<https://leetcode.com/problems/constrained-subsequence-sum/description/>

Given an integer array nums and an integer k, return the maximum sum of a non-empty subsequence of that array such that for every two consecutive integers in the subsequence, nums[i] and nums[j], where i < j, the condition j - i <= k is satisfied.

A subsequence of an array is obtained by deleting some number of elements (can be zero) from the array, leaving the remaining elements in their original order.

**Example 1:**

**Input:** nums = [10,2,-10,5,20], k = 2

**Output:** 37

**Explanation:** The subsequence is [10, 2, 5, 20].

**Example 2:**

**Input:** nums = [-1,-2,-3], k = 1

**Output:** -1

**Explanation:** The subsequence must be non-empty, so we choose the largest number.

**Example 3:**

**Input:** nums = [10,-2,-10,-5,20], k = 2

**Output:** 23

**Explanation:** The subsequence is [10, -2, -5, 20].

**Constraints:**

1 <= k <= nums.length <= 10^5

-10^4 <= nums[i] <= 10^4

**Attempt 1: 2024-10-26**

**Solution 1: Native DFS (120 min, TLE 8/40)**

class Solution {

    public int constrainedSubsetSum(int[] nums, int k) {

        int result = Integer.MIN\_VALUE;

        int n = nums.length;

        for(int j = 0; j < n; j++) {

            result = Math.max(result, helper(nums, k, j));

        }

        return result;

    }

    private int helper(int[] nums, int k, int j) {

        if(j < 0) {

            return 0;

        }

        // Let F(j) denote the maximum sum of the subsequence with

        // nums[j] as the last number.

        // F(j) = nums[j] + max(0, F(j-1), F(j-2), ..., F(j-i), ... F(j-k))

        // where j-i >= 0. Note that if all of the valid F(j-1),

        // F(j-2), ..., F(j-i), ... F(j-k) are negative, we will

        // not choose any of these values because choosing one of

        // them would lower the value of the required sum: hence we

        // have a 0 in the max term.

        int maxVal = 0;

        for(int i = 1; i <= k; i++) {

            maxVal = Math.max(maxVal, helper(nums, k, j - i));

        }

        return maxVal + nums[j];

    }

}

Time Complexity: O(k^n)

Space Complexity: O(n)

**Solution 2: DFS + Memoization (10 min, TLE 20/40)**

class Solution {

    public int constrainedSubsetSum(int[] nums, int k) {

        int result = Integer.MIN\_VALUE;

        int n = nums.length;

        Integer[] memo = new Integer[n];

        for(int j = 0; j < n; j++) {

            result = Math.max(result, helper(nums, k, j, memo));

        }

        return result;

    }

    private int helper(int[] nums, int k, int j, Integer[] memo) {

        if(j < 0) {

            return 0;

        }

        // Let F(j) denote the maximum sum of the subsequence with

        // nums[j] as the last number.

        // F(j) = nums[j] + max(0, F(j-1), F(j-2), ..., F(j-i), ... F(j-k))

        // where j-i >= 0. Note that if all of the valid F(j-1),

        // F(j-2), ..., F(j-i), ... F(j-k) are negative, we will

        // not choose any of these values because choosing one of

        // them would lower the value of the required sum: hence we

        // have a 0 in the max term.

        if(memo[j] != null) {

            return memo[j];

        }

        int maxVal = 0;

        for(int i = 1; i <= k; i++) {

            maxVal = Math.max(maxVal, helper(nums, k, j - i, memo));

        }

        return memo[j] = maxVal + nums[j];

    }

}

Time Complexity: O(k\*n)

Space Complexity: O(n)

**Solution 3: DP (120 min, TLE 20/40)**

class Solution {

    public int constrainedSubsetSum(int[] nums, int k) {

        int n = nums.length;

        // dp[j] represents the maximum sum of the subsequence

        // with nums[j] as the last element

        int[] dp = new int[n];

        dp[0] = nums[0];

        int result = dp[0];

        // 顶i -> 底0, 外层循环 j = [0,n), 但是0的状态是初始态,

        // 直接被dp[0]初始化代替了, 所以范围变成 j = [1,n)

        for(int j = 1; j < n; j++) {

            // 内层循环结构不变，继承 DFS 的 helper() 方法中的逻辑

            int maxVal = 0;

            for(int i = 1; i <= k; i++) {

                if(j - i >= 0) {

                    maxVal = Math.max(maxVal, dp[j - i]);

                }

            }

            dp[j] = nums[j] + maxVal;

            result = Math.max(result, dp[j]);

        }

        return result;

    }

}

Time Complexity: O(k\*n)

Space Complexity: O(n)

**Solution 4: Monotonic Decreasing Deque (120 min)**

class Solution {

    public int constrainedSubsetSum(int[] nums, int k) {

        // Monotonic Decreasing Deque

        Deque<Integer> deque = new ArrayDeque<>();

        int n = nums.length;

        int[] dp = new int[n];

        for(int i = 0; i < n; i++) {

            // If i minus the front of queue is greater than k,

            // remove from the front of queue.

            if(!deque.isEmpty() && i - deque.peek() > k) {

                deque.poll();

            }

            // Set dp[i] to dp[queue.peek()] + nums[i], since its

            // a Monotonic Decreasing Deque, the first position stored

            // the index represents maximum subsequence sum we can get

            // before reaching current index 'i', the previous maximum

            // + nums[i] guarantee the result is maximum for index 'i'.

            // If queue is empty, use 0 instead of dp[queue.peek()].

            dp[i] = (!deque.isEmpty() ? dp[deque.peek()] : 0) + nums[i];

            while(!deque.isEmpty() && dp[deque.peekLast()] < dp[i]) {

                deque.pollLast();

            }

            // We will only push positive values of dp[i] to queue, only

            // need to push index, because if push negative values, it

            // only produce smaller result if add later element, if negative

            // value happen, just not push and effect equal to push 0,

            // its better than produce smaller result.

            if(dp[i] > 0) {

                deque.offer(i);

            }

        }

        int result = Integer.MIN\_VALUE;

        for(int num : dp) {

            result = Math.max(result, num);

        }

        return result;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

**[Python] Thought process: Recursion-Memoization-Tabulation-Monoqueue**

<https://leetcode.com/problems/constrained-subsequence-sum/solutions/736798/python-thought-process-recursion-memoization-tabulation-monoqueue/>

In this post, I'll describe my thought process behind solving this question. I will start with a naive recursive solution and improve it to the optimal O(n) solution using DP + monoqueue.

**1. Recursion**

Let's say that nums[j] is the last number in the required subsequence. Since we need to maximize the sum of the numbers in the subsequence, the previous number in the subsequence could be max(nums[j-1], nums[j-2], ..., nums[j-i], ..., nums[j-k]) where j-i >= 0. However, if all nums[j-1], nums[j-2], ..., nums[j-i], ..., nums[j-k] are negative, there would be no previous number, as it would lower the required sum.

Let F(j) denote the maximum sum of the subsequence with nums[j] as the last number. Following the idea above, F(j) = nums[j] + max(0, F(j-1), F(j-2), ..., F(j-i), ... F(j-k)) where j-i >= 0. Note that if all of the valid F(j-1), F(j-2), ..., F(j-i), ... F(j-k) are negative, we will not choose any of these values because choosing one of them would lower the value of the required sum: hence we have a 0 in the max term. Since the solution to this problem is built off of solutions to the subproblems, we formuate a recursive solution.

For each j in the range [0, len(nums)-1], inclusive, we will call F(j), i.e, we consider every j as a potential last index of the subsequence. The required answer will be the max of all such F(j).

Note that this solution can also be formulated in a forward direction instead of going backwards. E.g. F(i) = nums[i] + max(0, F(i+1), F(i+2), ... F(i+j), ..., F(i+k) where i+j < len(nums) and F(i) denotes the maximum sum of the suquence with nums[i] as the first element. The reason why I chose to go backwards is because it is easier to convert it to an iterative approach as explained later.

class Solution:

def constrainedSubsetSum(self, nums: List[int], k: int) -> int:

def helper(j):

if j < 0:

return 0

max\_val = 0

for i in range(1, k+1):

max\_val = max(max\_val, helper(j-i))

return nums[j]+max\_val

ans = float("-inf")

for i in range(len(nums)):

ans = max(ans, helper(i))

return ans

**Time complexity: O(k^n) (TLE)**

**Space complexity: O(n)**

The recurrence relation is,

F(n) = F(n-1) + F(n-2) + ... + F(n-k)

.

.

.

F(0) = 1

If you draw a recursion tree, it will be a k-ary tree with depth n, where n is the length of nums. The time complexity is the number of nodes in this tree, O(k^n). The space complexity is the depth of this tree (recursion stack), O(n).

**2. Memoization**

The above solution has multiple calls to F(j) with the same j. To avoid exponential growth, we can store these intermediate results and retrieve them from memory next time the function is called with the same j. This is called Memoization (not Memorization), and it is a way of doing dynamic programming. A really convenient way to do memoization in Python is using the decorator lru\_cache.

from functools import lru\_cache

class Solution:

def constrainedSubsetSum(self, nums: List[int], k: int) -> int:

@lru\_cache(maxsize=None) # add this line to the previous solution

def helper(j):

if j < 0:

return 0

max\_val = 0

for i in range(1, k+1):

max\_val = max(max\_val, helper(j-i))

return nums[j]+max\_val

ans = float("-inf")

for i in range(len(nums)):

ans = max(ans, helper(i))

return ans

**Time complexity: O(nk) (TLE)**

**Space complexity: O(n)**

By doing memoization, F(j) is processed at most once, where j is in the range [0, n-1], inclusive. To determine each F(j), we recursively call F(j-1), F(j-2), ... , F(j-i), ... F(j-k). Hence, for each F(j), we do k amount of work. Think of it as using a for loop with k iterations. Since there are n such calls to F(j) as described previously, we do n \* k amount of work overall. Hence, the time complexity is O(nk). The space complexity has two components: recursion stack and the memory used for memoization. Together, they require 2 \* n amount of space. Hence the space complexity remains linear, O(n).

**3. Tabulation**

We can use an iterative approach for the above solution. Tabulation is another (iterative) way of doing dynamic programming. We create a list called dp with size n and initialize it with zeros. Here, dp[j] represents the maximum sum of the subsequence with nums[j] as the last element. Using the same idea described in the above solutions, dp[j] = nums[j] + max(0, dp[j-1], dp[j-2], ..., dp[j-i], ..., dp[j-k]), where j-i >= 0. To start with, we set dp[0] = nums[0], this is our initial condition. For each j in the range [1, n-1], inclusive, we determine dp[j]. The final answer is the max of all such dp[j].

class Solution:

def constrainedSubsetSum(self, nums: List[int], k: int) -> int:

dp = [0]\*len(nums)

dp[0] = nums[0]

ans = dp[0]

for j in range(1, len(nums)):

# use max(0, j-k) in the range to keep the index within bounds

for i in reversed(range(max(0, j-k), j)):

dp[j] = max(dp[j], dp[i])

dp[j] += nums[j]

ans = max(ans, dp[j])

return ans

**Time complexity: O(nk) (TLE)**

**Space complexity: O(n)**

The time and space complexity is the same as the memoization solution. Perhaps, the time complexity of the memoization solution can be better understood after relating it to this solution. The space complexity is O(n) due to the dp list. We can modify the code a little and improve it to O(k), since we care only about the previous k elements of dp at a time. Or we can avoid using a dp list and update the nums list directly to make it O(1).

**4. Tabulation with Monoqueue**

To understand this approach, I highly recommend you to solve 239. Sliding Window Maximum first. In the previous approach, we determine dp[j] by finding the maximum of the previous k elements of dp. The bottleneck of the previous approach is iterating k times for each j to find this maximum, resulting in O(nk) time complexity. This can be avoided by using a monoqueue: a monotonically decreasing (or increasing in some cases) queue. Essentially, we use the sliding window maximum approach on the dp list. The gist of this approach is figuring out the following:

Let's say we want to determine dp[j]. Consider the window dp[j-k] to dp[j-1]. There are two indexes in this window, p and q, where p < q.

nums[p] < nums[q]. In this case, nums[p] is no longer a candidate for the maximum since we have found nums[q], which is greater than nums[p], within the window. So we can discard nums[p].

nums[p] >= nums[q]. In this case, nums[p] is a candidate for the maximum. But we also need to keep track of nums[q] because it might become a candidate later when p goes out of the window. Note that index q comes after index p, therefore, as the window slides forward (while determining dp[j+1], dp[j+2], ..., dp[n-1]), we will lose p first.

From the above points, it is evident that we need to keep track of monotonically decreasing numbers.

While keeping track of such numbers, if the largest number (the first number in this monotonically decreasing list) goes out of the window for a certain dp[j], we need to remove it from the beginning of the list. Hence we use a monoqueue (and not a monostack).

That being said, we maintain a monoqueue while determining each dp[j]. For each j in the range [1, n-1], inclusive, we check whether the first element of the queue is within the window. If not, then we pop it from the beginning. If yes, then this element is, indeed, the maximum within the window due to the monoqueue property. Thus dp[j] = nums[j] + max(0, queue[0]). Now, we add this dp[j] to the monoqueue while maintaining its property by removing all the elements less than dp[j] from the end of the queue. As before, the final answer is the max of all dp[j].

Here is the code:

from collections import deque

class Solution:

def constrainedSubsetSum(self, nums: List[int], k: int) -> int:

dp = [0]\*len(nums)

dp[0] = nums[0]

ans = dp[0]

# queue holds the index

queue = deque([0])

for j in range(1, len(nums)):

if queue and queue[0] < j-k:

queue.popleft()

dp[j] = nums[j] + max(0, dp[queue[0]])

while queue and dp[queue[-1]] < dp[j]:

queue.pop()

queue.append(j)

ans = max(ans, dp[j])

return ans

**Time complexity: O(n)**

**Space complexity: O(n)**

This monoqueue takes care of the unnecessary k iterations in the previous approach and reduces the time complexity to O(n), since each element is added and popped out of the queue at most once. The space complexity is O(k) due to queue and O(n) due to the dp list. It can be reduced to O(k) by directly updating the nums list instead of creating a dp list.

**Refer to**

<https://leetcode.com/problems/constrained-subsequence-sum/editorial/>

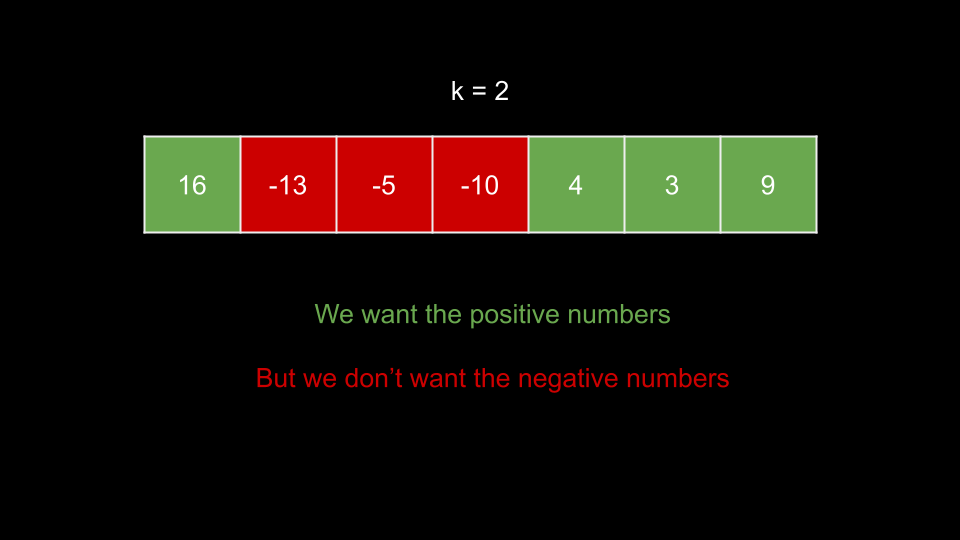
**Approach 1: Heap/Priority Queue**

**Intuition**

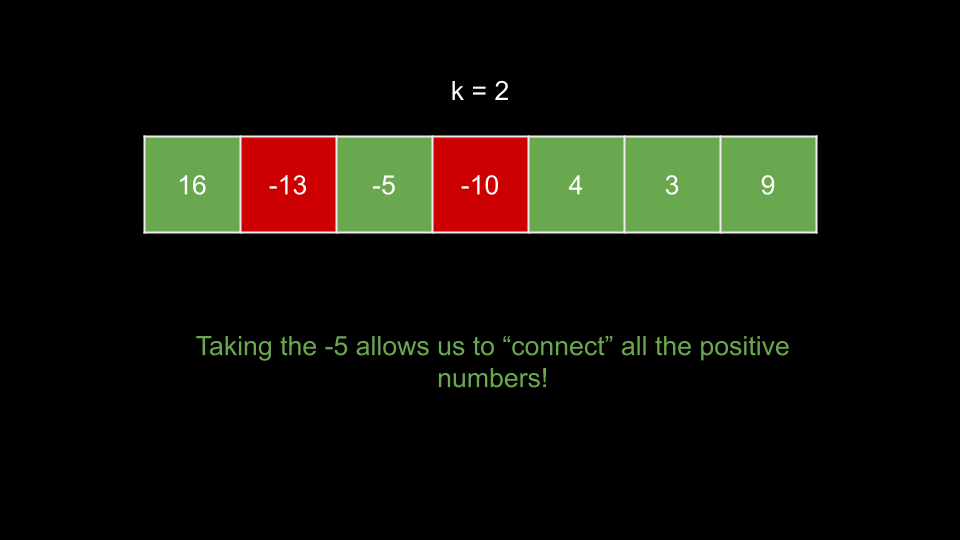
Before we start developing a strategy, we must carefully understand what the problem is asking for.

We need to maximize the sum of a subsequence. We can take as many integers as we want, but the primary constraint is that we **cannot** have a gap of k or more in our subsequence.

You may immediately notice that in an array of positive integers, we should always take the entire array. The tricky part comes in when we have negative integers. Of course, we would prefer to avoid negative integers since they will decrease our sum. However, it may be worth taking a negative integer as a sort of "bridge". Take a look at the following example:



In this example, we have a group of negative numbers separating a 16 and a group of positive numbers that sum to 16. We would like to take all the positive numbers while avoiding the negative numbers, but we aren't allowed to as that would result in a gap of three numbers. As k = 2, the biggest gap we can have is one number. The optimal solution here is to take the -5.



As you can see, the -5 acts as a bridge for the positive numbers. The question now is, how do we know when it is worth it to take negative numbers? In this case, taking the -5 allowed us to take the first element of 16. This results in a net gain of 11. Anytime we have a positive net gain, we should consider taking this element because it can contribute to a positive sum and potentially increase the sum of subsequent subsequences.

We will iterate over the input from left to right. At each index i, we will consider the maximum possible sum of a subsequence that **includes and ends at nums[i]**. Let's call this value curr. How do we calculate curr for a given index i? We want the maximum possible sum of a subsequence that ends within the last k indices. We will then add nums[i] to this sum.

We could solve this using dynamic programming - let dp[i] represent the maximum possible sum of a subsequence that includes and ends at nums[i]. We can calculate dp[i] by taking the maximum dp[j] for all j in the range [i - k, i - 1] (the last k indices), then adding nums[i] to it.

However, we would be iterating up to k times to calculate each state. As k can be large, this approach is too slow. We need a faster way to find the maximum dp[j] for all indices j in the range [i - k, i - 1].

Because we are only concerned with the maximum sum, we could use a max heap. The max heap would store dp[j] for all j in the last k indices. We can easily calculate curr by simply checking the top of this heap.

We need to make sure we don't use elements of the heap that are more than k away from the current index. Before we calculate curr, we pop from the top of the heap if it is outside our range. This means each entry in the heap will also need its associated index, so we can tell when an element is out of range.

Note that if the top of the heap is negative, it is better to not take it. This is a process very similar to Kadane's Algorithm, which solves the Maximum Subarray problem. When the top of the heap is negative, it indicates that selecting this subsequence would result in a sum less than 0. Every element in the array to the left of the current index should be abandoned - any "bridge" would not be worth taking. It's better to discard these subsequences altogether and reset the sum to 0.

**Algorithm**

Initialize a max heap with (nums[0], 0). Also initialize the answer ans = nums[0].

Iterate i over the indices of nums, starting from i = 1:

While i minus the index (second element) at the top of heap is greater than k, pop from heap.

Set curr to the value (first element) at the top of heap, plus nums[i]. Note that if the value at the top of heap is negative, we should take 0 instead.

Update ans with curr if it is larger.

Push (curr, i) to heap.

Return ans.

**Implementation**

Implementation note: Python's heapq module only implements min heaps, so we will make the values in the heap negative to simulate a max heap.

class Solution {

public int constrainedSubsetSum(int[] nums, int k) {

PriorityQueue<int[]> heap = new PriorityQueue<>((a, b) -> {

return b[0] - a[0];

});

heap.add(new int[] {nums[0], 0});

int ans = nums[0];

for (int i = 1; i < nums.length; i++) {

while (i - heap.peek()[1] > k) {

heap.remove();

}

int curr = Math.max(0, heap.peek()[0]) + nums[i];

ans = Math.max(ans, curr);

heap.add(new int[] {curr, i});

}

return ans;

}

}

**Complexity Analysis**

Given n as the length of nums,

**Time complexity: O(n⋅logn)**

We iterate over each index of nums once. At each iteration, we have a while loop and some heap operations. The while loop runs in O(1) amortized - because an element can only be popped from the heap once, the while loop cannot run more than O(n) times in total across all iterations.

The heap operations depend on the size of the heap. In an array of only positive integers, we will never pop from the heap. Thus, the size of the heap will grow to O(n) and the heap operations will cost O(logn).

**Space complexity: O(n)**

As mentioned above, heap could grow to a size of n.

**Approach 2: TreeMap-Like Data Structure**

**Intuition**

As we saw in the previous approach, the crux of the dynamic programming idea was finding the maximum value of dp in the last k indices. We accomplished this in O(logn) time with a heap, but we could achieve O(logk) with a tree map data structure (like a red-black tree). Because k <= n, this is a slight improvement in terms of big O.

Let's actually use the dp array that we spoke of in the previous approach this time. We will have a data structure window that holds all values of dp in the last k indices. We can easily calculate dp[i] as nums[i] plus the maximum value in window. Then, we can add dp[i] to window.

To maintain window, once we reach index k, we need to start removing dp[i - k] from window at each iteration.

In Java, we will use TreeMap. Each key will be a value in dp which we will map to its frequency. To remove dp[i - k] from the window, we will decrement its frequency, and if its frequency becomes 0, we will delete the key.

In C++, we will use std::map, which functions similarly to Java's TreeMap.

In Python, we will use sortedcontainers.SortedList, which is more like a list than a map, but still provides us with the efficient operations we require.

For all implementations, we will initialize window with a key of 0 to make the code cleaner, otherwise we would need to handle the first index differently (check if window is empty before accessing the maximum key).

The answer to the problem will be the max value in dp in the end.

**Algorithm**

Initialize window with 0: 0.

Initialize an array dp with the same length as nums.

Iterate i over the indices of nums:

Set dp[i] to nums[i] plus the maximum key in window.

Increment the frequency of dp[i] in window.

If i >= k:

Decrement the frequency of dp[i - k] in window. If the frequency becomes 0, delete it from window.

Return the max value in dp.

**Implementation**

class Solution {

public int constrainedSubsetSum(int[] nums, int k) {

TreeMap<Integer, Integer> window = new TreeMap();

window.put(0, 0);

int dp[] = new int[nums.length];

for (int i = 0; i < nums.length; i++) {

dp[i] = nums[i] + window.lastKey();

window.put(dp[i], window.getOrDefault(dp[i], 0) + 1);

if (i >= k) {

window.put(dp[i - k], window.get(dp[i - k]) - 1);

if (window.get(dp[i - k]) == 0) {

window.remove(dp[i - k]);

}

}

}

int ans = Integer.MIN\_VALUE;

for (int num : dp) {

ans = Math.max(ans, num);

}

return ans;

}

}

**Complexity Analysis**

Given n as the length of nums,

**Time complexity: O(n⋅logk)**

We iterate over each index of nums once. At each iteration, we have some operations with window. The cost of these operations is a function of the size of window. As window will never exceed a size of k, these operations cost O(logk).

**Space complexity: O(n)**

window will not exceed a size of k, but dp requires O(n) space.

**Approach 3: Monotonic Deque**

**Intuition**

This approach is very similar to the solution to Sliding Window Maximum. We recommend you try this problem as well if you haven't already.

Is it possible to find the maximum value of dp in the last k indices in O(1)? Yes, by using a monotonic queue!

A monotonic data structure is one where the elements are always sorted. If we have a monotonic **decreasing** data structure, then the elements are always sorted descending. Thus, if we can maintain a monotonic data structure that holds values of dp for the last k indices, then the first element in this data structure will be the value we are interested in.

To maintain this data structure, we need to make sure that whenever we push a new element, it will be the smallest value. Before we push an element dp[i], we check the last element. If it is less than dp[i], we must pop it, otherwise, the monotonic property would be broken. Since there may be multiple elements less than dp[i], we need to use a while loop to "clean" the data structure before pushing dp[i].

Only once there are no elements in the data structure less than dp[i] will we push dp[i]. **Additionally, we will only push positive values of dp[i] to queue.**

The reason we want to remove elements that are less than dp[i] is because dp[i] comes after those elements. Thus, those elements will be out of range before dp[i], and because dp[i] is greater than them, there is no chance those elements will ever be the maximum value in the last k indices anymore.

Before we check the max value, we must make sure it is not out of range. If it is, we will remove this invalid max value. As you can see, we need to remove elements from both the front and the back. Thus, we will use a deque (double-ended queue) as our data structure.

To detect if the max value is out of range, we must store the indices in the queue.

To check if the max value is out of range, we check if i - queue.front() > k.

To obtain the max value of the queue, we check dp[queue.front()]

To obtain the value at the end of the queue, we check dp[queue.back()]

Note that we could also store pairs (dp[i], i) on the queue.

**Algorithm**

Initialize a deque queue. Also initialize an array dp with the same length as nums.

Iterate i over the indices of nums:

If i minus the front of queue is greater than k, remove from the front of queue.

Set dp[i] to dp[queue.front()] + nums[i]. If queue is empty, use 0 instead of dp[queue.front()].

While dp[queue.back()] is less than dp[i], pop from the back of queue.

If dp[i] > 0, push i to the back of queue.

Return the max element in dp.

**Implementation**

class Solution {

public int constrainedSubsetSum(int[] nums, int k) {

Deque<Integer> queue = new ArrayDeque<>();

int dp[] = new int[nums.length];

for (int i = 0; i < nums.length; i++) {

if (!queue.isEmpty() && i - queue.peek() > k) {

queue.poll();

}

dp[i] = (!queue.isEmpty() ? dp[queue.peek()] : 0) + nums[i];

while (!queue.isEmpty() && dp[queue.peekLast()] < dp[i]) {

queue.pollLast();

}

if (dp[i] > 0) {

queue.offer(i);

}

}

int ans = Integer.MIN\_VALUE;

for (int num : dp) {

ans = Math.max(ans, num);

}

return ans;

}

}

**Complexity Analysis**

Given n as the length of nums,

**Time complexity: O(n)**

We iterate over each index once. At each iteration, we have a while loop. This while loop runs in O(1) amortized. Each element in nums can only be pushed and popped from queue at most once. Thus, this while loop will not run more than n times across all n iterations. Everything else in each iteration runs in O(1). Thus, each iteration costs O(1) amortized.

**Space complexity: O(n)**

dp requires O(n) space.

Since we always remove out-of-range elements from queue, so it contains at most k elements and requires O(k) space.

**Refer to**

[L53.Maximum Subarray (Ref.L821)](note://872ABD256AD34054B5D1F2E9992E2CAA)

[L239.P2.1.Sliding Window Maximum](note://6738BA39EFAD452FA70CC8B64512437D)

[L739.Daily Temperatures](note://6FB3A11E2B7746DCA411301F941DD948)