.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Programación Dinámica

10 de octubre de 2023

Guillermo Stancanelli	Simón Stein	Santiago Tissoni
104244	106127	103856

1. Consideraciones

En el trabajo práctico se nos propone realizar un análisis al siguiente problema:

Se plantea la situación en la cual Scaloni y la selección Argentina tienen que decidir un plan de entrenamiento a realizar para los próximos n días. Cada entrenamiento, tiene una ganancia Ei y a su vez se cuenta con una determinada cantidad de energía S_j para realizarlo. La ganancia máxima que se puede obtener por cada día de entrenamiento resulta del mínimo entre el E_i actual y el S_j actual. Por otro lado, si Scaloni decide descansar el día actual, al día siguiente contará con energía S_1 (la energía del primer día). Es decir, al descansar, las energías se reinician para el día siguiente.

En este documento, plantearemos una solución que nos de la ganancia máxima que puede obtener la selección para n días con ganancias y energías E y S usando un algoritmo de Programación Dinámica. Realizaremos un análisis tanto de optimalidad como de complejidad del mismo, mostrando mediciones y casos de prueba.

2. Algoritmo para encontrar el tiempo mínimo de análisis

Realizaremos a continuación el análisis de diferentes algoritmos que fuimos explorando para la solución al problema planteado: encontrar la ganancia para e y s.

2.1. Primer acercamiento de Programación Dinámica: Top Down

Al pararnos frente al problema, decidimos encararlo en menores subproblemas. La idea para determinar la ganancia máxima es encontrar el óptimo a partir de decidir descansar o entrenar un día, y luego encontrar el óptimo para los siguientes $n-1$ días en función de si se entrenó o si se descansó. Es decir, calculamos la ganancia por entrenar y repetimos para el siguiente día, con la energía correspondiente. A su vez calculamos la ganancia de descansar y repetimos para el siguiente día, pero con energía S_1. Siguiendo esta lógica, podemos encontrar la funcion de recurrencia:

$$OPT(i,j) = max(min(E[i], S[j]) + OPT(i+1, j+1), OPT(i+1, 0))$$

Donde i es el día de entrenamiento con ganancia máxima e_i y j es el índice del arreglo de energía disponible s. De esta forma la ganancia óptima queda determinada por el óptimo entre el máximo de entrenar el dia i, con energía j, sumado al óptimo de los días siguientes, o descansar sumado al óptimo de los siguientes días con energía reiniciada a S_1.

Pasando esta ecuacion de recurrencia a código Python, obtenemos:

```
1
2  def g_max(e, s, n, i):
3      if n == n_max:
4          return min(e[n], s[i])
5      else:
6          entrenado = min(e[n], s[i]) + g_max(e, s, n + 1, i + 1)
7          descansado = 0 + g_max(e, s, n + 1, 0)
8          return max(entrenado, descansado)
```

Como podemos observar, este algoritmo obtiene la ganancia máxima para n, s y e de forma recursiva. Esto implica que el árbol de combinaciones con todas las soluciones posibles y sus subárboles, sean calculados reiteradas veces, obteniendo una complejidad temporal exponencial. Para mejorar la complejidad podemos optimizar el algoritmo memorizando óptimos anteriores:

```
1  gmaxs = {}
2
3  def g_max(e, s, n, i):
4      if n == n_max:
5          return min(e[n], s[i])
6      if (n, i) in gmaxs:
7          return gmaxs[(n, i)]
8      else:
9          entrenado = min(e[n], s[i]) + g_max(e, s, n + 1, i + 1)
10         descansado = 0 + g_max(e, s, n + 1, 0)
11         gmaxs[(n, i)] = max(entrenado, descansado)
12         return max(entrenado, descansado)
```

De esta forma, obtenemos una solución recursiva al problema de forma Top Down. Sin embargo, la solución sigue siendo recursiva. Buscaremos encontrar una solución óptima con un enfoque Bottom Up.

2.2. Solución Bottom Up: memorizar bidimensionalmente

Debido al alto nivel de complejidad que presenta el algoritmo realizado (exponencial), realizaremos un pasaje a un algoritmo iterativo con un enfoque bottom up. Nos basaremos en la idea de memorizar para guardar todas las combinaciones de subproblemas posibles, pero sin tener que recorrerlos de cero una y otra vez.

Para empezar, recordemos nuestra ecuación de recurrencia:

$$OPT(i,j) = max(min(E[i], S[j]) + OPT(i+1, j+1), OPT(i+1, 0))$$

La ganancia máxima del problema para un dia i dado y un índice de energía j, es el máximo entre entrenar ese día usando S_j y entrenar el dia siguiente usando $S_j + 1$ y no entrenar ese día pero el siguiente entrenarlo con S_1.

Teniendo esto en mente, podemos pensar en una forma para guardar nuestros datos de forma bidimensional teniendo en cuenta ambas variables i y j.

Proponemos una matriz G con ambas dimensiones $n+1$, donde cada celda $G[i][j]$ representa para el dia i, descansar hace j dias y entrenar con energía S_j ($S[j-1]$). Completando esta matriz celda por celda, obtendremos todas las combinaciones posibles de ganancia para n, e y s. Memorizando las anteriores soluciones en celdas previas, podremos ir acumulando la ganancia máxima para cada día. Luego basta con quedarse con la celda de valor máximo para el último día y obtendremos la solución al problema inicial.

A continuación, mostramos la implementación de la construcción de la matriz en lenguaje Python.

```python
def g_max(n, e, s):
    g = [[0 for _ in range(n + 1)] for _ in range(n + 1)]

    max_g = 0

    for i in range(1, n + 1):
        for j in range(i + 1):
            g[i][0] = max(g[i][0], g[i - 1][j])
            if j > 0:
                training_earning = min(e[i - 1], s[j - 1])
                g[i][j] = g[i - 1][j - 1] + training_earning

                if g[i][j] > max_g:
                    max_g = g[i][j]

    return g, max_g
```

Aplicando este algoritmo con los siguientes parámetros, obtenemos la siguiente matriz G:

$n = 3, e = [1, 5, 4], s = [10, 2, 2]$

	$j=0$	$j=1$	$j=2$	$j=3$
$i=3$	5	5	7	5
$i=2$	1	5	3	0
$i=1$	0	1	0	0
$i=0$	0	0	0	0

De esta manera, obtenemos todas las combinaciones posibles de ganancias y como ganancia máxima tenemos a $G[3][2] = 7$ que representa para el tercer día, descansar hace dos y entrenar con energía $S_1 = 2$.

Además, notamos que tanto la primera fila como los valores a la derecha de la diagonal son nulos. Esto se debe a que la ganancia máxima para 0 días con cualquier índice de energía es 0 y a que nunca nos encontraremos un caso donde $j > i$ ya que implicaría que para un dia i se entrene con un índice de energía superior al mismo.

Otra observación es que los valores que quedan en la diagonal son los valores de entrenar todos los días sin ningún descanso ya que en este caso los índices de e y s son iguales.

2.3. Reconstrucción de la solución

A partir de la matriz G generada de ganancias posibles, podemos recorrerla con cierto criterio para reconstruir la solución, y mostrar de esa forma cuáles días son los que efectivamente conviene

entrenar, y cuáles descansar. Para esto, simplemente nos basamos en buscar el máximo en la última fila de la matriz y ver su índice j. Una vez hecho esto, j tiene el valor de los días consecutivos entrenados, es decir para un $j = 2$, sabemos que se entrenó el último día, se entrenó el penultimo día, y se descansó el antepenultimo día. Luego simplemente nos movemos $j + 1$ filas (días) en el índice i, y repetimos este proceso, hasta que $i = 0$. Podemos ver a continuación una implementación de la reconstrucción de la solución en lenguaje python:

```python
def max_index(row):
    max = row[0]
    max_idx = 0
    for i in range(len(row)):
        if row[i] > max:
            max = row[i]
            max_idx = i

    return max_idx

def show_trained_days(g, n):
    i = n
    while i > 0:
        j = max_index(g[i])

        for t in range(j):
            print("Dia " + str(i) + ": Entrenamiento")
            i -= 1

        if i == 0:
            break

        print("Dia " + str(i) + ": Descanso")
        i = i -1
```

Por ejemplo, si tomamos denuevo la siguiente matriz resultante G:

$n = 3, e = [1, 5, 4], s = [10, 2, 2]$

	$j = 0$	$j = 1$	$j = 2$	$j = 3$
$i = 3$	5	5	7	5
$i = 2$	1	5	3	0
$i = 1$	0	1	0	0
$i = 0$	0	0	0	0

Entonces veremos que nuestra reconstrucción imprime:

```
Dia 3: Entrenamiento
Dia 2: Entrenamiento
Dia 1: Descanso
```

2.4. Análisis de complejidad

Como podemos observar, el algoritmo para el cálculo de la máxima ganancia resulta en una matriz de dimensiones $(n + 1)^2$, exceptuando las celdas a la derecha de la diagonal. Es decir recorremos aproximadamente la mitad de la matriz G.

La complejidad del mismo, resulta entonces:

$$\mathcal{T}(n) = \mathcal{O}(\frac{(n+1)^2}{2})$$

Desestimando las constantes, resulta en:

$$\mathcal{T}(n) = \mathcal{O}(n^2)$$

Si luego además queremos tener en cuenta la reconstrucción de la solución a partir de la matriz G, deberemos en el peor caso mirar todas las filas para buscar el índice de máximo valor. Esto se da para el caso donde se descansan todos los días menos el último de la temporada de entrenamientos. Debido a la dimensión nxn de la matriz, esto también resulta en:

$$\mathcal{T}(n) = \mathcal{O}(n^2)$$

Y sumando ambas partes del algoritmo y desestimando las constantes:

$$\mathcal{T}(n) = \mathcal{O}(n^2 + n^2) = O(2n^2) = O(n^2)$$

2.5. Análisis de variabilidad de e y s

A continuación, analizaremos cómo la variabilidad de los valores de e y s afectan la optimalidad y el tiempo de ejecución de nuestro algoritmo:

- Variabilidad de e (esfuerzos/ganancias por día):

  La variabilidad de e no afecta directamente el tiempo de ejecución del algoritmo. Esto se debe a que el algoritmo itera sobre todos los posibles valores de e sin importar su valor específico. La complejidad temporal está más vinculada a la longitud de e (es decir, el número de días) que a los valores individuales de e.

  La optimalidad tampoco es afectada por la variación de los valores de e. Esto es así porque el algoritmo busca todas las soluciones posibles de entrenamientos y descansos y luego obtiene el máximo. Por lo que siempre obtendrá la óptima solución.

- Variabilidad de s (energía disponible después de cada día sin descanso):

  Al igual que con e, la variabilidad de s en sí misma no tiene un impacto significativo en el tiempo de ejecución. Sin embargo, la longitud de s (idealmente, el mismo número que el de días) sí influye, ya que el algoritmo necesita considerar todas las combinaciones posibles de entrenamientos y descansos basándose en s.

  En cuanto a la optimalidad, sucede lo mismo que con e. Como mencionamos, el algoritmo considera todas las combinaciones y se queda con la de mayor ganancia, por lo que siempre obtendrá la solución óptima.

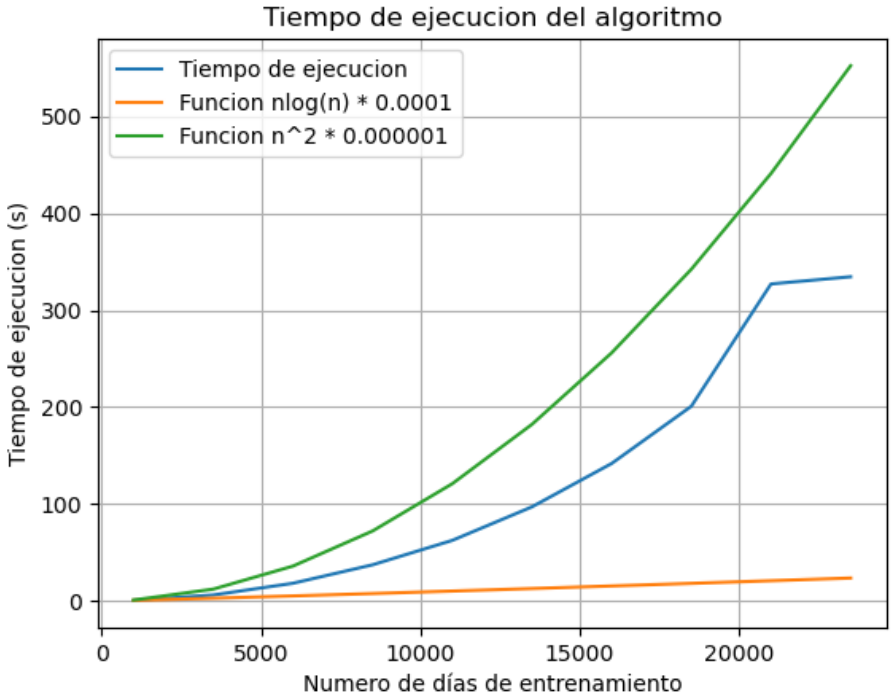
3. Mediciones

Para medir la complejidad temporal del algoritmo que propusimos para resolver el ejercicio, decidimos armar un script para la generación de casos de pruebas, aumentando la cantidad de días en intervalos de 2500, partiendo desde 1000 y finalizando en 26000. Para esto aprovechamos el módulo `random` de la librería estándar de Python para generar números aleatorios. Podemos ver a continuación el script utilizado:

```
1
2  def random_trainings_to_file(n, filename):
3      with open(filename, "w") as file:
4          file.write(str(n) + "\n")
5          random_gains = random_list(n)
6
7          for gain in random_gains:
8              file.write(str(gain) + "\n")
9
10         random_energies = random_list(n)
11         random_energies.sort(reverse=True)
12         for e in random_energies:
13             file.write(str(e) + "\n")
14
15 if __name__ == "__main__":
```

```
16      for i in range(1000, 26000, 2500):
17          filename = "testcase_" + str(i) + ".txt"
18          random_trainings_to_file(i, filename)
```

Una vez generados los archivos de prueba, los deserializamos y ejecutamos nuestro algoritmo de PD sobre los valores extraídos para hallar la matriz de ganancias G, midiendo el tiempo que tarda este último proceso. En el gráfico podemos ver como crece de manera $O(n^2)$ el tiempo de ejecución del algoritmo, además de una curva cuatrática y una del tipo $O(nlogn)$ que se presentan a fines de comparación y visualización.



4. Conclusiones

Durante el análisis del problema y las pruebas con los algoritmos planteados, nos dimos cuenta que un problema de programación dinámica no es inicialmente intuitivo de plantear. Identificar la dimensión de la memoization necesaria fue algo que nos llevó mucha prueba y error, y que nos indicó que nuestra primera ecuación de recurrencia no era correcta, tal como puede pasar en un primer intento del conocido "problema de la mochila". Una vez que se detectan los subproblemas dentro del problema mayor y la forma en la que estos se componen, resulta más intuitivo encontrar el caso base y la ecuación de recurrencia y convertirlos en código.

En nuestro caso, el concepto de ir memorizando las soluciones de los subproblemas definidos, nos ayudó a encontrar una solución que intuimos es más óptima en comparación a un algoritmo greedy.

Finalmente podemos decir que en nuestra opinión, reconstruir la solución a partir de la matriz resultante fue más fácil de idear que la generación inicial de la misma.