The Last of Us

First off, note that it is always beneficial to concatenate segments after any operation that splits the array. For, if we suppose that concatenation would worsen the answer, a better answer can be found by reordering the operations.

Let's see an example with $n = 7$ and $k = 1$:

$$a = [3, 1, 1, 1, 1, 1, 3]$$

In this case, if we perform the operation on a_4, the array is split into two segments: $[3, 1]$ and $[1, 3]$. Then the answer would be either $a_2 + a_4 + a_7 = a_1 + a_4 + a_6 = 5$, if we concatenate them, or $a_2 + a_4 + a_6 = 3$ otherwise (which is better). But removing either a_2 or a_6 first would make the latter cost possible as well.

Proof outline

More precisely, let $1 \leq i < j < l \leq n$ denote indices of the chosen elements in the original array, such that their sum equals the best answer. For this answer to be possible, the following conditions must hold:

- $i + k < j$ and $j + k < l$, since operating on one element must not remove the others; and
- $n \leq 3 \cdot (2k + 1)$, because the whole array must be emptied by the three operations; and
- $\max(n - i + 1 + k, l + k) \leq 3 \cdot (2k + 1)$, because all elements (including non-chosen ones) to the right of a_i and to the left of a_l must be removed.

If we suppose that performing the operation on a_j and not concatenating the resulting segments would be optimal, this simply means that the remaining (chosen) elements would each belong to their own segment and similar conditions would hold for them individually.

However, the above conditions also imply that there is at least one ordering of the operations that achieves the required answer. Therefore, concatenation does not worsen the answer (although it may improve it).

Solution 1 - DFS

To address this problem, we'll begin with a depth-first search of possible combinations of selected indices. We'll need to keep track of four variables during the search, namely:

- i — current array index
- l — index of the first picked element
- r — index of the last picked element
- c — number of picked elements

The recursive function must return the minimum cost to empty the array from the current point, going forward. The transition is summarized below:

$$A(i, l, r, c) = \min\left(A(i + 1, l, r, c), a_i + A(i + k + 1, min(l, i), i, c + 1)\right)$$

meaning that we take the best cost between picking or skipping the current element, with base case:

$$A(n + 1, l, r, c) = \begin{cases} 0 & \text{if} \max\{n, n - l + 1 + k, r + k\} \leq c(2k + 1) \\ \infty & \text{otherwise} \end{cases}$$

meaning that the selection must satisfy the aforementioned conditions. Finally, the answer will be $A(0, \infty, 0, 0)$.

This works, but takes $O(2^n)$ time. We can improve it by using memoization, as described next.

Solution 2 - DFS + memoization

Note that the result for a particular combination of (i, l, c) will be the same regardless of r. In fact, it will be aggregated over all possible values of r. Hence, we can maintain a map of results based on the key (i, l, c). This improves the runtime to $O(n^3)$, which should be enough to pass the tests.

In practice, the use of an ordered map may have a high overhead, so an unordered map or other associative container with constant access should be preferred. The required extra memory is also $O(n^3)$.

Solution 3 - DP

To further refine the solution, we can transform the DFS into a dynamic programming formulation. The easiest way to do so is using a bottom-up formulation. Specifically, we need to iterate over all possible values of l, then apply DP over all possible c and r.

The transition is summarized below:

$$DP(r + k + 1, c + 1) = \min\left(DP(r + k, c + 1), a_r + DP(r, c)\right)$$

or equivalently:

$$DP(r, c) = \min\left(DP(r - 1, c), a_{r-k-1} + DP(r - k - 1, c - 1)\right)$$

The answer can be updated at each valid combination of (l, r, c). Alternatively, it can be updated at each valid combination of l and c, by taking the value of $DP(c(2k + 1), c)$.