F2. Interesting Problem (Hard Version)

Leonardo Javier Ramirez Calatayud C411

Septiembre 2024

Contents

1 Definiendo el Problema

Se tiene un arreglo de enteros a de longitud n. En una operación, se realiza el siguiente proceso en dos pasos:

1. Elige un índice i tal que $1 \leq i < |a|$ y $a_i = i$.
2. Elimina a_i y a_{i+1} del arreglo y concatena las partes restantes.

Encuentra el número máximo de veces que puedes realizar la operación anterior.

2 Solución con Backtracking

La solución básica al problema descrito puede abordarse utilizando un algoritmo de *backtracking*, que explora todas las combinaciones posibles para eliminar los elementos del arreglo cumpliendo las condiciones impuestas. A continuación, se presenta el enfoque básico utilizando *backtracking*.

2.1 Definición del algoritmo

El algoritmo consiste en explorar de manera recursiva todas las posibles formas de elegir un índice i tal que $1 \leq i < |a|$ y $a_i = i$. Una vez encontrado dicho índice, eliminamos a_i y a_{i+1} del arreglo, y repetimos el proceso sobre el arreglo resultante.

```
def backtrack(arr):
    if len(arr) <= 1:
        return 0
    max_operations = 0
    for i in range(1, len(arr)):
        if arr[i] == i:
            # Eliminar a_i y a_{i+1}
            new_arr = arr[:i] + arr[i+2:]
            # Contabilizar la operación
            max_operations = max(max_operations, 1 + backtrack(new_arr))
    return max_operations
```

El código anterior define la función `backtrack`, que recibe como parámetro el arreglo y devuelve el número máximo de operaciones que se pueden realizar.

2.2 Complejidad temporal

El algoritmo de *backtracking* tiene una complejidad temporal exponencial debido a que explora todas las combinaciones posibles de eliminaciones. En el peor de los casos, en cada paso del algoritmo tenemos dos opciones: eliminar o no eliminar un par de elementos. Esto genera un árbol de búsqueda con $O(2^n)$ ramas, donde n es la longitud del arreglo original.

Debido a que el algoritmo intenta todas las combinaciones posibles de eliminaciones, su complejidad es:

$$O(2^n)$$

donde n es el tamaño del arreglo. Esto es ineficiente para valores grandes de n, pero proporciona una solución correcta y comprensible para casos pequeños.

3 Solución Mejorada

El enfoque a seguir para resolver el problema planteado anteriormente es utilizando la técnica de Programación Dinámica.

3.1 Observación:

Para que el número a_i sea eliminado, tiene que cumplirse que a_i e i tengan la misma paridad y $a_i \geq i$.

3.1.1 Demostración:

Si $a_i < i$, entonces no existe forma de desplazar a_i a la izquierda para que caiga en su posición. Si a_i no tiene la misma paridad que i, entonces como se eliminan siempre a_k y a_{k+1}, lo cual es equivalente a llevar a_{k+1} a la posición de a_k, entonces la paridad de a_{k+1} se mantiene.

3.2 Observación:

Se necesitan realizar $\frac{i-a_i}{2}$ operaciones exactamente a la izquierda de a_i para luego eliminar a_i.

3.2.1 Demostración:

$i - a_i$ es el espacio entre i y a_i, y cada operación elimina 2 elementos. Por lo tanto, $\frac{i-a_i}{2}$ es la cantidad de operaciones necesarias para llevar a a_i a la posición en la que coincide con i.

3.3 Definiendo $dp[l, r]$

Ahora supongamos que existe un intervalo $[a_l, a_r]$ que puede ser completamente eliminado. Luego, existe una cantidad mínima de operaciones que se deben hacer a la izquierda de a_l para eliminar dicho intervalo. Para ello, almacenaremos en $dp[l, r]$ dicho valor si existe y si no hay forma de eliminarlo completamente. Para el caso donde se cumple $(l > r)$ vamos a decir que $dp[l, r] = 0$.

3.3.1 ¿Cómo calcular $dp[l, r]$?

Inicialmente, vamos a definir $dp[l, r] = +\infty$ para el resto de casos no vistos.

Nuestro trabajo es minimizar dp. Para ello, es necesario observar que si es posible eliminar el intervalo $a_l, a_{l+1}, \ldots, a_r$.

De ser posible entonces $\exists\ l < m \leq r$ tal que l se elimine con él.

Nótese que m debe cumplir que sea de paridad distinta a la de l, pues para eliminar a a_l con a_m, primero se debe eliminar $a_{l+1} \ldots a_{m-1}$, y como cada operación elimina 2 elementos, el intervalo $a_{l+1} \ldots a_{m-1}$ debe ser par; y las paridades extremas de un intervalo de tamaño par son distintas.[1].

Para que a_m sea candidato a ser eliminado junto a a_l, debe cumplirse que $dp[l+1, m-1] \leq \frac{l-a_l}{2}$, puesto que si es $> \frac{l-a_l}{2}$, sería imposible aplicar la operación a (a_l, a_m).

Luego, restaría saber la cantidad mínima de operaciones que hacen falta realizar a la izquierda de l para eliminar el intervalo (a_{m+1}, a_r), que se calcularía de la siguiente forma: $dp[m+1, r] - \frac{m-l+1}{2}$, donde $\frac{m-l+1}{2}$ son las operaciones realizadas entre a_m y a_l.

[1] En un intervalo de tamaño $2K$, existen K índices pares y K índices impares, entonces, si eliminamos los extremos y estos tienen la misma paridad, el intervalo resultante de tamaño $2K - 2$ tendría $K - 2$ elementos de una paridad y K de otra (contradicción).

Por lo tanto, minimizar $dp[l,r]$ sería como sigue:

$$dp[l,r] = \min\left(dp[l,r], \max\left(\frac{l-a_l}{2}, dp[m+1,r] - \frac{m-l+1}{2}\right)\right)$$

3.4 Definiendo $dp2$

Una vez calculados todos los valores de dp, podríamos introducir un $dp2[r]$ tal que sea el máximo número de operaciones que se pueden realizar en el prefijo $a_1 \ldots a_r$.

3.5 Calculando $dp2[n]$

Inicialmente, $dp2[r] = 0 \ \forall \ 1 \leq r \leq n$.
Notemos que si existe forma de eliminar el intervalo $a_l \ldots a_r$ y la cantidad de operaciones necesarias es $\leq dp2[l-1]$, entonces podríamos estar en presencia de una actualización de $dp2[r]$.

Luego, $dp2[r]$ se actualizaría de la siguiente forma:

$$dp2[r] = \max\left(dp2[r], dp2[l+1] + \frac{r-l+1}{2}\right)$$

Donde $\frac{r-l+1}{2}$ son las operaciones realizadas entre l y r.

Finalmente, $dp2[n]$ nos daría la cantidad de operaciones máximas que se pueden realizar en el arreglo completo, dando respuesta al ejercicio.

4 Código

4.1 Complejidad temporal

La complejidad de la solución propuesta es $O(n^3)$ debido al triple ciclo for anidado.

5 Anexos

5.1 Código fuente

```python
import sys
input = sys.stdin.read
from math import inf

def solve(n, array):
    # Inicializar la matriz dp con infinito
    dp = [[inf] * (n + 1) for _ in range(n + 1)]

    # Establecer la diagonal principal en 0
    for i in range(n + 1):
        dp[i][i] = 0

    # Rellenar la matriz dp
    for length in range(1, n + 1):
        for left in range(n - length + 1):
            if array[left] % 2 != (left + 1) % 2:
                continue
            if array[left] > left + 1:
                continue
            v = (left + 1 - array[left]) // 2

            right = left + length
            for mid in range(left + 1, right, 2):
                if dp[left + 1][mid] <= v:
                    new_val = max(v, dp[mid + 1][right] - (mid -
                        left + 1) // 2)
                    dp[left][right] = min(dp[left][right], new_val)

    # Inicializar el array dp2
    dp2 = [0] * (n + 1)
    for i in range(n):
        dp2[i + 1] = dp2[i]

        for j in range(i):
            if dp[j][i + 1] <= dp2[j]:
                dp2[i + 1] = max(dp2[i + 1], dp2[j] + (i - j + 1)
                    // 2)

    # Imprimir el resultado final
    print(dp2[n])
```

Listing 1: Código solución