FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Programación Dinámica

9 de octubre de 2023

Martín González Prieto	Santiago Langer	Camila Teszkiewicz
105738	107912	109660

Índice

1. Introducción: análisis del problema

En este problema tenemos dos secuencias de variables: los esfuerzos por día y la energía por días desde el último descanso.
El esfuerzo de un día es la máxima ganancia que podemos sacar de ese día. Sacar la máxima ganancia requiere que hayamos llegado a ese día con suficiente energía, si llegamos con menos energía esa ganancia no será el máximo posible, sino que será la energía con la que llegamos a ese día. No podemos controlar cual es la máxima ganancia de los N días que queremos optimizar. Llamaremos e_i al esfuerzo de cada día.

e_1	e_2	e_3	e_4	e_5
1	5	4	10	7

La segunda secuencia de variables del problema es la energía de cada día. Los jugadores van perdiendo energía cuantos más días pasan desde el último descanso. Llamaremos s_j a la energía de cada día. Por ejemplo si no descansan nunca, la energía podría ser esta:

s_1	s_2	s_3	s_4	s_5
10	9	7	3	2

En el caso anterior, si descansamos al tercer día la energía sería:

d_1	d_2	d_3	d_4	d_5
s_1	s_2	D	s_1	s_2
10	9	D	10	9

y la ganancia de los 5 días sería 23:

G_1	G_2	G_3	G_4	G_5
1	5	0	10	7

La energía es independiente de la intensidad de los entrenamientos, solo se recupera cuando decidimos descansar. Al ser descendiente, luego de descansar siempre tendremos la máxima energía posible.

1.1. Recorrido implícito del espacio de posibilidades: algunas consideraciones lógicas

A cada e_i le puede llegar a corresponder cualquier s_j, siempre que $j <= i$, por esto no hay una relación lineal que nos permita calcular una ganancia óptima por día ya que las variables no son estáticas, la energía depende del descanso y la ganancia del entrenamiento depende de la energía con la que se realiza.
Por otro lado, no tendría sentido descansar dos días seguidos[1]. Justifiquemos esta afirmación: Siempre que los jugadores descansan un día vuelven a tener la máxima energia posible. Definamos:

- e_i = esfuerzo correspondiente al día i
- s_j = energia disponible
- G_i = ganancia en caso de entrenar el día i ; $G_i = \min(s_j, e_i)$

Si $e_i > s_j$ para todo s_j posible, la ganancia máxima G_i será el primer s_j, ya que es el más grande. Es decir, la ganacia de realizar el entrenamiento e_i con energia s_j será menor o igual a la ganancia de realizar el entrenamiento e_i con energia s_1, para todo e_i.
Ahora tomemos como dato que por la cantidad de esfuerzo que requiere un día i, me conviene realizarlo con energia s_1. Para que se cumpla basta con haber descansado tan solo el día anterior, y ya recupero la energía. Sin importar cuántas veces haya entrenado antes del descanso, en vez de descansar dos dias seguidos va a convenir realizar el entrenamiento e_{i-2} con la energia disponible. Aunque la ganancia sea pequeña la beneficio siempre será mayor que cero.

[1]Esto se comprueba experimentalmente más adelante

- $i > 2$, para tener más de dos días
- $G_i = (e_i, s_1) \longrightarrow G_{i-1} = 0$
- $G_{i-2} = max(Descanso = 0, (e_{i-2},$ cualquier s_j $))$, con$j <= i - 2$
- $(e_{i-2},$ cualquier s_j $) > 0$ para todo i,j

Si el día $i - 1$ se descansa, con el fin de maximizar G_i, jamás tendría sentido descansar el día $i - 2$. Toda esta lógica nos va induciendo una idea: cada día tenemos dos posibilidades, entrenar o no entrenar. *¿Podríamos modelar el problema como Juan el Vago?*

2. Primer modelo: Juan el vago

Para esta primera propuesta haremos un planteo similar al ejercicio de Juan el Vago: recorremos día a día los entrenamientos definidos y decidimos cada día si debemos entrenar o no, almacenando el óptimo para cada día. Los subproblemas son cuál es el óptimo para i entrenamientos. Para este primer modelo la ecuación de recurrencia es relativamente sencilla:

$$e_i = max(G_i \text{ habiendo entrenado ayer} + e_{i-1}, G_i \text{ habiendo descansado ayer} + e_{i-2}) \quad (1)$$
$$= max((e_i, s_i) + e_{i-1}, (e_i, s_1) + e_{i-2}) \quad (2)$$

Tomamos los casos base $i = 1$, $i = 2$
llamamos (e_i, s_j) a la ganancia generada por e_i, s_j

$$e_1 = (e_1, s_1)$$
$$e_2 = max(e_1 + (e_2, s_2), (e_2, s_1))$$

El algoritmo:

```
def get_alternative_training(effort_list, energy_list):
    cant_e = len(effort_list)

    first_train = min(effort_list[0], energy_list[0])
    second_train = max(first_train + min(effort_list[1], energy_list[1]), min(
    effort_list[1], energy_list[0]))
    opt = [first_train, second_train]

    for i in range(2, cant_e):
        a = min(effort_list[i], energy_list[i]) + opt[i-1]
        b = min(effort_list[i], energy_list[0]) + opt[i-2]
        opt.append(max(a, b))

    return opt[cant_e - 1]
```

Primero pensemos si podemos encontrar un contraejemplo sencillo y descriptivo de la situación.
Caso:

e_1	e_2	e_3	e_4	e_5	e_6	e_7
40	1	40	6	1	40	1

s_1	s_2	s_3	s_4	s_5	s_6	s_7
40	6	5	4	3	2	1

Solución óptima:

$$G = ((e_1, s_1) + 0 + (e_3, s_1) + (e_4, s_2) + 0 + (e_6, s_1) + (e_7, s_2)) = 127$$

s_1	s_2	s_3	s_4	s_5	s_6	s_7
E	D	E	E	D	E	E

Pensemos paso a paso el algoritmo:
Casos base:

$$e_1 = (e_1, s_1) = 40$$

$$e_2 = max(e_1 + (e_2, s_2), (e_2, s_1)) = max(1 + 40, 1) = 41$$

Iteraciones:

- $e_3 = max((e_3, s_3) + e_2, (e_3, s_1) + e_1) = max(5 + 41, 40 + 40) = 80$
- $e_4 = max((e_4, s_4) + e_3, (e_4, s_1) + e_2) = max(4 + 80, 6 + 41) = 84$
- $e_5 = max((e_5, s_5) + e_4, (e_5, s_1) + e_3) = max(1 + 84, 1 + 80) = 85$
- $e_6 = max((e_6, s_6) + e_5, (e_6, s_1) + e_4) = max(2 + 85, 40 + 84) = 124$
- $e_7 = max((e_7, s_7) + e_6, (e_7, s_1) + e_5) = max(1 + 124, 1 + 85) = 125$

Solución del estilo Juan el Vago:

s_1	s_2	s_3	s_4	s_5	s_6	s_7
E	D	E	E	D	E	E

La misma, pero... G = 125. Esto es menor que la solución óptima de G = 127.

Elegimos este ejemplo porque se ve claramente el error algorítmico.[2] En la solución óptima, se entrena e_4 con energía s_2. Ahora, si vamos al calculo de e_4 en nuestro algoritmo, esta opción no se tiene en cuenta. Solo se comparan las posibilidades suponiendo (e_4,s_1) y (e_4,s_4), y luego se elige (e_4,s_4), lo cual es incorrecto viendo la solución que venimos construyendo.
¿Cómo podríamos corregir este error?

Intentemos *emparcharlo*: el problema resultó ser que el algoritmo no tiene en cuenta la progresión de los s_j. Necesitamos actualizar de alguna manera esa información. Si lo hacemos de manera aislada en relación a los e_i pareciera que nuestra solución va a ser muy dificil de reconstruir, pero insistimos con esta idea, porque tiene un problema mucho más importante. La ganancia de cada día depende de e_i y s_j, además cada ei podría combinarse con cualquier sj (recordemos la restricción $j <= i$). Toda esta información se almacenaría en una matriz, lo que nos permite concluir que nuestro problema tiene dos dimensiones.

3. Solución óptima: la matriz

Como se ha visto en el caso anterior, para obtener una solución válida se requiere explorar todo el espacio de posibilidades de soluciones donde se pueda contemplar $min(e_i, s_j) \quad \forall \quad i, j$.
Es por esto que elegimos confeccionar una matriz para resolver dicho problema. Antes de construirla tengamos en cuenta algunas consideraciones.

[2]También se puede ver cómo falla en la sección *Puesta a Prueba*

3.1. Consideraciones previas

1. Primeramente, cabe destacar que no será una matriz completa, ya que es imposible realizar un entrenamiento i con una energía j dónde $i < j$. Es decir, es imposible que se pueda realizar un entrenamiento con una energía que se encuentre más adelante cronologicamente, ya que para eso se debería haber entrenado j - i veces antes.

2. Como anteriormente se ha dicho en la introducción, no hay ninguna razón por la cual sería óptimo no entrenar durante dos días seguidos, por lo cual la matriz va a considerar dos subproblemas esenciales: si se entrena el día anterior o no se entrena. La forma de poder ver esto, es si se encuentra en la columna de S_1 o no, es decir en caso de que se encuentre en ese caso, no se entrenó el día anterior, y en caso contrario sí.

3. El último aspecto a tener en cuenta, que servirá para conseguir la solución global del problema, es que siempre en conveniente entrenar el último día. Esto se debe a que no importa si $min(e_i, s_j) = s_j$ para cualquier s_j ya que al no tener que entrenar el día siguiente, nunca sería conveniente ahorrarse dicho entrenamiento para renovar la energía.

3.2. Ecuación de recurrencia

Según la consideracion previa (2) es que podemos entender mejor los subproblemas de dónde nace la ecuación de recurrencia:

- Si estreno el dia anterior $(s_j \neq S_1)$:
  $M(e_i, s_j) = min(e_i, s_j) + M(e_{i-1}, s_{j-1})$

- Si no entreno el dia anterior $(s_j = S_1)$
  $M(e_i, s_j) = min(e_i, S_1) + max(M(e_{i-2}, s_{j'}) \; \forall \; 1 < j' < i, i > 2)$[3]

Cabe destacar que para conseguir la solución de la ganacia máxima, dado el considerando (3), alcanza con conseguir el $max(M(e_n, s_j) \; \forall \; 1 < j < n)$ siendo n la cantidad de entrenamientos, es decir, el valor máximo del último entrenamiento dada cualquier energia. Esto es un paso importante para poder reconstruir el algoritmo.

Teniendo en cuenta el ejemplo anteriormente utilizado en el caso de Juan el vago, la matriz quedaria de esta manera:

	s_1	s_2	s_3	s_4	s_5	s_6	s_7
e_1	40	0	0	0	0	0	0
e_2	1	41	0	0	0	0	0
e_3	80	7	46	0	0	0	0
e_4	47	86	12	50	0	0	0
e_5	81	48	87	13	51	0	0
e_6	126	87	53	91	16	53	0
e_7	88	127	88	54	92	17	54

Cuadro 1: Matriz para el ejemplo de Juan el vago

Como se puede ver en la tabla el máximo de la última fila, correspondiente al último entrenamiento, es 127, igual a la ganancia esperada.
Cabe destacar que hay casos base dónde la ecuación de recurrencia tiene excepciones. No solo para todo j > i no se puede calcular, sino que la búsqueda del máximo para e_{i-2} no se puede calcular para el caso de e_1 y e_2, por lo que además, si se llega a (e_2, s_1) solamente debe considerarse que no se entrenó el primer entrenamiento.

[3] Para evaluar en dichas cotas se tiene en cuenta la consideracion previa (2)

3.3. Algoritmo

El algoritmo utilizado para conseguir dicha matriz fue el siguiente.

```python
def get_matrix_of_training(effort_list, energy_list):
    matrix = [[0 for j in range(len(effort_list))] for i in range(len(energy_list))]
    # fil: ei -> entrenamientos(effort), col: si -> energia(energy)

    for ei in range(len(effort_list)):
        for sj in range(len(energy_list)):
            if(sj > ei): continue
            if(sj == 0):
                if(ei == 0 or ei == 1):
                    matrix[ei][sj] = min(effort_list[ei], energy_list[sj])
                else:
                    matrix[ei][sj] = min(effort_list[ei], energy_list[sj]) + max(matrix[ei-2])
            else:
                matrix[ei][sj] = min(effort_list[ei], energy_list[sj]) + (matrix[ei-1][sj-1])

    return matrix
```

3.4. Reconstrucción de la secuencia de entrenamientos

La reconstrucción de la secuencia de entrenamientos se basa en invertir el algortimo anterior, teniendo en cuenta en el único caso en el que se descansa y no se realiza un entrenamiento, es cuando se llega a la columna de s_1.

	s_1	s_2	s_3	s_4	s_5	s_6	s_7
e_1	40	0	0	0	0	0	0
e_2	1	41	0	0	0	0	0
e_3	80	7	46	0	0	0	0
e_4	47	86	12	50	0	0	0
e_5	81	48	87	13	51	0	0
e_6	126	87	53	91	16	53	0
e_7	88	127	88	54	92	17	54

Cuadro 2: Reconstrucción de la secuencia de entrenamientos

Como se ve en la tabla, no se realizaron los entrenamientos e_2 y e_5, quedando así la secuencia esperada y anteriormente mostrada en el ejemplo de Juan el vago, [E,D,E,E,D,E,E]. El funcionamiento se resume en arrancar desde el máximo del último entrenamiento e_7, y avanzar en diagonal hasta llegar a la primera columna, allí se considera que no se realizó el e_{i-1}, y se avanza al máximo de e_{i-2}. Obviamente al final teniendo en cuenta los casos base explicados en la ecuación de recurrencia.

El algoritmo que se consguió para realizar esta reconstrucción fue el siguiente.

```
def get_best_secuence_of_trainings(effort_list, energy_list):
    matrix = get_matrix_of_training(effort_list, energy_list)
    ei = len(effort_list)-1
    sj = matrix[ei].index(max(matrix[ei]))
    secuence = []

    while(not (ei == 0 and sj == 0)):
        secuence.insert(0, 'E')
        if(sj == 0):
            secuence.insert(0, 'D')
            if(ei == 1):
                ei -= 1
                continue
            else:
                ei -= 2
            sj = matrix[ei].index(max(matrix[ei]))
        else:
            ei -=1
            sj -= 1
        if(ei == 0): secuence.insert(0, 'E')

    return secuence
```

4. Complejidad del algoritmo

Es sencillo comprender la complejidad temporal del algoritmo si observamos la matriz con la que resolvemos el problema.

	s_1	s_2	s_3	s_4	s_5	s_6	s_7
e_1	40	0	0	0	0	0	0
e_2	1	41	0	0	0	0	0
e_3	80	7	46	0	0	0	0
e_4	47	86	12	50	0	0	0
e_5	81	48	87	13	51	0	0
e_6	126	87	53	91	16	53	0
e_7	88	127	88	54	92	17	54

Cuadro 3: Matriz para el ejemplo de Juan el vago

Primero, debemos armar la matriz. Esta operación es $O(N^2)$ ya que esta matriz tiene N filas y N columnas. Luego necesitamos calcular los valores de estas N^2 celdas. Hay dos tipos de celdas que tienen costos de calculo distintas. Comencemos por las más sencillas:
Para la gran mayoría del gráfico hay que realizar operaciones $O(1)$ como continuar, comparar, buscar un valor o guardar un valor[4]. Esto tiene un costo de $O(N^2)$, ya que son casi N^2 celdas con operaciones $O(1)$.
Además hay N celdas de la columna 1 que son más costosas: deben realizar una operación $O(N)$, recorrer la fila anterior. N celdas realizando operaciones $O(N)$ significa que esta parte del algoritmo también es $O(N^2)$.
Teniendo 3 operaciones $O(N^2)$, la construcción de toda la matriz termina siendo una operación $O(N^2)$ según la notación Big Oh.

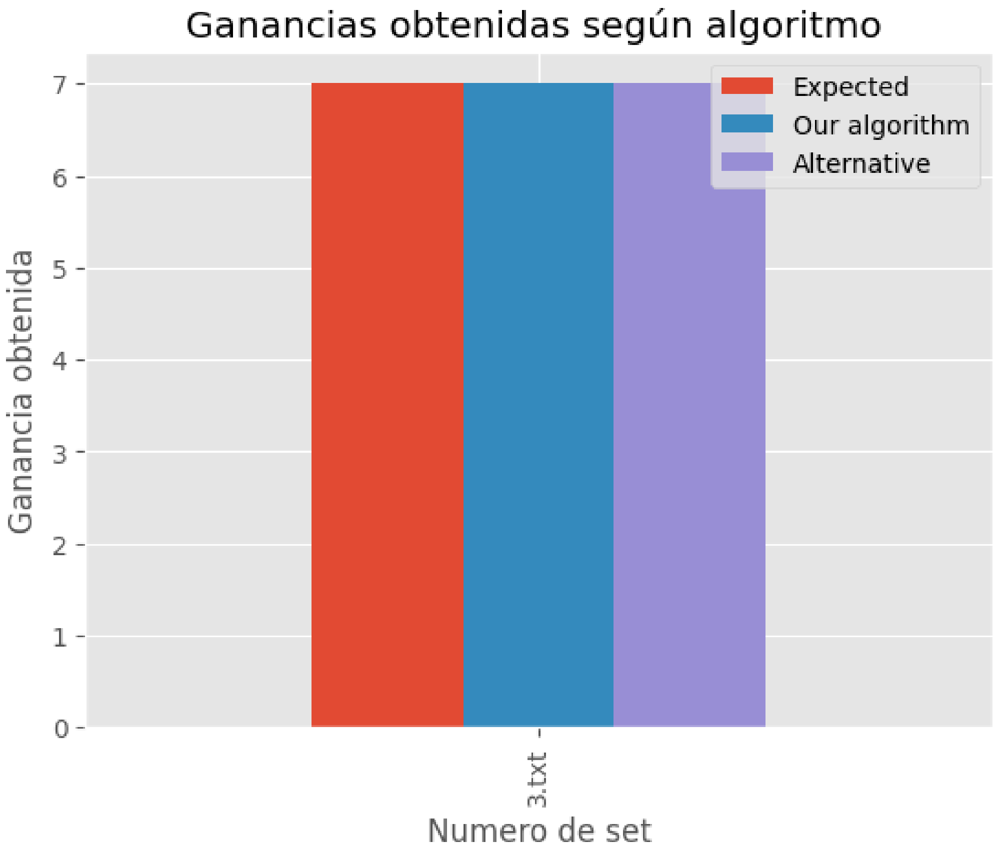
Por el lado de la reconstrucción de la secuencia, nos encontramos con una complejidad $O(N)$, ya que se realiza un recorrido lineal de las columnas de la matriz utilizando las mismas comparaciones $O(1)$ que en el caso de la construcción de la matriz.

[4]Algunas celdas son más complejas que otras (por ejemplo la mitad de la matriz simplemente se llena con ceros) pero para la notación Big Oh esto no es importante.

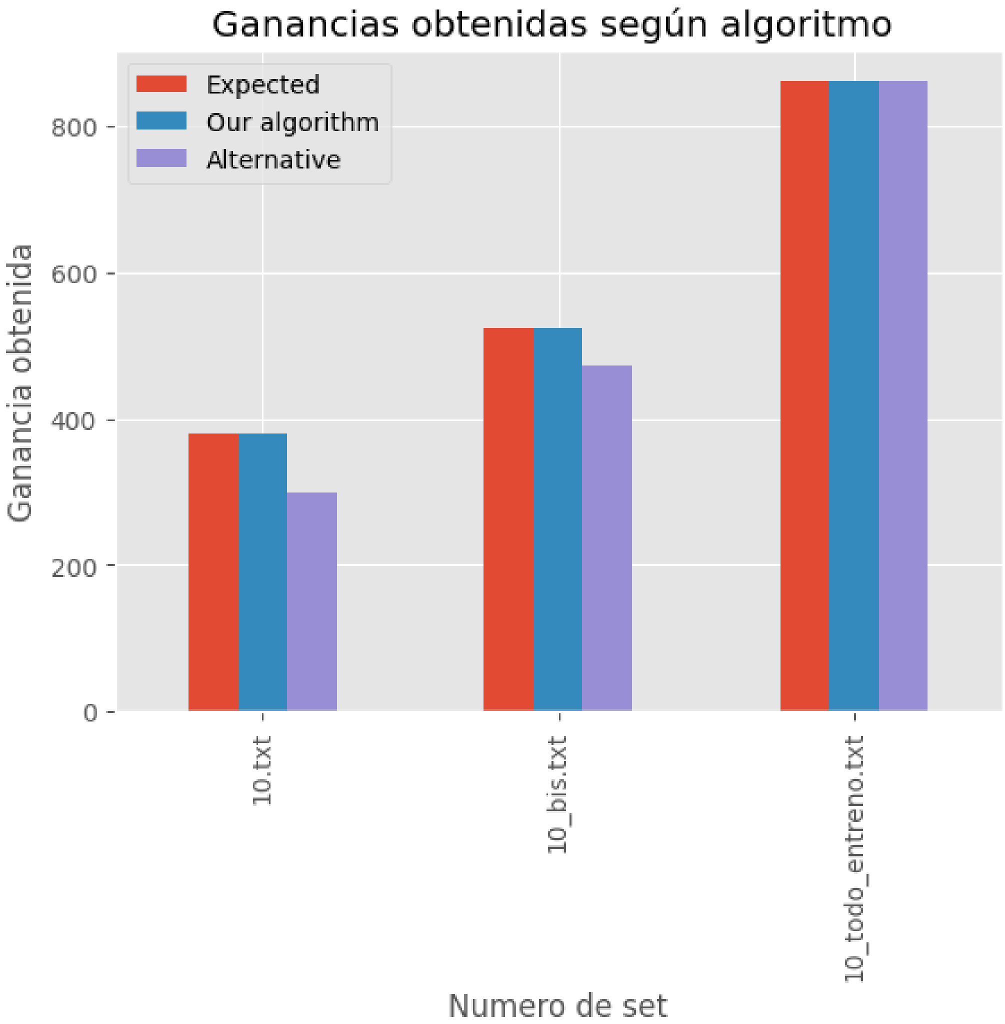
5. Puesta a prueba

Habiendo justificado el algoritmo diseñado, lo pondremos a prueba. Primero, veremos qué resultados da con los sets de prueba de la cátedra. Luego, utilizaremos nuestros propios sets de prueba.

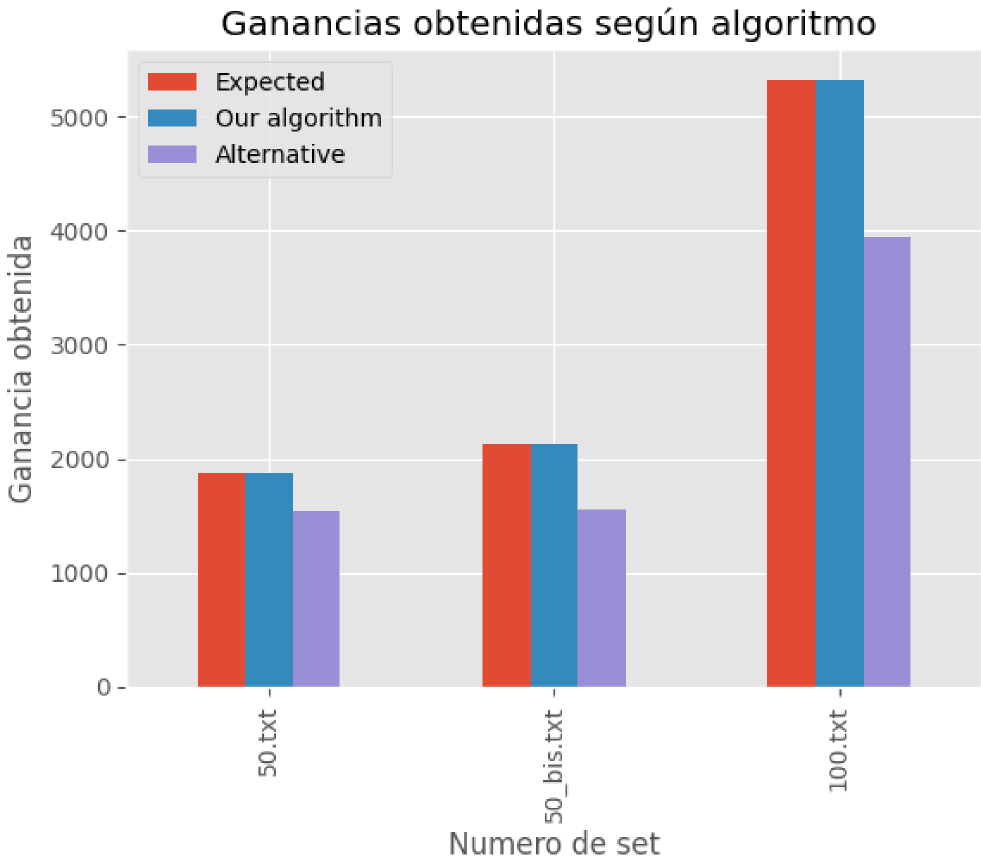
5.1. Sets provistos por la cátedra

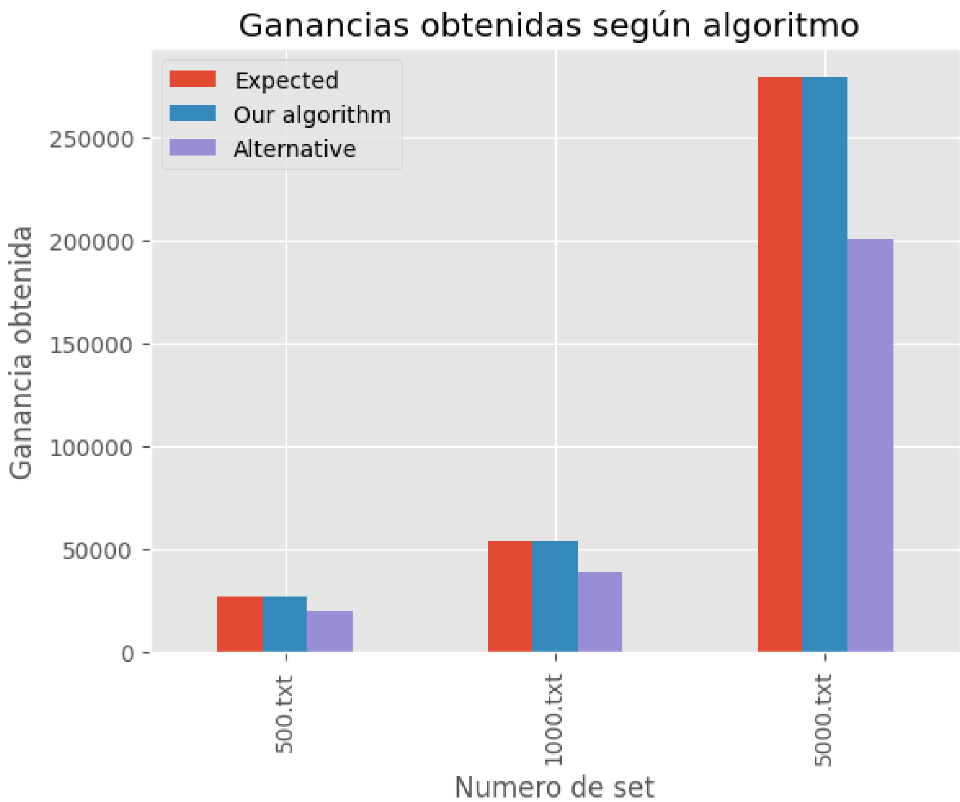


En el primer set de 3 elementos podemos ver que nuestro algoritmo propuesto alcanza la ganancia esperada de 7. Sin embargo, la propuesta alternativa también consigue el resultado esperado. Veremos en los próximos sets si sigue así.



En estos 3 sets nuestra propuesta sigue dando las respuestas correctas. Sin embargo el método alternativo muestra algunos casos donde no llega al resultado óptimo.

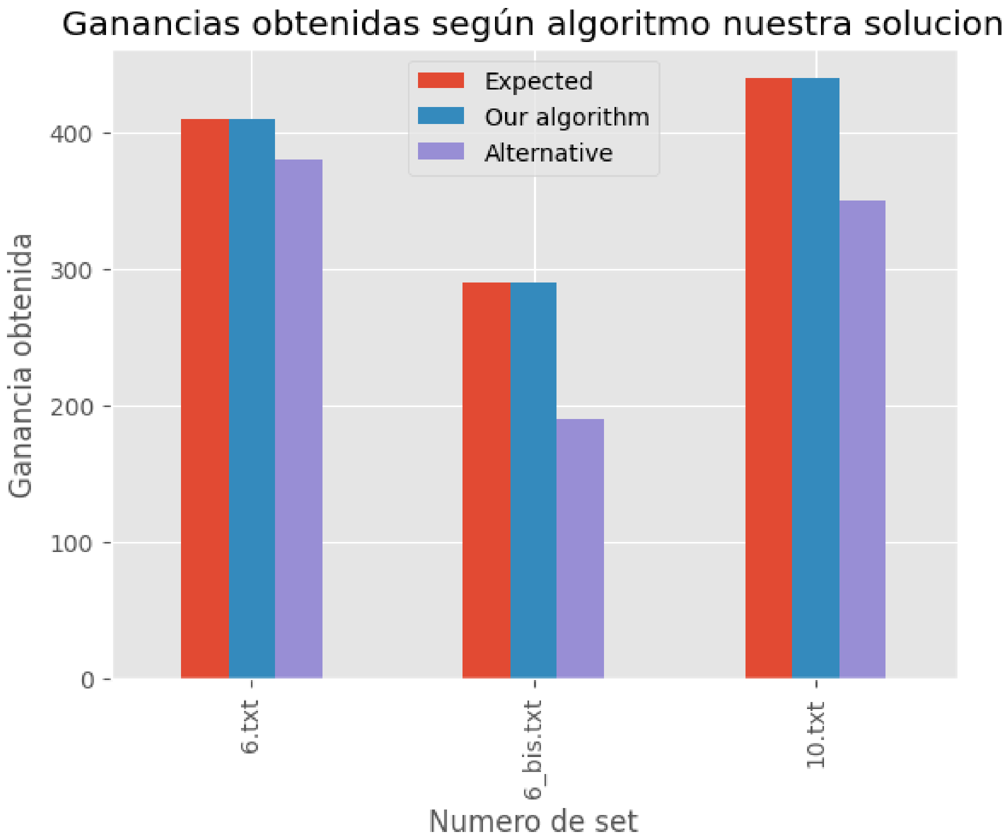




En estos últimos dos gráficos podemos ver que nuestro algoritmo siempre alcanza los resultados correctos con los sets provistos por la cátedra.

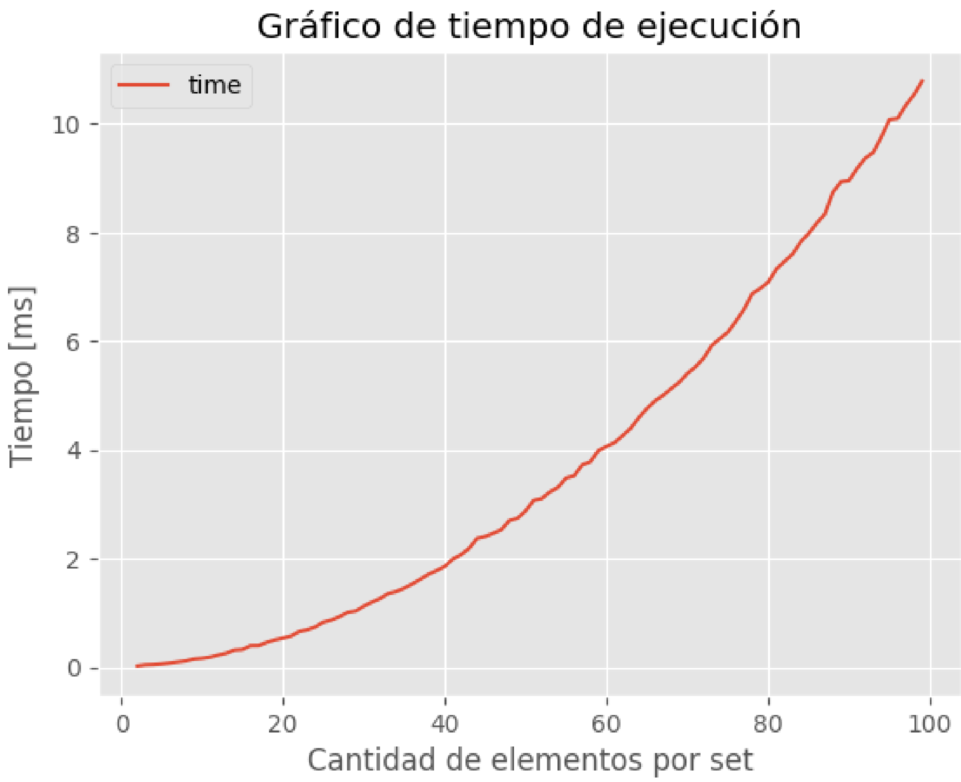
5.2. Sets propios

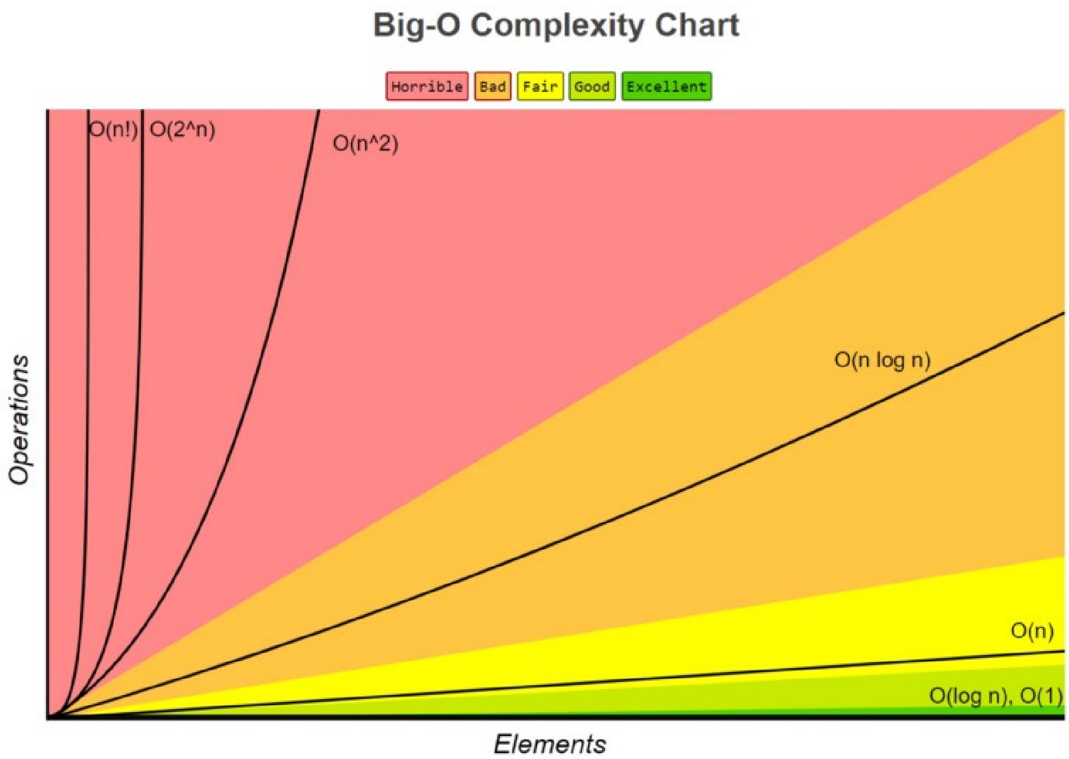
Para seguir poniendo a prueba a nuestro algoritmo, creamos nuestros propios sets de datos. Tenemos 3 nuevos sets de datos, dos de 6 elementos y uno de 10 (pueden verse en detalle en el repositorio, junto a su Jupyter notebook). Buscamos la solución correcta de cada uno con lapiz y papel.



Como podemos ver, también en estos nuevos sets el algoritmo llega a la solución esperada.

6. Tiempos de ejecución





Como podemos ver en los gráficos de tiempo de ejecución, nuestro algoritmo coincide con el analisis formulado en la sección *Complejidad del algoritmo*. La linea curva del gráfico coincide con la linea de $O(N^2)$ del cuadro de referencia.

7. Conclusión

Durante el desarrollo de este trabajo práctico, evaluamos varias posibles soluciones hasta que logramos encontrar con una que nos retorne soluciones óptimas.

Estudiando el caso de Juan el vago, por ejemplo, notamos que tener en cuenta un solo eje en algunos casos funcionaba, pero otros no. Esto nos permitió notar que el problema se solucionaba evaluado dos ejes distintos (a diferencia del trabajo anterior). Esto nos permite concluir que a veces podemos seleccionar qué información nos es util, mientras que en otros casos todas las variables del problema son importantes.

A partir de nuestra solución, se puede trazar un claro paralelismo con el problema de la mochila (Knaspack). Si en vez de tener un vector de diferentes energías, tuviesemos un valor escalar que se va reduciendo por cada entrenamiento entrenado, nos encontraríamos exactamente con el mismo problema. Como es sabido, el problema de la mochila tiene una complejidad pseudo polinomial de $O(N * W)$, como en nuestro caso W no es un valor numérico sino la longitud del vector S, la complejidad encontrada es $O(N^2)$. Por lo tanto, podemos concluir que tanto el tiempo como la complejidad no se ven afectados por los valores de los arreglos *effort* y *energy*, ya que para la confección de la matriz solo repercute la longitud de los mismos.