$

Hagamos de cuenta de que vino Messi y nos calculó todas las posiciones del arreglo anteriores a OPT[5]. Si ese es el caso, ¿cómo podemos utilizar éstas para resolver OPT[5]? La pregunta clave es, ¿de dónde nos conviene haber venido? ¿Nos conviene haber atacado en el minuto anterior? ¿Nos conviene haber atacado 2 minutos atrás? ¿Nos conviene no haber atacado nunca todavía? Para este caso sencillo podemos escribir todas las posibilidades.

Por supuesto que vamos a atacar en este minuto, dado que es el último minuto parcial de la batalla (en este caso justo es el último minuto total). La pregunta es ¿con qué potencia vamos a atacar?. Podríamos haber atacado un minuto atrás y estaríamos atacando en el actual con una potencia de f(1). Podríamos haber atacado dos minutos atrás y estaríamos atacando en el actual con una potencia de f(2). Así hasta llegar al caso en donde todavía no atacamos, y el minuto actual sería el primero en que lo hagamos, con una potencia máxima parcial (en este caso total).

Atacar con una potencia de f(j) en el minuto actual (k), hará que eliminemos min(x_k, f(j)) enemigos. ¿Pero que cantidad de enemigos eliminamos previamente? La respuesta está en OPT. Si atacamos en el minuto anterior, atacamos en el actual con una potencia de f(1). ¿Y cuántos enemigos habíamos eliminado hasta el minuto anterior? Pues OPT[minuto_anterior], en donde suponemos tener el óptimo para ese subproblema.

Volviendo al ejemplo. Queríamos resolver OPT[5]. Y nos hacemos la pregunta, ¿desde dónde podríamos haber venido?

1. Desde OPT[4], con una potencia de f(1) para el minuto actual
2. Desde OPT[3], con una potencia de f(2) para el minuto actual
3. Desde OPT[2], con una potencia de f(3) para el minuto actual
4. Desde OPT[1], con una potencia de f(4) para el minuto actual
5. Desde OPT[0], con una potencia de f(5) para el minuto actual

Según desde dónde hayamos venido, la potencia del ataque para el minuto actual será distinta, y la cantidad de enemigos eliminados podrá variar. Veamos ahora cuantos enemigos se eliminarían en cada opción:

- Para 1. se eliminan en total OPT[4] + mín($x_5, f(1)$) enemigos.
- Para 2. se eliminan en total OPT[3] + mín($x_5, f(2)$) enemigos.
- Para 3. se eliminan en total OPT[2] + mín($x_5, f(3)$) enemigos.
- Para 4. se eliminan en total OPT[1] + mín($x_5, f(4)$) enemigos.
- Para 5. se eliminan en total OPT[0] + mín($x_5, f(5)$) enemigos.

¿Con cuál opción me quedo? Con la máxima, pues se busca maximizar la cantidad de enemigos que se pueden atacar.

Así es como llegamos a la Ecuación de Recurrencia que define el problema.

1.1. Ecuación de Recurrencia

$$\text{OPT}[n] = \max_{\forall j \in [1,n]} \{\text{OPT}[n-j] + \min(x_n, f(j))\} \tag{1}$$

1.2. Programación Dinámica

Voluntaria o involuntariamente, estuvimos aplicando los conceptos fundamentales de Programación Dinámica, para llegar a la ecuación de recurrencia que define el problema. Nos parece relevante enumerarlos:

- Memoization (técnica de guardar resultados previamente calculados)
- Lógica inductiva (¿cómo podemos usar las soluciones anteriores ya calculadas -casos más pequeños- para construir la que necesito?)
- Estrategia de construcción de solución Bottom UP
- Estrategia de reconstrucción de solución TOP DOWN
- Exploración implícita del espacio de soluciones
- La forma que tienen los subproblemas (**ataques realizados en función de la potencia del ataque, sabiendo que ésta depende de la última vez que se ha utilizado la función de recarga**)
- La forma en que dichos subproblemas se componen para solucionar problemas más grandes (**siempre atacamos en el minuto que estamos analizando y componemos con la mejor solución de todos los óptimos anteriores, sin olvidarnos que también cuenta la cantidad de enemigos que eliminemos en el minuto actual, lo que depende de la recarga con la que contemos**)

2. Proposición de un algoritmo por Programación Dinámica

Proponemos el siguiente algoritmo de resolución, que construye la solución siguiendo una estrategia Bottom Up (construimos iterativamente la solución desde subproblemas más pequeños hasta subproblemas más grandes hasta llegar a la solución del problema original), y luego obtiene la reconstrucción de la solución (en qué momentos se hacen los ataques) siguiendo una estrategia Top Down recursiva.

```
CARGAR = 'Cargar'
ATACAR = 'Atacar'

def reconstruir_solucion(n, x_i, f, OPT, solucion):
    if n == 0:
        return solucion

    solucion[n-1] = ATACAR

    maximo = 0
    pos_ultimo_ataque = n-1

    for j in range(1, n + 1):
        if OPT[n-j] + min(x_i[n], f[j]) > maximo:
            maximo = OPT[n-j] + min(x_i[n], f[j])
            pos_ultimo_ataque = n-j

    return reconstruir_solucion(pos_ultimo_ataque, x_i, f, OPT, solucion)

def pd(x_i, f):

    n = len(x_i) - 1

    OPT = [0] * (n + 1)
    OPT[1] = min(x_i[1], f[1])

    for i in range(2, n + 1):
        maximo = 0
        for j in range(1, i + 1):
            if OPT[i-j] + min(x_i[i], f[j]) > maximo:
                maximo = OPT[i-j] + min(x_i[i], f[j])
        OPT[i] = maximo

    solucion = [CARGAR] * n
    return OPT[n], reconstruir_solucion(n, x_i, f, OPT, solucion)
```

3. Complejidad del algoritmo

Para estudiar la complejidad del algoritmo planteado, separemos al algoritmo en 2: la obtención del óptimo y la reconstrucción de la solución.

En cuanto a la obtención del óptimo, la complejidad espacial es $\mathcal{O}(n)$ pues construimos un arreglo de óptimos de tamaño n. Luego, calculamos cada una de las i posiciones de este arreglo realizando operaciones $\mathcal{O}(1)$ tantas veces como i, pues se necesita comparar todos los óptimos anteriores, sumados a los ataques actuales con la correspondiente potencia para ver cuál de ellos es el máximo. Por cada posición, se hacen operaciones $\mathcal{O}(1)$ que involucran todas las posiciones anteriores. La complejidad temporal entonces es $\mathcal{O}(n^2)$.

En cuanto a la reconstrucción de la solución, la complejidad espacial también es $\mathcal{O}(n)$ pues construimos un arreglo solución de tamaño n. Para la complejidad temporal, tomamos el peor caso: haber atacado en todos los minutos. Eso implica que recorramos nuevamente todas las posiciones del arreglo de óptimos a la inversa, de manera recursiva, y que por cada una de ellas realicemos operaciones $\mathcal{O}(1)$ contra todas las posiciones anteriores, lo que recae en una complejidad de $\mathcal{O}(n^2)$ nuevamente.

En consecuencia, las complejidades finales del algoritmo son:

- Complejidad espacial: $\mathcal{O}(n)$
- Complejidad temporal: $\mathcal{O}(n^2)$

4. Variabilidad de los valores

4.1. Tiempos del algoritmo

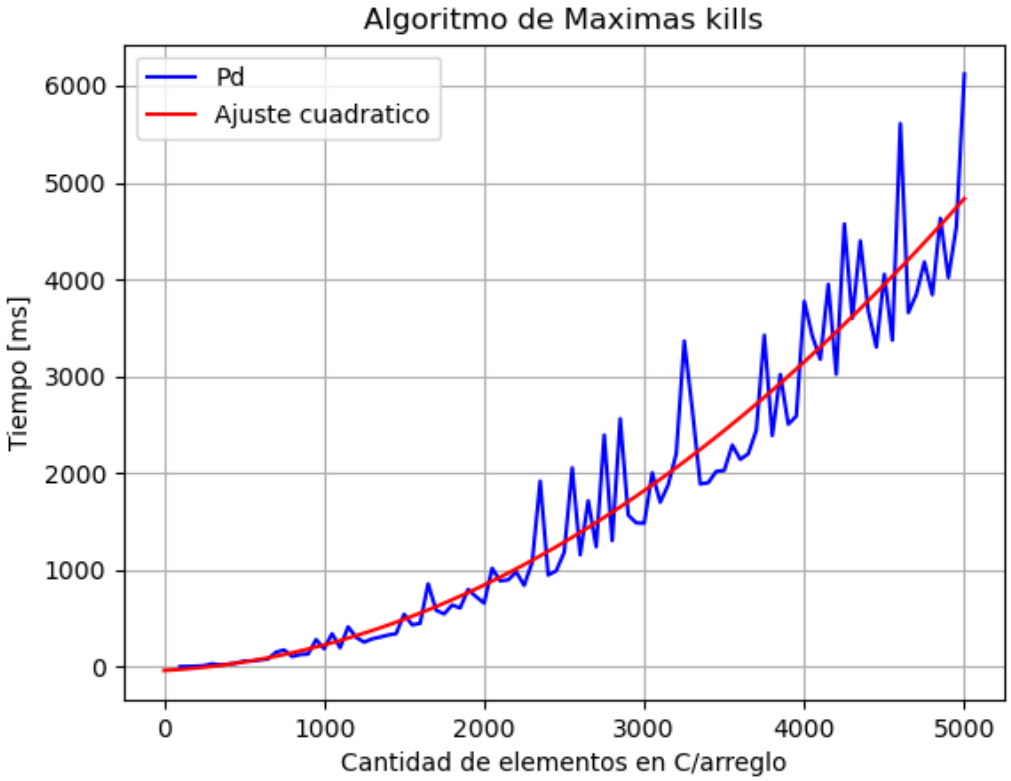
- La variabilidad de los valores de la cantidad de enemigos y los valores de la función de recarga no van a alterar el tiempo de ejecución del problema siempre y cuando esos mismos valores no tengan un tamaño numérico mayor al largo de los sets de datos (n). En caso de que el tamaño numérico de todos los valores si sea lo suficientemente grande, la complejidad del algoritmo pasaría de ser $\mathcal{O}(n^2)$ a $\mathcal{O}((X+F)*n^2)$ (siendo X el tamaño de los números en el arreglo x_i y F el tamaño de los números en el arreglo f).
- Si bien el tiempo de ejecución no va a ser distinto para el algoritmo que calcula el óptimo, el tiempo de ejecución correspondiente a la reconstrucción de la solución sí puede variar dependiendo de los óptimos previamente calculados. Por ejemplo, si resulta que la solución del problema es Cargar en todos los turnos menos en el último, la reconstrucción será mucho más rápida, ya que ésta busca cuál es el siguiente elemento del arreglo de óptimos en el cual se debe Atacar. El propio algoritmo saltaría al caso base y de esta manera la reconstrucción terminaría mucho más rápido que en un caso general donde hay muchos ataques, o en el peor caso donde siempre conviene Atacar. Esto es gracias al arreglo de solución, el cual está inicializado con todos sus elementos en Cargar, teniendo únicamente que buscar en cuáles índices es que convenía Atacar.
- Como caso extremadamente particular, si resulta que: $f(1) > x_i$, para todo i entre $[1, n]$ entonces podemos realizar una optimización al problema, puesto que siempre convendría atacar en cada iteración. Corroborar esta condición y devolver la suma correspondiente no resulta en más que una complejidad $\mathcal{O}(n)$. Esta optimización no se lleva a cabo con Programación Dinámica pero nos es relevante mencionarla puesto que mejoraría el tiempo de ejecución si es que se implementara.

4.2. Optimalidad del algoritmo

Dado que el problema planteado es resuelto mediante un algoritmo de Programación Dinámica podemos asumir que siempre se encuentra la solución óptima al problema sin importar la estructura o variabilidad de los sets de datos (siempre y cuando mantengan la estructura prevista) ya que mediante la PD se analiza todo el universo de posibles soluciones gracias al uso de *Memoization*, técnica de guardar resultados previamente calculados, que nos permite ir construyendo soluciones a subproblemas hasta llegar a la solución del problema original. En particular, en una iteración arbitraria i en nuestro algoritmo, se analiza cuál es el mayor número de eliminaciones posibles dados los óptimos ya calculados, el valor de función de recarga que corresponda, y la cantidad de enemigos que llegan en el minuto actual.

5. Mediciones

Con el objetivo de corroborar que la complejidad algorítmica es consistente con los tiempos de ejecución del programa, se crearon varios sets de datos variando de 50 en 50 desde 100 hasta 5000 elementos (para la construcción de los sets de datos se utilizó el modulo **random** que provee el lenguaje Python; cabe destacar que para los valores correspondientes a la función f(∗) nos aseguramos de que se cumpla la condición de que sea monótona creciente, ver en el (**Anexo**, sección **7.1**). Una vez obtenidos los sets de datos, ejecutamos el algoritmo para cada uno y tomamos el tiempo de ejecución 5 veces para posteriormente obtener el promedio correspondiente. Finalmente realizamos un ajuste de función cuadrática $f(x) = ax^2 + bx + c$ (la cual corresponde a la complejidad propuesta) sobre nuestras mediciones obtenidas (utilizamos la función **curve_fit** del módulo **scipy.optimize**), viendo así que la curva cuadrática ajusta bien los datos obtenidos. Ambas gráficas se muestran a continuación:



6. Conclusión

Mediante el análisis del problema dado y el seguimiento del caso particular "5.txt"logramos plantear la ecuación de recurrencia para la resolución de este problema. El descubrimiento de la misma nos permitió luego desarrollar fácilmente el algoritmo. Luego corrimos el algoritmo sobre nuestros propios tests y pudimos corroborar la optimalidad del mismo, confirmando que la ecuación de recurrencia era la correcta. También analizamos el cómo podría afectar la variabilidad de los valores en los tiempos de ejecución del algoritmo y en su optimalidad. Para finalizar este informe, podemos concluir en que este problema se puede resolver de forma óptima con Programación Dinámica y que Ba Sing Se va a estar totalmente protegida de los ataques de la Nación del Fuego.

7. Anexos

7.1. Creación de elementos de f(∗)

```
    from random import randint

    MIN_F = 0
    MAX_F = 10000

    margen_de_aumento = MAX_F // n
    ultimo = MIN_F
    for _ in range(n):
        f = ultimo + randint(0, margen_de_aumento)
        archivo.write(str(f) + "\n")
        ultimo = f + 1
```