Longest Max Min Subsequence Round 967

Leonardo Javier Ramirez Calatayud C411

September 2024

Índice

1. Definición del problema

Dado una secuencia de enteros $a_1, a_2, \ldots, a_n$.
Sea S el conjunto de todas las subsecuencias no vacías posibles de a sin elementos duplicados. El objetivo es encontrar la secuencia más larga en S. Si hay varias de ellas, encuentra la que minimice el orden lexicográfico después de multiplicar los términos en posiciones impares por -1.

Por ejemplo: dado
$a = [3, 2, 3, 1]$,
$S = \{[1], [2], [3], [2, 1], [2, 3], [3, 1], [3, 2], [2, 3, 1], [3, 2, 1]\}$.

Entonces, $[2, 3, 1]$ y $[3, 2, 1]$ serían las más largas, y $[3, 2, 1]$ sería la respuesta, ya que $[-3, 2, -1]$ es lexicográficamente menor que $[-2, 3, -1]$.

Una secuencia c es una subsecuencia de una secuencia d si c puede obtenerse de d mediante la eliminación de varios (posiblemente, cero o todos) elementos.

Una secuencia c es lexicográficamente menor que una secuencia d si y solo si se cumple una de las siguientes condiciones:

1. c es un prefijo de d, pero $c \neq d$.
2. En la primera posición donde c y d difieren, la secuencia c tiene un elemento más pequeño que el elemento correspondiente en d.

2. Solución

2.1. Observación:

La longitud máxima del arreglo final es igual a la cantidad de números diferentes del arreglo.

2.1.1. Demostración:

Suponiendo que existe k tal que el arreglo de longitud máxima tiene tamaño k y existen $k + 1$ elementos distintos. Luego tomamos el elemento distinto que no pertenece al arreglo máximo y lo agregamos, obteniendo un nuevo máximo, por lo que el arreglo inicial no era máximo. Entonces, la longitud máxima es igual al número mínimo de elementos distintos del arreglo.

2.2. Greedy:

Una vez conocida la longitud del arreglo final, nuestro siguiente paso es lograr que los elementos de las posiciones $1, 3, \ldots, k+1$ sean los máximos posibles, mientras que los $2, 4, \ldots, 2k$ sean los mínimos posibles. Para ello, utilizaremos una estrategia *greedy* de forma tal que en cualquier punto de nuestro problema se tenga *la mejor solución hasta el momento.*

2.3. Definiendo *"la mejor solución hasta el momento"*

"La mejor solución hasta el momento k" se podría definir como la subsecuencia encontrada desde $a_1, \ldots, a_k$ tal que sea la menor lexicográficamente y que, de haber elementos en el arreglo $a_1, \ldots, a_k$ que no pertenezcan a la subsecuencia obtenida, entonces la cantidad de elementos de dicho valor que se encuentran en $a_{k+1}, \ldots, a_n > 0$ (a esto último lo llamaremos $\boldsymbol{cnt}(a_i)$).

2.4. Agregando el elemento a_{k+1}

Una vez calculada la subsecuencia hasta el índice k, al observar a_{k+1}, este solo podría ocupar la posición de $S_{k'}, S_{k'-1}, \ldots, S_{k'-t}$, donde $S_{k'}$ es el último elemento agregado a la subsecuencia hasta el momento y $S_{k'-t-1}$ es el último elemento en la subsecuencia con cnt = 0, puesto que si a_{k+1} ocupa el lugar de $S_{k'-t-1}$, entonces se incumpliría 2.3.

2.5. Observación:

- *Si el elemento observado a_{k+1} no optimiza $S_{k'}$ ni $S_{k'-1}$, entonces no puede optimizar $S_{k'-t}$ con $t \geq 2$, puesto que la subsecuencia S ya está optimizada.*
- *Por otra parte, si a_{k+1} ya se encuentra en la subsecuencia, simplemente podemos ignorarlo, pues ya el elemento se encuentra optimizado en S.*

2.6. Construyendo la solución:

Para construir la solución final, se procede de la siguiente manera:

1. *Inicialmente, se define un arreglo vacío S que contendrá la subsecuencia óptima.*
2. *Se itera sobre cada elemento de la secuencia original $a = [a_1, a_2, \ldots, a_n]$. Para cada elemento a_i, se evalúa si este puede ser añadido al arreglo S de acuerdo con las condiciones establecidas en las observaciones anteriores.*

3. *Si a_i puede ocupar una posición en S de acuerdo con la subsección 2.4, se añade a S en la posición correspondiente. Esto se realiza asegurando que S mantenga la propiedad de ser la menor subsecuencia lexicográficamente posible, con los elementos en posiciones impares multiplicados por -1.*

4. *Durante la construcción de S, se debe mantener el control de la cantidad de elementos restantes de cada valor en el arreglo original a, para asegurar que las decisiones tomadas son óptimas. Este control se realiza utilizando la función $cnt(a_i)$, que cuenta cuántos elementos de valor a_i quedan por procesar en las posiciones posteriores a i.*

5. *Si a_i no optimiza ninguna de las posiciones candidatas en S, se ignora y se continúa con el siguiente elemento de la secuencia a.*

6. *Al final del proceso, el arreglo S contendrá la subsecuencia más larga que cumple con las condiciones establecidas, siendo además la mínima en orden lexicográfico.*

3. Código

3.1. Comentarios y nombres de variables

- *better(value, position, answer): Función para verificar si un valor es mejor para una posición dada en la subsecuencia.*
- *n: Longitud del arreglo.*
- *array: Arreglo de números.*
- *count: Lista para contar la frecuencia de cada número en el arreglo.*
- *answer: Lista para almacenar la subsecuencia resultante.*
- *in_answer: Lista booleana para verificar si un número está en la subsecuencia.*
- *pointer: Puntero para la posición actual en la subsecuencia.*

3.2. Complejidad temporal

La complejidad de la solución propuesta es $O(n^2)$ debido al ciclo for anidado con el ciclo while.

4. Anexos

4.1. Código fuente

```
def better(value, position, answer):
    # Check if the value is better for the given position
    if position % 2 == 1:
        return value < answer[position]
    return value > answer[position]

def main():
    import sys
    input = sys.stdin.read
    data = input().split()

    index = 0
    t = int(data[index])  # Number of test cases
    index += 1

    results = []

    for _ in range(t):
        n = int(data[index])  # Length of the array for this test case
        index += 1

        array = list(map(int, data[index:index + n]))  # Read the array
        index += n

        count = [0] * (max(array) + 1)  # Frequency count of each number in the array
        answer = [0] * n  # Array to store the resulting subsequence
        in_answer = [False] * (max(array) + 1)  # Boolean array to check if a number is in the answer
        pointer = 0  # Pointer to the current position in the answer array

        # Count the frequency of each number in the array
        for num in array:
            count[num] += 1

        # Process each number in the array
        for num in array:
            count[num] -= 1  # Decrease the count as we process the number

            if pointer == 0:
                # If the answer array is empty, add the number
                answer[pointer] = num
                pointer += 1
                in_answer[num] = True
            else:
                if in_answer[num]:
                    # If the number is already in the answer, skip it
                    continue
```

```
47
48                  # Check if the current number can replace elements
                        in the answer to form a better subsequence
49                  while ((pointer and count[answer[pointer - 1]] and
                        better(num, pointer - 1, answer)) or
50                      (pointer >= 2 and count[answer[pointer - 1]]
                            and count[answer[pointer - 2]] and better
                            (num, pointer - 2, answer))):
51
52                      if pointer and count[answer[pointer - 1]] and
                            better(num, pointer - 1, answer):
53                          in_answer[answer[pointer - 1]] = False
54                          pointer -= 1
55                      else:
56                          in_answer[answer[pointer - 1]] = False
57                          pointer -= 1
58                          in_answer[answer[pointer - 1]] = False
59                          pointer -= 1
60
61                  # Add the current number to the answer
62                  answer[pointer] = num
63                  pointer += 1
64                  in_answer[num] = True
65
66          # Store the result for this test case
67          results.append(f"{pointer}\n{' '.join(map(str, answer[:
                pointer]))}")
68
69          # Reset the count and in_answer arrays for the next test
                case
70          for num in array:
71              count[num] = 0
72              in_answer[num] = False
73          pointer = 0
74
75      # Print all results
76      print("\n".join(results))
77
78  if __name__ == "__main__":
79      main()
```

Listing 1: Código solución