Dynamic Programing

2.a)

Para representar matematicamente as funções recursivas `maxValue(i, k)` e `lastItem(i, k)` para resolver o problema da mochila (knapsack problem), podemos definir da seguinte forma:

1. maxValue(i, k):
   - Se `i = 0`, então `maxValue(i, k) = 0`.
   - Se `weight[i] > k`, então `maxValue(i, k) = maxValue(i - 1, k)`.
   - Caso contrário, `maxValue(i, k) = max(value[i] + maxValue(i - 1, k - weight[i]), maxValue(i - 1, k))`.
2. lastItem(i, k):
   - Se `i = 0`, então `lastItem(i, k)` não está definido.
   - Se `weight[i] > k`, então `lastItem(i, k) = lastItem(i - 1, k)`.
   - Caso contrário, se `maxValue(i, k) = value[i] + maxValue(i - 1, k - weight[i])`, então `lastItem(i, k) = i`.
   - Caso contrário, `lastItem(i, k) = lastItem(i - 1, k)`.

Essas definições matemáticas capturam a natureza recursiva do problema da mochila. Elas descrevem como o valor máximo e o índice do último item utilizado são calculados com base nas soluções dos subproblemas menores.

2.b)

```
def knapsackDP(values, weights, n, maxWeight):
    # Inicialize a tabela de valores e a tabela de índices
    dp_maxValue = [[0] * (maxWeight + 1) for _ in range(n + 1)]
    dp_lastItem = [[0] * (maxWeight + 1) for _ in range(n + 1)]

    # Preencha a tabela de valores e a tabela de índices
    for i in range(1, n + 1):
        for k in range(1, maxWeight + 1):
            if weights[i - 1] > k:
                dp_maxValue[i][k] = dp_maxValue[i - 1][k]
                dp_lastItem[i][k] = dp_lastItem[i - 1][k]
            else:
                if values[i - 1] + dp_maxValue[i - 1][k - weights[i - 1]] > dp_maxValue[i - 1][k]:
                    dp_maxValue[i][k] = values[i - 1] + dp_maxValue[i - 1][k - weights[i - 1]]
                    dp_lastItem[i][k] = i
                else:
                    dp_maxValue[i][k] = dp_maxValue[i - 1][k]
```

```
                dp_lastItem[i][k] = dp_lastItem[i - 1][k]

    # Reconstrua a solução
    result = []
    i = n
    k = maxWeight
    while i > 0 and k > 0:
        if dp_lastItem[i][k] != dp_lastItem[i - 1][k]:
            result.append(1)
            k -= weights[i - 1]
        else:
            result.append(0)
        i -= 1

    # Inverta a lista de resultados, pois a construímos de trás para frente
    result.reverse()

    return result, dp_maxValue[n][maxWeight]

# Entrada de exemplo
values = [10, 7, 11, 15]
weights = [1, 2, 1, 3]
n = 4
maxWeight = 5

# Calcula os valores e índices máximos
result, max_value = knapsackDP(values, weights, n, maxWeight)

print("Result:", result)
print("Total value:", max_value)
```

2.d)

Vamos analisar a complexidade temporal e espacial do algoritmo `knapsackDP`:

1. Complexidade Temporal:
    - O preenchimento da tabela de programação dinâmica requer um loop duplo sobre o número de itens `n` e a capacidade máxima da mochila `maxWeight`, resultando em uma complexidade de tempo de O(n * maxWeight).
    - O loop de reconstrução da solução para determinar quais itens foram selecionados tem uma complexidade de tempo de O(n).
    - Portanto, a complexidade temporal total é de O(n * maxWeight).
2. Complexidade Espacial:
    - A tabela de memoização (`dp`) tem dimensões `(n + 1) x (maxWeight + 1)`, resultando em uma complexidade espacial de O(n * maxWeight).

- A matriz de rastreamento (`trace`) tem a mesma dimensão, resultando em uma complexidade espacial adicional de O(n * maxWeight).
- Além disso, o vetor `usedItems` para rastrear os itens selecionados tem um tamanho de `n`, contribuindo com uma complexidade espacial adicional de O(n).
- Portanto, a complexidade espacial total é de O(n * maxWeight).

Em resumo, o algoritmo `knapsackDP` possui uma complexidade temporal e espacial de O(n * maxWeight), onde `n` é o número de itens e `maxWeight` é a capacidade máxima da mochila. Isso faz com que o algoritmo seja eficiente para casos de entrada moderadamente grandes, mas pode se tornar impraticável para valores muito grandes de `n` ou `maxWeight`.

3.1)

A fórmula recursiva para a distância de edição (ou distância de Levenshtein) entre duas strings A e B, denotada por $D(i, j)$, pode ser definida da seguinte forma:

$$D(i,j) = \begin{cases} 0 \\ i \\ j \\ \min \begin{cases} D(i-1,j)+1 & \text{(deleção)} \\ D(i,j-1)+1 & \text{(inserção)} \\ D(i-1,j-1)+(A[i] \neq B[j]) & \text{(substituição o} \end{cases} \end{cases}$$

Nesta fórmula:

- $D(i-1,j)+1$ representa o custo de deleção, ou seja, transformar a substring $A[0:i-1]$ em $B[0:j]$ e, em seguida, excluir o caractere $A[i]$.
- $D(i,j-1)+1$ representa o custo de inserção, ou seja, transformar a substring $A[0:i]$ em $B[0:j-1]$ e, em seguida, adicionar o caractere $B[j]$.
- $D(i-1,j-1)+(A[i] \neq B[j])$ representa o custo de substituição, que é 1 se os caracteres $A[i]$ e $B[j]$ forem diferentes e 0 se forem iguais.

Essa fórmula baseia-se na ideia de que a distância de edição entre duas strings pode ser obtida combinando as distâncias de edição de substrings menores.

3.b)

Para calcular a distância de edição entre "money" e "note", podemos usar a fórmula recursiva que discutimos anteriormente. Aqui está a computação passo a passo:

1. **Inicialização**:
   - $D(0,0) = 0$ (strings vazias)
   - $D(i,0) = i$ para $i > 0$ (string B vazia)
   - $D(0,j) = j$ para $j > 0$ (string A vazia)
2. **Cálculo da distância de edição**:
   - Para cada posição (i,j) na matriz D, onde i percorre os caracteres de "money" e j percorre os caracteres de "note":
     - Se $A[i] = B[j]$, então $D(i,j) = D(i-1,j-1)$ (sem operação necessária)
     - Caso contrário, $D(i,j)$ é o mínimo entre:
       - $D(i-1,j) + 1$ (deleção)
       - $D(i,j-1) + 1$ (inserção)
       - $D(i-1,j-1) + 1$ (substituição)
3. **Resultado**:
   - A distância de edição entre "money" e "note" é o valor em $D(5,4)$.

Vamos calcular isso:

$$D(0,0) = 0$$
$$D(1,0) = 1$$
$$D(2,0) = 2$$
$$D(3,0) = 3$$

Vamos calcular isso:

$$
\begin{aligned}
D(0,0) &= 0 \\
D(1,0) &= 1 \\
D(2,0) &= 2 \\
D(3,0) &= 3 \\
D(4,0) &= 4 \\
D(5,0) &= 5 \\
D(0,1) &= 1 \\
D(0,2) &= 2 \\
D(0,3) &= 3 \\
D(0,4) &= 4 \\
D(i,j) &= \min \begin{cases} D(i-1,j)+1 & \text{(deleção)} \\ D(i,j-1)+1 & \text{(inserção)} \\ D(i-1,j-1)+1 & \text{(substituição)} \end{cases}
\end{aligned}
$$

Agora, preenchemos a tabela:

	0	1	2	3	4
0	0	1	2	3	4
1	1	1	2	3	4
2	2	2	1	2	3
3	3	3	2	1	2
4	4	4	3	2	1
5	5	5	4	3	2

Portanto, a distância de edição entre "money" e "note" é $D(5,4) = 2$.

3.d)

A complexidade temporal e espacial da função `editDistance` em relação aos comprimentos dos strings A e B (|A| e |B|, respectivamente) é

�(|�|×|�|)

$O(|A| \times |B|)$.

1. Complexidade Temporal: A função preenche uma matriz de dimensões
2. (|�|+1)×(|�|+1)
3. $(|A|+1) \times (|B|+1)$ com base nos comprimentos dos strings A e B. O loop duplo utilizado para preencher esta matriz percorre todas as células uma vez. Para cada célula, são realizadas operações de comparação e atribuição constantes. Portanto, o tempo de execução é proporcional ao produto dos comprimentos dos dois strings, resultando em uma complexidade de
4. �(|�|×|�|)
5. $O(|A| \times |B|)$.
6. Complexidade Espacial: A função utiliza uma matriz de tamanho
7. (|�|+1)×(|�|+1)
8. $(|A|+1) \times (|B|+1)$ para armazenar os valores da distância de edição entre os prefixos dos strings A e B. Como essa matriz é o principal uso de memória adicional na função e seu tamanho é proporcional ao produto dos comprimentos dos dois strings, a complexidade espacial também é
9. �(|�|×|�|)
10. $O(|A| \times |B|)$.