.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Programación Dinámica

7 de febrero de 2024

Emanuel Pelozo 99444

Agustina Fraccaro 103199

Agustina Schmidt 103409

Matias Gonzalez 104913

1. Análisis del problema

El problema consiste en que Scaloni debe determinar la mejor manera de organizar los días de entrenamiento para maximizar la ganancia. Se deben tener en cuenta las limitaciones de energía de los jugadores, que con el paso de los días va disminuyendo, teniendo que contemplar si conviene o no descansar un día.

Tenemos 3 variables en nuestro problema:

- n: Cantidad de días disponibles para entrenar.
- e_i: Esfuerzo que demanda el entrenamiento del día i.
- s_j: Cantidad de energía disponible para cada día, j puede ser distinto de i ya que la energía se renueva si se descansa.

La ganancia de cada día será el esfuerzo requerido siempre y cuando la energía sea suficiente. Si para un mismo día $s_j < e_i$ entonces la ganancia será s_j. Teniendo esto en cuenta, podemos decir que la ganancia del día i será: $g_i = min(e_i, s_j)$.

Por último, cabe mencionar que si un día se decide descansar, no se obtendrá ganancia y las energías se renovarán, siendo la energía para el siguiente día al descanso de s_1 nuevamente.

Con base en las condiciones mencionadas, podemos deducir entonces que no es conveniente descansar más de un día seguido, esto es porque si bien descansar significa no sumar ganancia, sí nos brinda el hecho de renovar la energía. Con un solo día de descanso ya renovamos la energía y tendrá el valor de s_1, si descansáramos dos días o más, la energía seguiría siendo s_1 (que es la máxima energía posible) y tampoco sumaríamos ganancia, por esto mismo no es conveniente.

1.1. Utilizando programación dinámica

Para resolver el problema correctamente deberíamos poder considerar todas las combinaciones posibles de las variables mencionadas anteriormente, para esto utilizamos el enfoque de Programación Dinámica, ya que su optimización a través de Memoization nos permite poder considerar y reutilizar soluciones a problemas de menor tamaño para la solución final.

Para comenzar con la resolución y teniendo presente la premisa de que no nos conviene descansar más de un día, el enfoque que elegimos fue ver hacia atrás y realizar la mejor elección para el día actual considerando si venimos de haber entrenado o descansado el día anterior.

Comenzando con nuestro análisis de los primeros subproblemas identificamos los siguientes casos:

- Si entrenamos un día ($n = 1$), la respuesta entre entrenar o descansar siempre será entrenar, sin importar de cuánto es la ganancia, dado que si se descansa no se obtiene ninguna ganancia. Este es nuestro caso base.
- Para 2 días ($n = 2$), podemos llegar al mismo habiendo entrenado o habiendo descansado. Las opciones son sumar las ganancias de entrenar ambos días o tener solo la ganancia del segundo día dado que el primero descanso.
- Para 3 días ($n = 3$), hay 3 opciones de cómo llegamos al tercer día:
  - Habiendo entrenado los dos días anteriores.
  - Habiendo descansado el día 1 y entrenado el día 2.
  - Habiendo descansado el día 2 y entrenado el día 1.
- Para 4 días ($n = 4$), tenemos las siguientes opciones:
  - Habiendo entrenado los tres días anteriores.

- Habiendo entrenado los dos días anteriores y descansado el día 1.
- Habiendo entrenado el día anterior, descansado el día 2 y entrenado el día 1.
- Habiendo descansado el día 3 y entrenado el día 1 y 2.

Vemos que a medida que aumenta n, empiezan a variar la cantidad de combinaciones de entrenamientos de los días previos al actual (n) teniendo en cuenta los descansos de por medio.

Generalizando, podemos decir que estando en el día n, podemos llegar habiendo entrenado de manera consecutiva $n-1$ días, $n-2$ días y así sucesivamente hasta llegar a 0 días (en este caso descansé el día anterior: $n-n$).

1.2. Ecuación de Recurrencia y Memoization

Para poder recordar nuestras soluciones parciales, vamos a modelar el problema en una matriz de tamaño n x n. Para nuestra explicación, las filas serán la cantidad de días entrenados sin interrupciones y las columnas serán los distintos valores de n.

Cabe aclarar otro caso borde, para $n = 0$ la ganancia siempre será 0 dado que eso significa que no hay ningún día disponible para entrenar.

Si el día $n-1$ descansé, necesito el óptimo del día $n-2$ para sumarle a eso mi ganancia del día n ya que en este escenario, el día $n-1$ no hubo ganancia.

En el caso de haber entrenado el día $n-1$, entra en juego una nueva variable: j, que es la cantidad de días entrenando consecutivamente con la que llego al día actual. Mi ganancia de entrenar n días una cantidad j de días consecutivos será la ganancia del día n más la ganancia del día $n-1$ habiendo entrenado $j-1$ días seguidos anteriormente.

Vamos a poner un ejemplo para poder visualizar lo explicado, estableciendo $n = 5$ con los siguientes valores de esfuerzo y energía:

e_i : $67, 8, 12, 34, 6$

s_i : $76, 47, 39, 22, 10$

Comenzamos resolviendo para $n = 1$, en este caso, siempre será conveniente entrenar por lo que el valor en esa celda será $min(e_1, s_1) = 67$.

Continuamos para $n = 2$:

- Para el valor $j = 0$ que representa no haber entrenado el día anterior, el valor será únicamente la ganancia de entrenar ese día: $min(e_2, s_1) = 8$.
- Para el valor $j = 1$ que representa haber entrenado el día anterior, el valor será la ganancia del día 1 más la ganancia del día 2: $67 + min(s_2, e_2) = 67 + 8 = 75$.

La matriz queda de la siguiente manera:

1	2	3	4	5	n/j
					4
					3
					2
	75				1
67	8				0

Figura 1: Matriz resolución de ejemplo, cargada para los días n = 1 y n = 2.

Para $n = 3$, tenemos 3 opciones:

- Para el valor $j = 0$, que representa no haber entrenado el día anterior, el valor será el óptimo del día 1 más la ganancia del día 3, estableciendo como óptimo al mayor valor dentro de las soluciones para ese día: $OPT(n-2) + min(s_1, e_3) = 67 + 12 = 79$.
- Para el valor $j = 1$, que representa haber llegado entrenando **un solo** día consecutivo, el valor será la ganancia propia más la ganancia del día anterior para su caso de $j = 0$, ¿Por qué para $j = 0$? Porque para haber llegado entrenando (al día actual) un día consecutivo, significa que nuestro día anterior lo empezamos desde un descanso. Nos queda: $8 + min(s_2, e_3) = 8 + 12 = 20$.
- Para el valor $j = 2$, que representa haber entrenado dos días seguidos (este sería el caso de entrenar los 3 días), el valor será la ganancia del día anterior para el caso $j = 1$ más la ganancia propia: $75 + min(s_3, e_3) = 75 + 12 = 87$.

La matriz queda de la siguiente manera:

1	2	3	4	5	n/j
					4
					3
		87			2
	75	20			1
67	8	79			0

Figura 2: Matriz cargada para días n = 1, n = 2 y n = 3.

Continuamos aplicando la misma lógica al resto de la matriz y ésta queda de la siguiente manera:

1	2	3	4	5	n/j
				115	4
			109	60	3
		87	54	119	2
	75	20	113	115	1
67	8	79	109	93	0

Figura 3: Matriz cargada completamente, se finalizó el algoritmo.

Donde el máximo valor encontrado en la columna final, representa el resultado óptimo que podríamos obtener teniendo disponibles 5 días de entrenamiento, en nuestro ejemplo sería 119.

Con todo este análisis, podemos deducir la ecuación de recurrencia, dadas las variables n y j definidas anteriormente queda de la siguiente manera:

$$E(j,n) = \begin{cases} 0 & si \quad n = 0 \\ OPT(n-2) + min(s_1, e_n) & si \quad j = 0 \\ E(j-1, n-1) + min(s_j, e_n) & si \quad j \neq 0, n \neq 0 \end{cases}$$

1.3. Reconstrucción de la solución total

Para construir la solución y saber qué días entrenar y cuáles no, simplemente debemos buscar el óptimo para el día n e ir reconstruyendo hacia atrás dependiendo del j, o sea considerando cuantos

días consecutivos de entrenamiento llevo en el día actual. Si me encuentro en $j = 0$, significa que el día anterior descansé, por lo que en esos casos, debería continuar en la posición correspondiente al día actual - 2. A partír de ahí simplemente se repite el paso inicial de buscar el óptimo, siendo definido el óptimo de n_i como el valor máximo de todas las celdas de la columna n_i.

Vemos en el ejemplo anterior la solución marcada:

1	2	3	4	5	n/j
				115	4
			109	60	3
		87	54	119	2
	75	20	113	115	1
67	8	79	109	93	0

Figura 4: Matriz con el camino de la resolución marcado. Los días que tengan una celda en amarillo se entrenará, los que no tengan se descansará.

Vemos que el óptimo para $n = 5$ está en la fila $j = 2$, eso quiere decir que entrenamos dos días consecutivos, o sea que se entrenó el día 4 y el día 3. Al llegar al día 3 llegamos a $j = 0$, eso quiere decir que el día anterior (el 2) descansamos, y, como no se puede descansar más de un día consecutivo, el día 1 también entrenamos.

En resumen, si Scaloni nos pregunta, deberíamos decirle que debe entrenar los días 1, 3, 4 y 5, con un descanso el día 2.

2. Algoritmo para encontrar la solución óptima

A continuación se detallan las implementaciones de los algoritmos propuestos para la resolución del problema de cómo maximizar la ganancia que se obtiene de los entrenamientos, teniendo en cuenta los descansos.

Contamos con dos algoritmos: por un lado armamos todas las soluciones y por otro buscamos la mejor de esas soluciones generando la reconstrucción.

2.1. Generación de matriz y obtención de mejor ganancia

Para facilitar el entendimiento del código, al recorrer el array de días lo hacemos empezando por 1 en vez de 0.

```
def pd_entrenamientos(si, ei):

    soluciones = [([0] * len(ei)) for _ in range(len(ei))]
    mejores_ganancias_en_conjunto_dias = []

    for dias_disponibles_para_entrenar in range(1, len(ei) + 1):

        mejor_ganancia_actual = 0

        for dias_seguidos_entrenando in range(0, len(si)):

            if dias_seguidos_entrenando > dias_disponibles_para_entrenar-1:
                break

            ganancia_para_energia_actual = min(
                # -1 porque el for de dias disponibles esta numerado de 1..n
                ei[dias_disponibles_para_entrenar - 1],
                si[dias_seguidos_entrenando]
            )
```

```
20
21              ganancia_anterior = 0
22
23              if dias_seguidos_entrenando == 0:
24                  if dias_disponibles_para_entrenar > 2:
25                      # -3 porque el for de dias disponibles esta numerado de 1..n
26                      ganancia_anterior = mejores_ganancias_en_conjunto_dias[
    dias_disponibles_para_entrenar - 3]
27
28              else:
29                  # -2 porque el for de dias disponibles esta numerado de 1..n
30                  ganancia_anterior = soluciones[dias_disponibles_para_entrenar - 2][
    dias_seguidos_entrenando - 1]
31
32              ganancia = ganancia_anterior + ganancia_para_energia_actual
33              soluciones[dias_disponibles_para_entrenar-1][dias_seguidos_entrenando]
    = ganancia
34
35              if ganancia > mejor_ganancia_actual:
36                  mejor_ganancia_actual = ganancia
37
38          mejores_ganancias_en_conjunto_dias.append(mejor_ganancia_actual)
39
40      print(
41          f"La mejor ganancia que se puede obtener entrenando es: {
    mejores_ganancias_en_conjunto_dias[-1]}"
42      )
43
44      return soluciones
```

La ganancia del día se obtiene del minimo entre el esfuerzo que demanda el entrenamiento de ese día y la energia según cuantos días consecutivos se esté planeando entrenar.

Si la cantidad de días consecutivos entrenados es 0 (es decir, el día anterior no figura entrenamiento) para calcular la nueva ganancia se debe tomar el mejor valor del último día entrenado, que luego se sumará junto con la ganancia del día actual. Para evitar tener que realizar una búsqueda de máximo cada vez que lleguemos a esta condición, en cada paso vamos guardando esa información en la lista de **mejores_ganancias_en_conjunto_de_dias**, asi al momento de necesitarla podemos acceder de manera más rápida y sin alterar la complejidad.

Para el caso en que la cantidad de días consecutivos sea distinto de cero, simplemente se tomará el valor que hace referencia a la cantidad consecutiva entrenada sin contar el día actual, que se encuentra en la posición anterior de la diagonal de la matriz.

El algoritmo retorna una matriz que representa la combinación entre días entrenados (consecutivos o no) y ganancias, no sin antes imprimir cual fue la mejor ganancia obtenida para el último día.

Finalmente, cabe destacar que la matriz generada es una matriz triangular inferior ya que los elementos por encima de la diagonal se encuentran en 0 por ser escenarios inválidos para nuestro problema ya que estariamos considerando datos de energia para días que no tenemos disponibles para entrenar.

2.2. Reconstrucción de los días a entrenar

Una vez generada la matriz, nos encontramos con el algoritmo para poder obtener cual es la combinación más óptima de todas las alternativas:

```
1 def pd_reconstruccion(soluciones):
2     filas = len(soluciones)
3     columnas = len(soluciones[0])
4
5     dias_entrenados = []
6     dias_consecutivos_por_visitar = -1
7
8     for i in range(filas - 1, -1, -1):
```

```
9
10          if dias_consecutivos_por_visitar == 0:
11              # El dia siguiente no se entrena pues son 0 los dias consecutivos
12              dias_consecutivos_por_visitar = -1
13              continue
14          else:
15              if dias_consecutivos_por_visitar == -1:
16                  # Busco la mayor ganancia para saber cuantos dias consecutivos se
    entrena
17                  ganancia_maxima_de_la_fila = float('-inf')
18                  for j in range(columnas - 1, -1, -1):
19                      if soluciones[i][j] > ganancia_maxima_de_la_fila:
20                          ganancia_maxima_de_la_fila = soluciones[i][j]
21                          dias_consecutivos_por_visitar = j
22              else:
23                  # Ya se que dias consecutivos se entrena
24                  dias_consecutivos_por_visitar -= 1
25
26              dias_entrenados.append(i + 1)
27
28      return list(reversed(dias_entrenados))
```

Para recorrer la matriz y obtener el resultado final es conveniente hacerlo de atrás hacia adelante para poder empezar con el mejor valor posible.

Primero se toma el mayor valor posible, una vez que tenemos ese valor, se analiza cómo se llegó a ese punto, es decir, cuantos días consecutivos se tuvieron que marcar para tener esa ganancia, este dato lo inferimos a través del índice de la fila en la que encontramos el valor máximo.

Desde ese punto, lo que hacemos es ir hacia atrás siguiendo esa información, tomaremos las ganancias por las que se fueron atravesando hasta finalizar esa cantidad de días consecutivos y luego, para continuar, tomaremos el siguiente máximo valor y así seguimos hasta terminar la matriz.

En cada paso del ciclo guardamos el día actual entrenado en un array para luego saber la mejor combinación.

Previo a retornar la combinación realizamos un reversed()[1] con el fin de visualizar el resultado en el orden correcto, ya que se encuentra en el orden inverso por la forma en la que recorremos la matriz.

3. Complejidad algorítmica

Para establecer el análisis de la complejidad, vamos a definir los valores n y m, siendo n la cantidad de entrenamientos disponibles y m la cantidad de datos de energias, dicho de otra manera, los tamaños de E_i y S_i respectivamente. Particularmente, sabemos que por la definición del problema, $n = m$, ya que no podemos resolver el problema si hubiera menos datos de energia que dias disponibles y en el caso de que $m > n$, los elementos de S_i sobrantes no aportarian a la resolución ya que no tiene sentido considerarlos.

3.1. Complejidad de obtención de óptimo

Teniendo en cuenta lo mencionado previamente, para nuestra solución utilizamos una matriz de tamaño $n\ x\ n$ para almacenar los resultados parciales, sumado a un arreglo para mantener el valor máximo obtenido por iteración, por lo que la complejidad espacial termina siendo $n^2 + n$, lo que se puede acotar a $\mathcal{O}(n^2)$.

Para la carga de datos se tienen dos ciclos anidados, el primero de tamaño n y el segundo de tamaño m, cabe destacar que el segundo ciclo siempre se recorre hasta el valor $j = n_i$ siendo n_i el valor de n actual en la iteración ciclo. Sin embargo, acotando superiormente hacia el infinito, este tamaño se pueden definir como $n\ x\ n$.

Dentro de los ciclos las operaciones que se hacen son:

1. Calculo del mínimo entre $s_i, ei = \mathcal{O}(1)$.
2. Consulta en el arreglo de valores máximos y asignación de valores = $\mathcal{O}(1)$.
3. Consulta de la matriz de soluciones para obtener ganancia anterior = $\mathcal{O}(1)$.
4. Suma de variables = $\mathcal{O}(1)$.
5. Actualización de la matriz de soluciones $\mathcal{O}(1)$.
6. Actualización de máximos = $\mathcal{O}(1)$.

Debido a que la suma de todas las operaciones dentro de los ciclos son constantes ($\mathcal{O}(k)$), estas se pueden acotar hacia $\mathcal{O}(1)$. Por esto, las operaciones totales se pueden escribir como:

$$\mathcal{O}(f(n) = \mathcal{O}(n) \cdot \mathcal{O}(n) = \mathcal{O}(n^2) \tag{1}$$

3.2. Complejidad de reconstrucción de días

Para la reconstrucción, nuestro algoritmo recorre la matriz de soluciones (n x n) de manera inversa, buscando el máximo en la última columna y accediendo consecutivamente a las filas y columnas anteriores ($col - 1, fila - 1$) hasta llegar a la posición de la primer columna, luego de eso vuelve a buscar el máximo en la columna siguiente a través de un segundo sub ciclo y repitiendo los mismos pasos iniciales hasta llegar a la última fila.

El algoritmo cuenta con un ciclo principal recorriendo las filas y luego un subciclo a la hora de buscar el máximo de una fila, dentro del primer ciclo las operaciones que se hacen son:

1. Actualización de variables = $\mathcal{O}(1)$.
2. Inserción en la lista de días = $\mathcal{O}(1)$.
3. Subciclo con comparación de valores para obtener el mayor en una fila si llegamos a la primer columna = $\mathcal{O}(n)$.

Estas operaciones dentro del ciclo principal se pueden acotar como $\mathcal{O}(k) + \mathcal{O}(n) = \mathcal{O}(n)$.

Finalmente, como la reconstrucción se hace en orden inverso, sumamos una operación final para invertir el orden de la lista, esto lo hacemos a través del método reversed[1] brindado por el lenguaje, este método según la documentación nos brinda un iterador inverso el cual terminamos usando para crear una nueva una nueva lista invertida, por lo que esta operación es $\mathcal{O}(n)$.

Por esto, en el peor de los casos, donde los tamaños de S_i sean grandes y los de E_i no nos alcancen para poder entrenar dos dias seguidos, obtendremos una solución donde nos va a convenir entrenar solo la mitad de los dias, es decir $\frac{n}{2}$. En este peor escenario, la cantidad de veces que tendremos que buscar el máximo será $\frac{n}{2}$, por lo que teniendo en cuenta que obtener el máximo nos cuesta $\mathcal{O}(n)$, la complejidad total nos queda de esta manera:

1. Iteraciones donde inserto el día en la lista sin buscar el máximo = $\mathcal{O}(\frac{n}{2})$.
2. Iteraciones donde tengo que buscar el máximo de una fila = $\mathcal{O}(\frac{n}{2}) \cdot \mathcal{O}(n)$.
3. Inversión de la lista solución = $\mathcal{O}(n)$.

Sumando las operaciones:

$$\mathcal{O}(f(n)) = \mathcal{O}(\frac{n}{2}) \cdot \mathcal{O}(n) + \mathcal{O}(n) \tag{2}$$

$$\mathcal{O}(f(n)) = \frac{1}{2} \cdot \mathcal{O}(n^2) + \mathcal{O}(n) = \mathcal{O}(n^2) \tag{3}$$

Y acotando:

$$\mathcal{O}(f(n)) = \mathcal{O}(n^2) \tag{4}$$

A través de este cálculo, podemos observar como la reconstrucción tiene una complejidad cuadrática en el peor caso. Cabe destacar que, en el mejor de los casos, donde no tengamos ningun descanso, la operación de buscar el máximo desaparece, en este escenario la complejidad total nos quedaría lineal $\mathcal{O}(n)$.

3.3. Medición de la complejidad algorítmica

Para verificar la complejidad algorítmica se generaron set de datos random con diferentes tamaños. Por cada tamaño de set de datos, se hicieron 10 mediciones con datos distintos y se tomó el promedio del tiempo de ejecución.

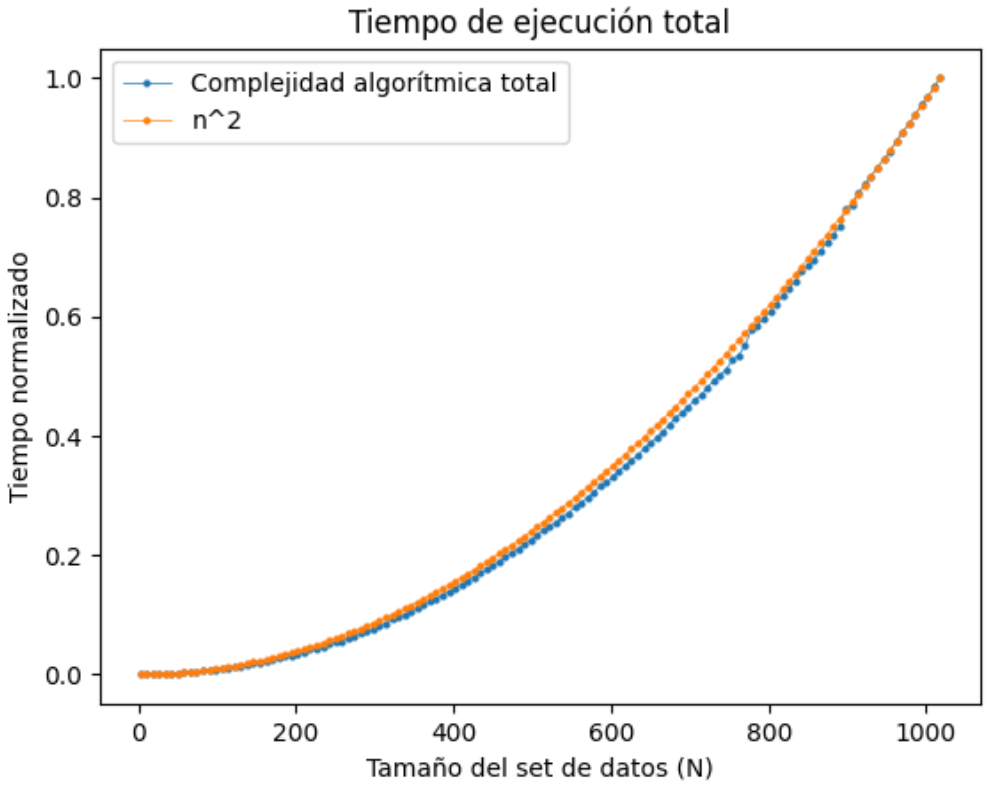
Se midió por separado el tiempo de ejecución para obtener el óptimo (t_o) y el tiempo de ejecución para la reconstrucción de días (t_r). El tiempo total queda entonces $t = t_o + t_r$

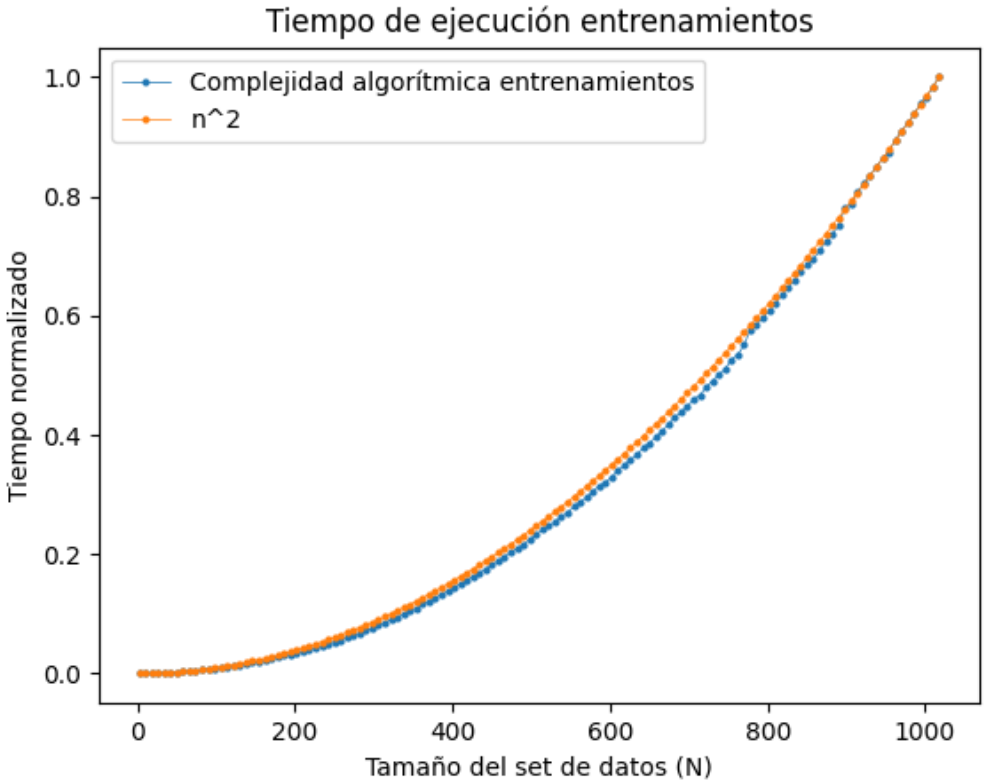
Tamaños de set de datos: $\{2 + 8n \ / \ n \in \mathbb{Z},\ 0 \leq n \leq 2^7\}$

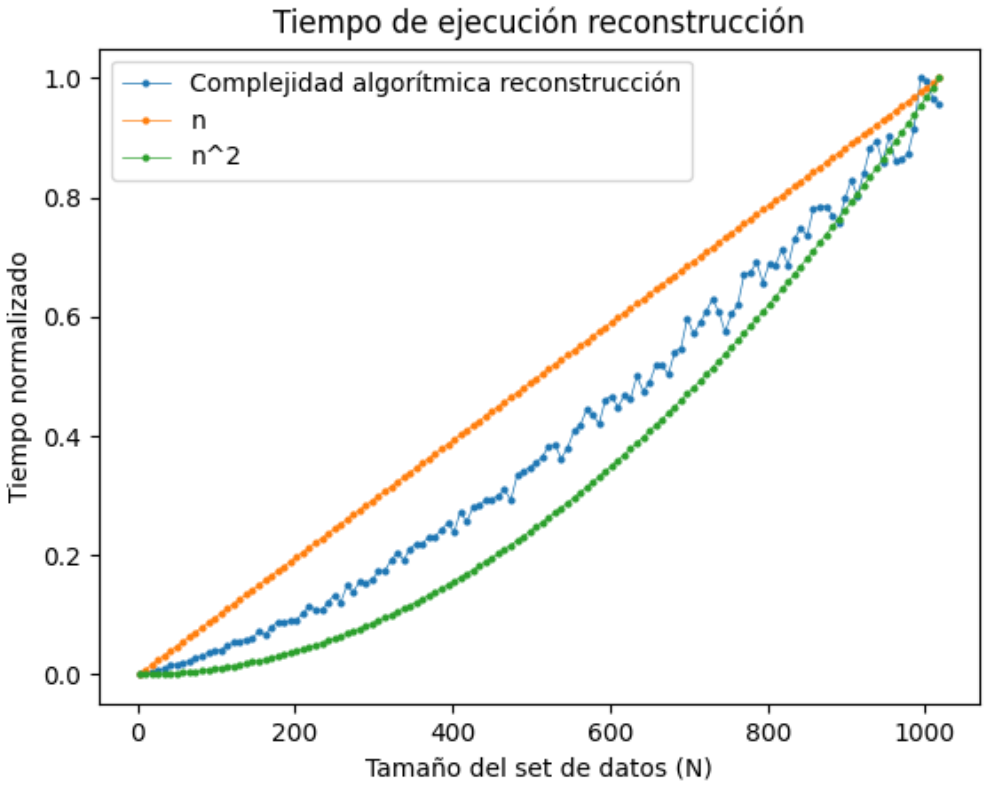
Cantidad de puntos: 128

Mediciones por punto: 10

Mediciones totales: 1280




Tiempo de ejecución entrenamientos
Complejidad algorítmica entrenamientos
n^2
Tiempo normalizado
Tamaño del set de datos (N)
0.0
0.2
0.4
0.6
0.8
1.0
0
200
400
600
800
1000


Tiempo de ejecución reconstrucción
Complejidad algorítmica reconstrucción
n
n^2
Tiempo normalizado
Tamaño del set de datos (N)
0.0
0.2
0.4
0.6
0.8
1.0
0
200
400
600
800
1000

Se puede ver que tanto el tiempo de ejecución total y de la obtención del óptimo tienen una complejidad de $\mathcal{O}(n^2)$ a pesar de que el tiempo de ejecución de la reconstrucción de días no sigue $\mathcal{O}(n^2)$. Esto se debe a que $t_r \ll t_o$.

En el gráfico de t_r se observa que la complejidad esta comprendida entre $\mathcal{O}(n)$ y $\mathcal{O}(n^2)$ tal como se explica en la sección **3.2**. La complejidad de este algoritmo es sensible a la variabilidad de los datos y a continuación se presentarán las mediciones en el mejor y peor de los casos.

3.3.1. Complejidad algorítmica en el mejor de los casos

El mejor de los casos para la reconstrucción de días sería que el óptimo consista en entrenar todos los dias, es decir $S_i > E_i$. De esta forma no hay que buscar el máximo en cada iteración.

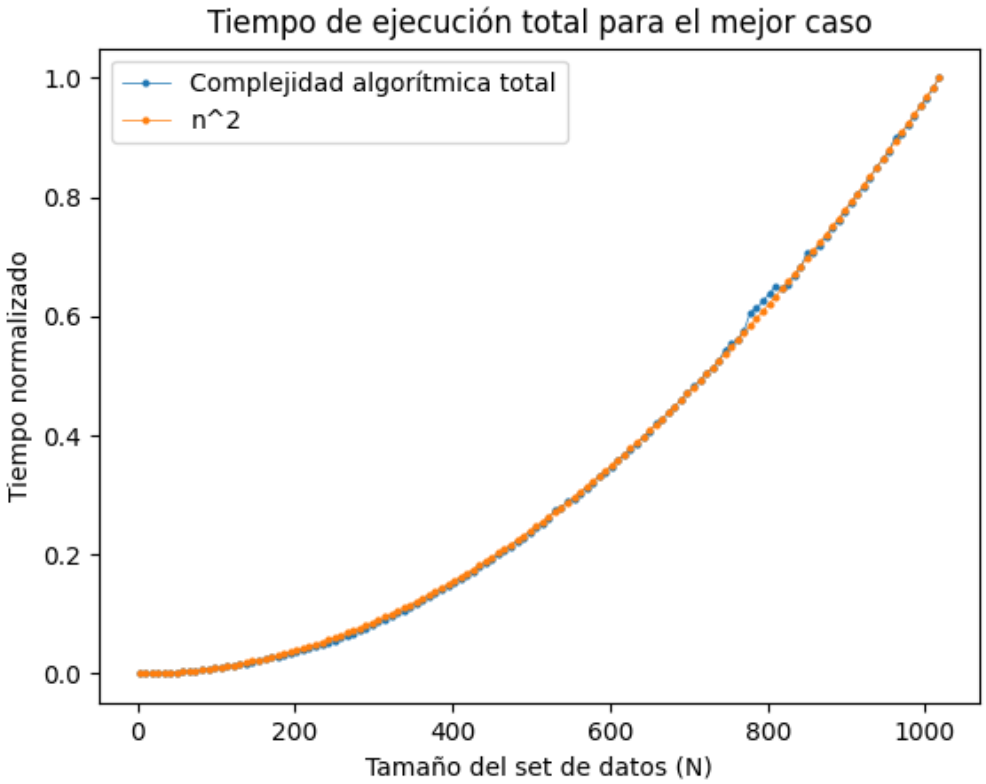
La generación de set de datos para este experimento consiste en poner valores mayores en S_i para cada E_i. Se generaron set de datos de distintos tamaños y por cada tamaño se midieron 10 set de datos distintos.

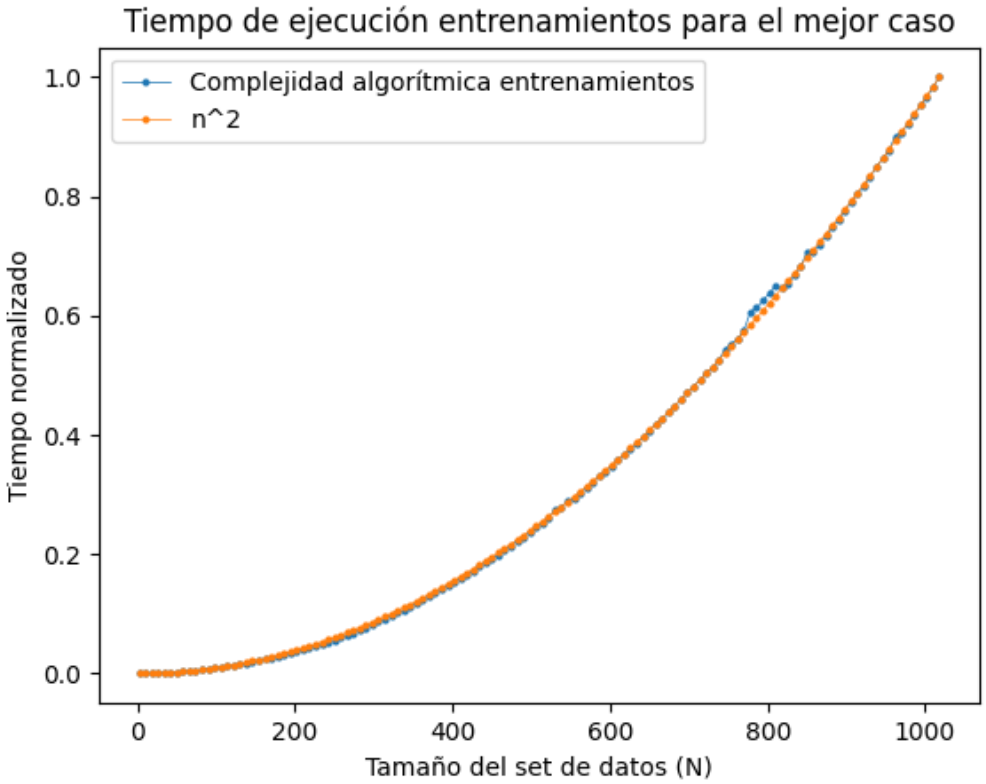
Tamaños de set de datos: $\{2 + 8n \ / \ n \in \mathbb{Z},\ 0 \leq n \leq 2^7\}$

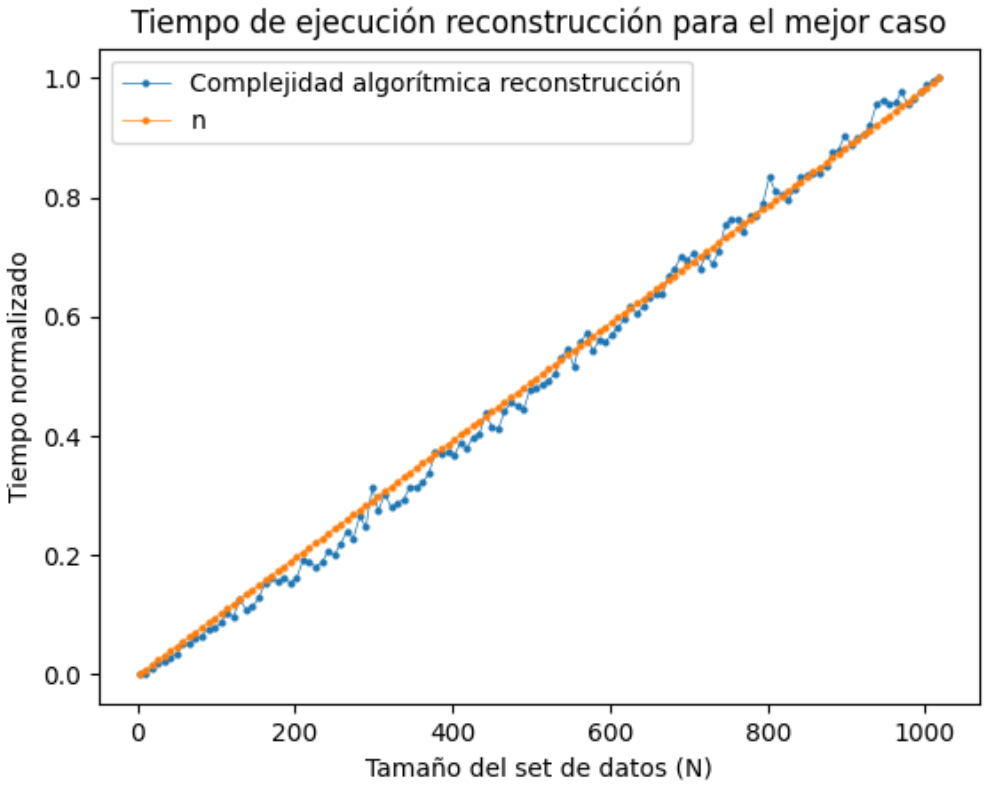
Cantidad de puntos: 128

Mediciones por punto: 10

Mediciones totales: 1280




Tiempo de ejecución entrenamientos para el mejor caso
Complejidad algorítmica entrenamientos
n^2
Tiempo normalizado
Tamaño del set de datos (N)
0.0
0.2
0.4
0.6
0.8
1.0
0
200
400
600
800
1000


Tiempo de ejecución reconstrucción para el mejor caso
Complejidad algorítmica reconstrucción
n
Tiempo normalizado
Tamaño del set de datos (N)
0.0
0.2
0.4
0.6
0.8
1.0
0
200
400
600
800
1000

Se puede apreciar que la complejidad algorítmica de la obtención del óptimo no se ve afectada por la variabilidad de datos pero sí el tiempo de ejecución de la reconstrucción de días. Para este escenario, la complejidad algorítmica de la reconstrucción de días queda en $\mathcal{O}(n)$.

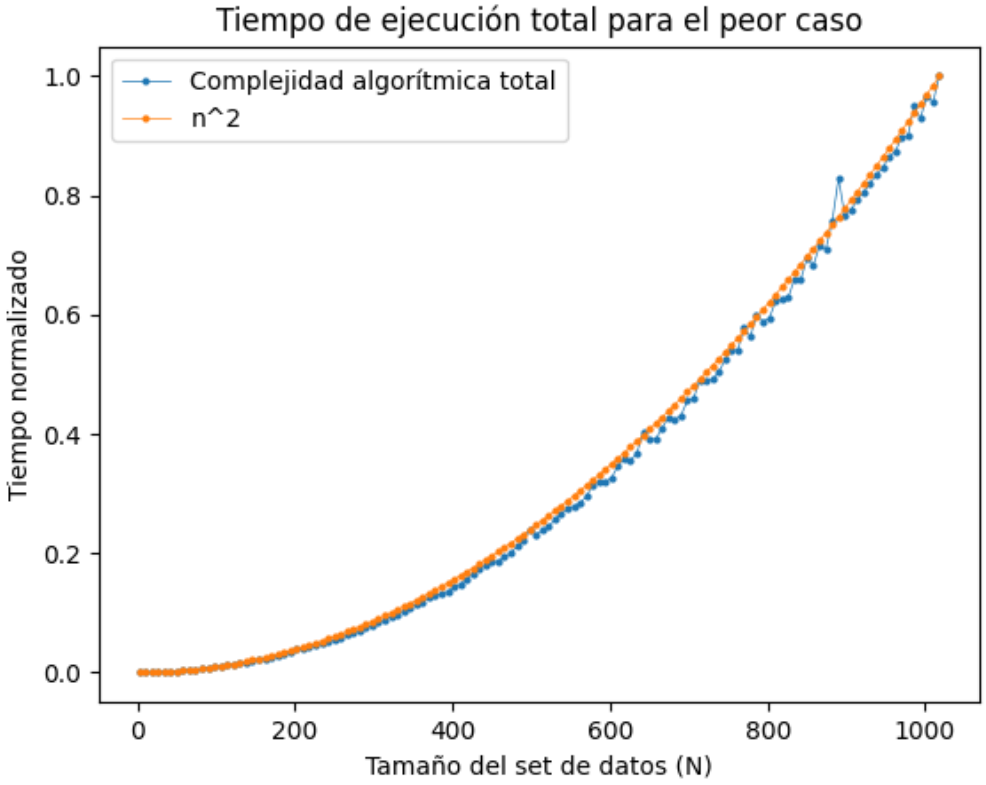
3.3.2. Complejidad algorítmica en el peor de los casos

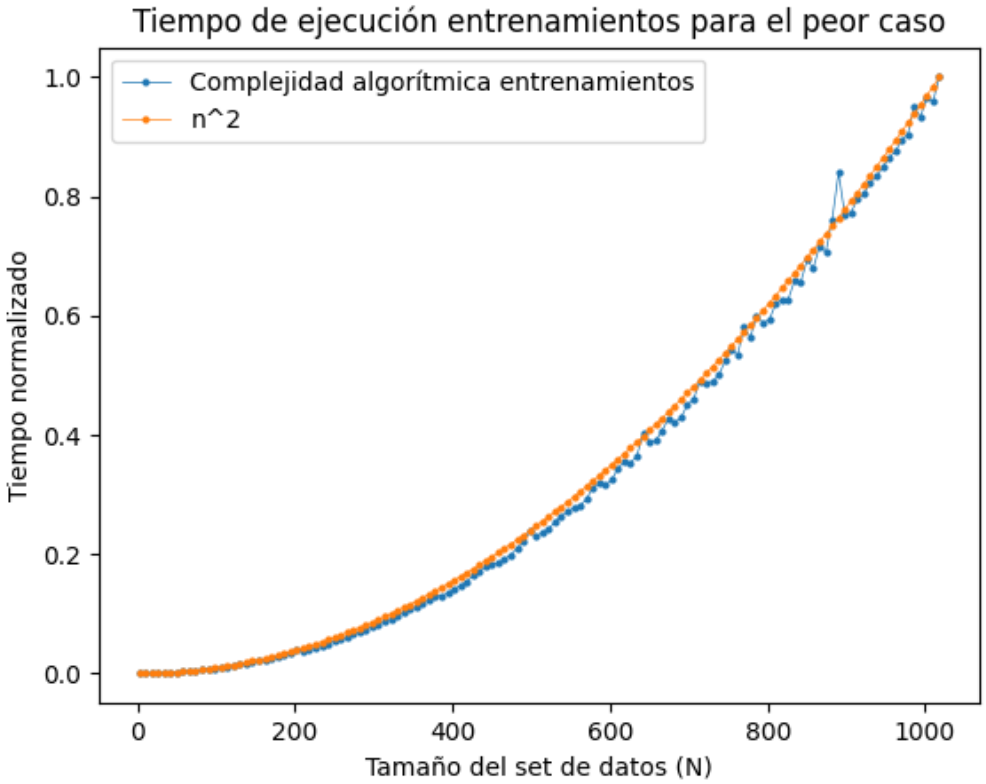
El peor de los casos para la reconstrucción de días sería no poder entrenar dos días seguidos, lo que forzaría la búsqueda del máximo la mayor cantidad de veces.

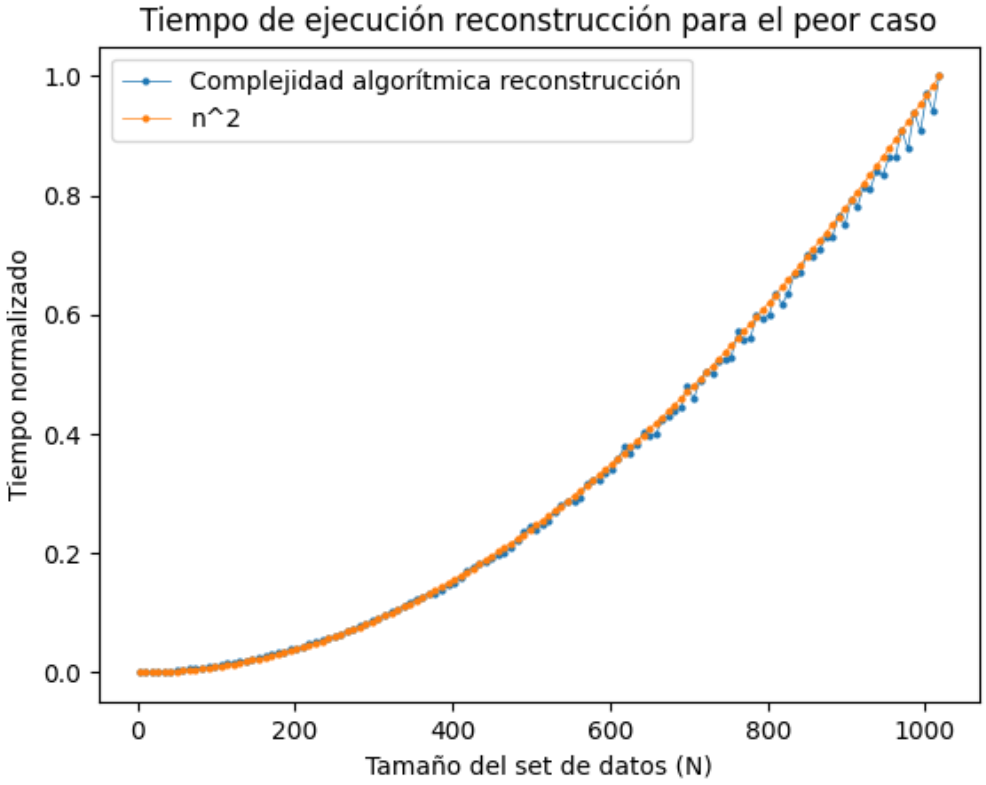
La generación de set de datos para este experimento consiste en poner un valor muy alto para cada E_i, y valores menores para cada S_i excepto el primero. Se generaron set de datos de distintos tamaños.

Tamaños de set de datos: $\{2 + 8n \ / \ n \in \mathbb{Z},\ 0 \leq n \leq 2^7\}$

Cantidad de puntos: 128




Tiempo de ejecución entrenamientos para el peor caso
Complejidad algorítmica entrenamientos
n^2
Tiempo normalizado
Tamaño del set de datos (N)
0.0
0.2
0.4
0.6
0.8
1.0
0
200
400
600
800
1000


Tiempo de ejecución reconstrucción para el peor caso
Complejidad algorítmica reconstrucción
n^2
Tiempo normalizado
Tamaño del set de datos (N)
0.0
0.2
0.4
0.6
0.8
1.0
0
200
400
600
800
1000

Al igual que en el escenario anterior, se puede apreciar que la complejidad algorítmica de la obtención del óptimo no se ve afectada por la variabilidad de datos pero sí el tiempo de ejecución de la reconstrucción de días. Para este escenario, la complejidad algorítmica de la reconstrucción de días queda en $\mathcal{O}(n^2)$.

4. Probando la solución

Se realizaron pruebas de efectividad del algoritmo. Estas se hicieron sobre sets de datos generados pseudoaleatoriamente, de tamaño relativamente chico para que sea posible resolverlos manualmente para luego comprobar esas resoluciones con las dadas por el algoritmo.

La metodología de la prueba es realizar un bucle sobre los distintos archivos de prueba, obtener de cada uno los e_i y s_i correspondientes y correr con ellos el algoritmo del problema. Éste nos devuelve la matriz con la solución, la cual procedemos a enviar a la función de reconstrucción de la solución.

La solución se nos devuelve como un vector con los días de entrenamiento. Se leen de un archivo .txt la resolución correcta al caso dado (calculada a mano por nosotros) y se compara el vector de la solución esperada con el de la solución obtenida. Si son iguales, se devuelve true.

Por último, se va guardando en un vector los booleanos devueltos por la función que realiza el procedimiento mencionado anteriormente. Al final del bucle, se calcula el porcentaje de valores true que tenemos en ese vector.

Se probaron 7 casos y la efectividad fue del 100 %.

La prueba de efectividad realizada se puede correr utilizando el siguiente comando:

```
python3 efectividad.py
```

Veremos que en la terminal aparece el siguiente mensaje:

```
Porcentaje de efectividad: 100.0%
```

5. Conclusiones

A través los análisis presentados en el presente informe, se pudo evidenciar que la resolución del problema requería tener en cuenta todo el conjunto de combinaciones de energias requeridas para los entrenamientos con la energia disponible para la cantidad de dias disponibles para entrenar. A través del uso de Memoization pudimos reaprovechar los resultados parciales a la hora de ir construyendo la solución haciendo uso de la ecuación de recurrencia definida. Por esto, nuestros algoritmos de solución y reconstrucción son ambos de complejidad $\mathcal{O}(n^2)$ donde pudimos probar que la variabilidad de los datos no afectaba a la optimalidad del resultado pero si afecta a la complejidad algoritmica del algoritmo de reconstrucción.

Referencias

[1] Python reversed built-in: https://docs.python.org/es/3/library/functions.html#reversed