Diseño y Análisis de Algoritmos - Proyecto 1

Sebastián Robles 202411827
Juan Camilo Caldas 202322445

September 2025

1. Explicación

1.1. Ecuación de Recurrencia

$$f(i,c) = \begin{cases} 0 & \text{si } i = 0, \\ \max\limits_{\substack{cand \in \text{candList} \\ cand \le i}} \{f(i - cand, c-1) + \text{creatividad}[cand]\} & \text{si } i \ge 1 \\ -\infty & \text{si } c = 0 \end{cases}$$

Notación: **Notación:**

- $c :=$ celda de energía que se está iterando (k)
- $i :=$ número que se debe repartir (n)
- $cand :=$ candidato que se está verificando

1.2. Grafo de Dependencias

k = 5; n = 15; $P_1 = 1, P_2 = 2, P_3 = 3, P_4 = 4, P_5 = 5$;candidateList = [0, 3, 6, 9, 13]; candidateList ≤ 10

$c \backslash i$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
1	0	$-\infty$	$-\infty$	1	$-\infty$	$-\infty$	2	$-\infty$	$-\infty$	3	$-\infty$	$-\infty$	$-\infty$	1	$-\infty$	$-\infty$
2	0	$-\infty$	$-\infty$	1	$-\infty$	$-\infty$	2	$-\infty$	$-\infty$	3	$-\infty$	$-\infty$	4	1	$-\infty$	5
3	0	$-\infty$	$-\infty$	1	$-\infty$	$-\infty$	2	$-\infty$	$-\infty$	3	$-\infty$	$-\infty$	4	1	$-\infty$	5

Cuadro 1: Tabla de valores de $dp^{(c)}[i]$

Figura 1: Tiempo promedio en ms de resolución de casos.

1.3. Justificación de Aproximación

La solución que proponemos se compone de 3 principales partes, primero la busqueda de los 10 candidatos más significativos para n, el calculo preliminar de las creatividades de los candidatos, el dp que aparece en la ecuación de recurrencia.

En primera instancia, la selección candidadidatos se basa en el principio de que todo numero se puede descomponer en una suma de numeros donde máximo uno de los terminos contiene digitos no creativos. Además, con el fin de reducir aun más el numero de candidatos partimos del principio de que los candidatos mas pequeños de un numero solo aportan para generar uno de los candidatos más grandes. Es por este motivo que hemos decidido unicamente tomar los numeros con creatividad mayor que 0 (primer principio), y solo considerar los 10 más significativos ya que son los que verdaderamente aportan a la solución (segundo principio).

Para esto hemos implementado en la la función de generación de candidatos un max-heap con el cual a medida que se van encontrando candidatos se van ordenando de acuerdo a los que más creatividad aporten, para finalmente solo tomar los 10 más significativos. El hecho de usar un max-heap para esta implementación favorece la complejidad del algoritmo al ser una de las implementaciones más eficientes para una cola de prioridad. Finalmente, retornamos un arreglo para favorecer el acceso a estos candidatos a lo largo del dp.

En cuanto al calculo de creatividades nuestro objetivo era no tener que calcular la creatividad que aportan los candidatos cada vez que se emplearan, si no que lograran ser accesos menos costosos en terminos de tiempo como al almacenarlos en un arreglo. De esta manera evitamos hacer el ciclo while de los digitos de los numeros creativos constantemente y lo sustituimos por una operación sencilla como lo es la busqueda en un arreglo.

Finalmente, el dp que generamos aprovecha al maximo los principios antes mencionados para la generación de candidatos dado que, nuestra principal diferencia con respecto a la implementación de knapsack original que tiene complejidad de $O(kn^2)$ es que iteramos una menor cantidad de casillas del dp con respecto a la otra aproximación (descrita en alternativas consideradas). Como se puede apreciar en el grafo de necesidades, existe una gran cantidad de celdas que no aportan a la solución final ya que nunca toman una creatividad a partir de los candidatos iniciales. Estas celdas que no aportan a la solución son las que evitamos al solo iterar sobre las celdas que aportan a la solución, los valores de estas celdas son los que almacenamos en int[] explore, con el cual nos guiamos para decidir que celdas son las que nos falta iterar y que aporten a la solución. En cada celda iterada comparamos con nuestros candidatos depurados que en el peor de los casos son 10 y tomamos la creatividad maxima que pueda llegar a tener el numero, para que finalmente en la ultima celda obtengamos la creatividad asociada a este numero.

Por ende, es posible concluir que el algoritmo que presentamos toma los 10 candidatos más significativos para la solución, itera sobre k celdas disponibles, busca las creatividades más altas de los numeros que no se han visitado previamente de acuerdo con los candidatos obtenidos para finalmente obtener la creatividad del numero en la celda nk.

2. Análisis de Complejidad Temporal

2.1. Calculo de Creatividad

$$T_1(n,P) = 2c_1 + d_n * (4c_1 + 5c_2 + 5c_3 + c_{getArray}); d_n := \text{cantidad digitos}$$
$$T_1(n,P) = O(d_n); 1 \leq d_n \leq 7$$

2.2. Calculo de Candidatos

$$T_2(n,P) = n \cdot (c_1 \cdot T_1(n,P)) + 2c_1 + n \cdot \big(3c_1 + 2c_{\text{getArray}} + 4c_2 + T_{\text{insert}}(\text{KEEP}) + T_{\text{remove}}(\text{KEEP})\big) + \text{KEEP} \cdot \big(\text{KEEP} \cdot T_{\text{remove}}(\text{KEEP})\big) + T_{\text{sort}}(\text{KEEP})$$

$$T_2(n,P) = O(d_n \cdot n) + 2c_1 + O(n) + O(1) + O(1) = O(d_n \cdot n) = O(n); \text{con KEEP :=cte =10} \wedge \max(d_n)\text{=7} \Rightarrow \text{despreciable}$$

2.3. DP main

$$T_3(n,k,P) = 4c_1 + T_2(n,P) + n \cdot T_1(n,P) + c_1 + n \cdot c_{\text{fill}} + k\Big(3c_1 \cdot O(n) + O(n) + n \cdot \big(c_2 + \text{cand} \cdot (4c_1 + 2c_3 + 5c_{\text{getArray}} + 2c_2 + T_{\text{addArrayList}}(\text{cand}))\big) + c_2 + c_{\text{arrayListToArray}}\Big)$$

(Usando $T_2(n,P) = O(n)$, $T_1(n,P) = O(d_i)$, $T_{\text{addArrayList}}(\text{cand}) = O(\text{cand})$, y $c_{\text{getArray}} = O(1)$):

$$T_3(n,k,P) = 4c_1 + O(n) + n \cdot O(d_i) + c_1 + O(n) + k\Big(3c_1 \cdot O(n) + O(n) + n \cdot (c_2 + O(\text{cand}^2)) + c_2 + O(n)\Big)$$

(Agrupando y simplificando términos $O(\cdot)$):

$$T_3(n,k,P) = O(n) + n \cdot O(d_i) + k \cdot (O(n) + O(n \cdot \text{cand}^2)) = O(n) + O(d_i \cdot n) + k \cdot O(n) + O(n \cdot \text{cand}^2)$$

(Como cand $\leq KEEP = 10$ es constante):

$$T_3(n,k,P) = O(d_i \cdot n) + k \cdot O(n) = O(k \cdot n).$$

En conclusión se puede observar como de acuerdo con el análisis de complejidad temporal, los terminos más significativos en cuestión de tiempo de ejecución del algoritmo propuesto son n y k, ya que para valores de n mayores a 40, los candidatos se vueleven constantes al ser máximo 10. No obstante, se puede observar en el análisis como los candidatos a explorar entre n y k afectan considerablemente la complejidad del algoritmo por ende en casos donde la diferencia entre n y k sea muy grande el algoritmo podria llegar a tardarse una cantidad de tiempo mucho más cercana a la aproximación de knapsack ïngenuo".

3. Complejidad Espacial

La complejidad espacial del algoritmo es

$$S(n,C) = O(n+C) = O(n),$$

donde n es el número objetivo y C el número de candidatos creativos ($C \leq n$) $\Rightarrow$ C=n en el peor caso.

Estructuras de Datos Principales

- **Arreglo DP** `int[] dp` $\rightarrow O(n)$: almacena la máxima creatividad para cada suma.
- **Arreglo de creatividad** `int[] creatividad` $\rightarrow O(n)$: cachea valores precalculados. (memoización)
- **Lista de candidatos** `int[] candList` $\rightarrow O(C)$: contiene solo los números con creatividad positiva.
- **Auxiliares temporales** (`next`, `explore`) $\rightarrow O(n)$ como máximo.

Por lo tanto, la complejidad espacial se mantiene en $O(n)$ independientemente del valor que tenga k, lo que representa una mejora significativa sobre implementaciones más básicas del problema knapsack.

4. Alternativas Consideradas

4.1. DP Knapsack con Restricción

$$DP[i][j] = \begin{cases} 0 & i = 0 \wedge j = 0 \\ 0 & i = 0 \wedge j > 0 \\ 0 & j = 0 \wedge i > 0 \\ \text{máx}(DP[i][j], DP[i-1][j-c] + creatividad) & c \leq j \wedge DP[i-1][j-c] \neq -1 \end{cases}$$

$$\text{Complejidad: } O(k \cdot n^2) \approx O(n^3); \quad \text{En el peor caso } k = n$$

La complejidad de este algoritmo era muy alta considerando el peor caso, es por este motivo que se descartó la solución. En casos en los que el DP era de 10^7 el algoritmo tardaba más de 40 segundos y los casos de 10^9 ni siquiera era capaz de ejecutarlos. Esta aproximación era claramente inviable para una solución.
Además se trato de solucionar, los problemas almacenando el máximo hasta el momento o recorriendo solamente los de la misma fila de k modificando un poco el código para arrastrar creatividades. Esta solución presentaba varios fallos en cierto tipo de números con determinados dígitos para implementaciones ."$^{\text{ef}}$icientes", para implementaciones que aseguraran la correctitud de las respuestas era inviable visitar tantas casillas del dp.

4.2. Algoritmo Greedy

En este último caso también manteníamos la ecuación de recurrencia presentada en 4.1, pero con una heurística greedy para casos donde $k \geq 1000$ o $n > 10000$. El algoritmo selecciona las k creatividades más altas disponibles tras verificar factibilidad mediante DP, reduciendo la complejidad de $O(k \cdot n^2)$ a $O(n^2)$. Sin embargo, presenta limitaciones evidentes: al evaluar el paso de $n = 1000$ a $n = 1001$, se observa un salto anómalo en la creatividad de 330 a 3600, matemáticamente imposible dado que la diferencia está acotada por los valores P_i. Este contraejemplo demuestra que la estrategia greedy no garantiza optimalidad y puede producir resultados inconsistentes.