

# Longest Max Min Subsequence Round 967

Leonardo Javier Ramirez Calatayud C411

September 2024

# Índice

<b>1. Definición del problema</b>	<b>3</b>
<b>2. Solución con Backtracking</b>	<b>3</b>
2.1. Descripción de la Solución . . . . .	3
2.1.1. Algoritmo . . . . .	3
2.1.2. Código en Python . . . . .	4
2.2. Complejidad Temporal . . . . .	4
<b>3. Solución Mejorada</b>	<b>5</b>
3.1. Observación: . . . . .	5
3.1.1. Demostración: . . . . .	5
3.2. Greedy: . . . . .	5
3.3. Definiendo " <i>la mejor solución hasta el momento</i> " . . . . .	5
3.4. Agregando el elemento $a_{k+1}$ . . . . .	5
3.5. Observación: . . . . .	6
3.6. Construyendo la solución: . . . . .	6
<b>4. Código</b>	<b>7</b>
4.1. Comentarios y nombres de variables . . . . .	7
4.2. Complejidad temporal . . . . .	7
<b>5. Anexos</b>	<b>8</b>
5.1. Código fuente . . . . .	8

## 1. Definición del problema

Dado una secuencia de enteros  $a_1, a_2, \dots, a_n$ . 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 con Backtracking

### 2.1. Descripción de la Solución

El enfoque básico de la solución con backtracking consiste en generar todas las subsecuencias posibles del conjunto dado y luego elegir la más larga que cumpla con las restricciones establecidas, es decir, maximizar los números en las posiciones impares multiplicadas por  $-1$  y minimizar los números en las posiciones pares.

#### 2.1.1. Algoritmo

El algoritmo recorre todas las subsecuencias no vacías del arreglo original y utiliza una técnica de backtracking para probar diferentes combinaciones de elementos, evaluando si es posible agregar un nuevo elemento sin violar las restricciones impuestas. El objetivo es construir subsecuencias que sean lexicográficamente menores en orden y que maximicen los valores en las posiciones impares y minimicen los valores en las posiciones pares.

El algoritmo puede resumirse de la siguiente manera:

1. Generar todas las subsecuencias posibles del arreglo.

2. Para cada subsecuencia, calcular su transformación aplicando  $-1$  a los elementos en las posiciones impares.
3. Comparar las subsecuencias en términos de longitud, y si tienen la misma longitud, seleccionar la que sea menor lexicográficamente.

### 2.1.2. Código en Python

El siguiente es un ejemplo de la implementación básica del algoritmo de backtracking:

```

1 def generate_subsequences(arr, idx, current, result):
2     if idx == len(arr):
3         if current:
4             result.append(list(current))
5         return
6     # Exclude the current element
7     generate_subsequences(arr, idx + 1, current, result)
8     # Include the current element
9     current.append(arr[idx])
10    generate_subsequences(arr, idx + 1, current, result)
11    current.pop()
12
13 def transform_and_compare(subsequences):
14     best_subsequence = None
15     for subsequence in subsequences:
16         transformed = [-x if i % 2 == 0 else x for i, x in
17                        enumerate(subsequence)]
18         if best_subsequence is None or len(transformed) > len(
19             best_subsequence) or \
20            (len(transformed) == len(best_subsequence) and
21             transformed < best_subsequence):
22             best_subsequence = transformed
23     return best_subsequence
24
25 arr = [3, 2, 3, 1]
26 subsequences = []
27 generate_subsequences(arr, 0, [], subsequences)
28 best = transform_and_compare(subsequences)
29 print(best)

```

## 2.2. Complejidad Temporal

La complejidad temporal del algoritmo con backtracking es  $O(2^n)$ , donde  $n$  es el tamaño del arreglo de entrada. Esto se debe a que el número total de subsecuencias posibles de un arreglo de tamaño  $n$  es  $2^n$ . Cada vez que se genera una subsecuencia, se realiza una comparación que toma tiempo proporcional a la longitud de la subsecuencia. Este enfoque no es eficiente para grandes valores de  $n$ .

### 3. Solución Mejorada

#### 3.1. Observación:

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

##### 3.1.1. 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.

#### 3.2. Greedy:

Una vez conocida la longitud del arreglo final, nuestro siguiente paso es lograr que los elementos de las posiciones  $1, 3, \dots, k + 1$  sean los máximos posibles, mientras que los  $2, 4, \dots, 2k$  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*.

#### 3.3. Definiendo "la mejor solución hasta el momento"

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

#### 3.4. 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  $S_{k'}, S_{k'-1}, \dots, S_{k'-t}$ , 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  $\mathit{cnt} = 0$ , puesto que si  $a_{k+1}$  ocupa el lugar de  $S_{k'-t-1}$ , entonces se incumpliría 2.3.

### 3.5. 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  $S$ .

### 3.6. 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  $a = [a_1, a_2, \dots, a_n]$ . 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  $\text{cnt}(a_i)$ , 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.

## 4. Código

### 4.1. 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.

### 4.2. Complejidad temporal

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

## 5. Anexos

### 5.1. Código fuente

```
1 def better(value, position, answer):
2     # Check if the value is better for the given position
3     if position % 2 == 1:
4         return value < answer[position]
5     return value > answer[position]
6
7 def main():
8     import sys
9     input = sys.stdin.read
10    data = input().split()
11
12    index = 0
13    t = int(data[index]) # Number of test cases
14    index += 1
15
16    results = []
17
18    for _ in range(t):
19        n = int(data[index]) # Length of the array for this test
20        case
21        index += 1
22
23        array = list(map(int, data[index:index + n])) # Read the
24        array
25        index += n
26
27        count = [0] * (max(array) + 1) # Frequency count of each
28        number in the array
29        answer = [0] * n # Array to store the resulting subsequence
30        in_answer = [False] * (max(array) + 1) # Boolean array to
31        check if a number is in the answer
32        pointer = 0 # Pointer to the current position in the answer
33        array
34
35        # Count the frequency of each number in the array
36        for num in array:
37            count[num] += 1
38
39        # Process each number in the array
40        for num in array:
41            count[num] -= 1 # Decrease the count as we process the
42            number
43
44            if pointer == 0:
45                # If the answer array is empty, add the number
46                answer[pointer] = num
47                pointer += 1
48                in_answer[num] = True
49            else:
50                if in_answer[num]:
51                    # If the number is already in the answer, skip
52                    it
53                    continue
```



```

47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
# Check if the current number can replace elements
# in the answer to form a better subsequence
while ((pointer and count[answer[pointer - 1]] and
better(num, pointer - 1, answer)) or
(pointer >= 2 and count[answer[pointer - 1]]
and count[answer[pointer - 2]] and better
(num, pointer - 2, answer))):

    if pointer and count[answer[pointer - 1]] and
better(num, pointer - 1, answer):
        in_answer[answer[pointer - 1]] = False
        pointer -= 1
    else:
        in_answer[answer[pointer - 1]] = False
        pointer -= 1
        in_answer[answer[pointer - 1]] = False
        pointer -= 1

# Add the current number to the answer
answer[pointer] = num
pointer += 1
in_answer[num] = True

# Store the result for this test case
results.append(f"{pointer}\n{' '.join(map(str, answer[:
pointer]))}")

# Reset the count and in_answer arrays for the next test
case
for num in array:
    count[num] = 0
    in_answer[num] = False
pointer = 0

# Print all results
print("\n".join(results))

if __name__ == "__main__":
    main()

```

Listing 1: Código solución