<https://leetcode.com/problems/jump-game-vi/>

You are given a **0-indexed** integer array nums and an integer k.

You are initially standing at index 0. In one move, you can jump at most k steps forward without going outside the boundaries of the array. That is, you can jump from index i to any index in the range [i + 1, min(n - 1, i + k)] **inclusive**.

You want to reach the last index of the array (index n - 1). Your **score** is the **sum** of all nums[j] for each index j you visited in the array.

Return *the* ***maximum score*** *you can get*.

**Example 1:**

Input: nums = [1,-1,-2,4,-7,3], k = 2

Output: 7

Explanation: You can choose your jumps forming the subsequence [1,-1,4,3] (underlined above). The sum is 7.

**Example 2:**

Input: nums = [10,-5,-2,4,0,3], k = 3

Output: 17

Explanation: You can choose your jumps forming the subsequence [10,4,3] (underlined above). The sum is 17.

**Example 3:**

Input: nums = [1,-5,-20,4,-1,3,-6,-3], k = 2

Output: 0

**Constraints:**

* 1 <= nums.length, k <= 105
* -104 <= nums[i] <= 104

**Attempt 1: 2023-01-23**

**Solution 1:  Native DFS (30 min, TLE)**

class Solution {

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

return helper(nums, k, 0);

}

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

if(index == nums.length - 1) {

return nums[nums.length - 1];

}

int score = Integer.MIN\_VALUE;

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

if(i < nums.length) {

score = Math.max(score, nums[index] + helper(nums, k, i));

}

}

return score;

}

}

Time Complexity : O(k^N) ~ exponential,

where N is the size of array and k is max jump length. We have k choices at each index and we are trying out each choice every time and recursing for remaining indices. So overall time complexity becomes k\*k\*k\*...N times = O(k^N)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/jump-game-vi/solutions/1260737/optimizations-from-brute-force-to-dynamic-programming-w-explanation/>

❌ ***Solution - I (Brute-Force)***

Let's start with doing just what the questions asks us. For any index *i* (such that *i < n-1*), we can jump at most k-steps forward and we need to reach n-1 index such that we maximize the sum of scores of intermediate jumps.

For all the indices, we will just try each jump of lengths from 1...k and return the one which maximizes the score at the end. Since we cannot go outside array bounds, we will return nums[n-1] (last index we can reach) when we reach i >= n-1.

int maxResult(vector<int>& nums, int k, int i = 0) {

if(i >= size(nums) - 1) return nums.back();

int score = INT\_MIN;

for(int j = 1; j <= k; j++)

score = max(score, nums[i] + maxResult(nums, k, i + j));

return score;

}

***Time Complexity :*** **O(k^N)**, where *N* is the size of array and *k* is max jump length. We have k choices at each index and we are trying out each choice every time and recursing for remaining indices. So overall time complexity becomes *k\*k\*k\*...N times* = *O(k^N)*

***Space Complexity :*** **O(N)**, required by the recursive stack.

<https://leetcode.com/problems/jump-game-vi/solutions/1261213/java-journey-from-brute-force-to-most-optimized-dp-sliding-window-algorithm/>

In this approach we try out the above thought and try to process all the nodes with their k neighbors and get the max score.

private int maxResult(int[] nums, int i, int k) {

if(i >= nums.length - 1) return nums[nums.length - 1];

int max = Integer.MIN\_VALUE;

for(int j = i + 1; j <= Math.min(i + k, nums.length); j++) // Mininum of i + k, nums.length to prevent unnecessary calls for out of bounds.

max = Math.max(max, maxResult( nums, j , k));

return max;

}

*Time Complexity* - O(k^n) - As at each node we have k next nodes to be processed. So *k,k,k.....n times* gives k^n.

**Solution 2:  Top Down DP (Memoization) (10 min, still TLE)**

class Solution {

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

Integer[] memo = new Integer[nums.length];

return helper(nums, k, 0, memo);

}

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

if(index == nums.length - 1) {

return nums[nums.length - 1];

}

if(memo[index] != null) {

return memo[index];

}

int score = Integer.MIN\_VALUE;

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

if(i < nums.length) {

score = Math.max(score, nums[index] + helper(nums, k, i, memo));

}

}

memo[index] = score;

return score;

}

}

Time Complexity : O(k\*N)

Space Complexity : O(N)

**Refer to**

<https://leetcode.com/problems/jump-game-vi/solutions/1260737/optimizations-from-brute-force-to-dynamic-programming-w-explanation/>

❌ ***Solution - II (Dynamic Programming (Memoization)- Top-Down Approach)***

In the above brute force approach, **we are re-calculating for the same index multiple times**. But we can observe that the maximum score which we can obtain by starting from an index *i* will always remain the same.

So, instead of doing unnecessary repeated calculations, we can use **dynamic programming** to store the calculated results and directly return it whenever required in the future calculations. Here, we will maintain an dp array, where ***dp[i]* will denote the maximum score that we can obtain starting from ith index**. We will initialize *dp[n-1] = nums[n-1]* (since that's the only score we can obtain starting at n-1th index) and rest of dp[i] to *-infinity* denoting that they have not been computed yet.

Now, for each *i*, we have the choice to make jumps of lengths *1,2...k* and we will store the maximum score in *dp[i]* after trying out jump of each length. If *dp[i]* is already computed once, we will just return it without doing any re-computations.

int maxResult(vector<int>& nums, int k) {

vector<int> dp(size(nums), INT\_MIN);

dp.back() = nums.back(); // dp[n-1]=nums[n-1]

return solve(nums, dp, k, 0);

}

// recursive solver which finds max score to reach n-1 starting from ith index

int solve(vector<int>& nums, vector<int>& dp, int k, int i) {

if(dp[i] != INT\_MIN) return dp[i]; // already calculated result for index i

for(int j = 1; j <= k; j++) // try jumps of all length and choose the one which maximises the score

if(i + j < size(nums))

dp[i] = max(dp[i], nums[i] + solve(nums, dp, k, i + j));

return dp[i];

}

***Time Complexity :*** **O(k\*N)** For each index, we are trying out k jumps and storing the results to avoid future re-computations. Overall, the time complexity required is *k+k+k...N times* = O(k\*N)

***Space Complexity :*** **O(N)**

<https://leetcode.com/problems/jump-game-vi/solutions/1261213/java-journey-from-brute-force-to-most-optimized-dp-sliding-window-algorithm/>

As it might be evident that we are recalculating many of the nodes again and again while their solution would remain the same as from a given node the maximum score to reach end would be same. e.g. 1,2,3,4,5,6,7 k - 3, so when we are at Node 1 - we process Node 2,3,4 and get the maximum value. When we move to node Node 2 - we process Node 3,4,5 So if we see we are again processing the Node 3, 4 which is not required at all.

So, this calls for introducing a memory to store the already processed nodes and just provide a lookup of O(1).

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

int[] mem = new int[nums.length]; // Lookup table to score max score for given node

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

mem[i] = Integer.MIN\_VALUE;

return maxResult(mem, nums, 0, k);

}

private int maxResult(int[] mem, int[] nums, int i, int k) {

if(i >= nums.length - 1) return nums[nums.length - 1];

if(mem[i] != Integer.MIN\_VALUE) return mem[i];

int max = Integer.MIN\_VALUE;

for(int j = i + 1; j <= Math.min(i + k, nums.length); j++)

max = Math.max(max, maxResult(mem, nums, j , k));

mem[i] = nums[i] + max;

return mem[i];

}

*Time Complexity* - O(k^n) - As at each node we have k next nodes to be processed. So *k,k,k.....n times* gives k^n. Even though the lookup prevents further re-processing of already processed nodes(which can make it little faster).

**Solution 3:  Bottom Up DP (120 min, still TLE)**

**Style 1: Traverse backward**

class Solution {

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

int n = nums.length;

// dp[i] denotes maximum achievable score to reach dp[i] starting from (len - 1)th index

int[] dp = new int[n];

Arrays.fill(dp, Integer.MIN\_VALUE);

dp[n - 1] = nums[n - 1];

for(int i = n - 2; i >= 0; i--) {

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

if(i + j < n) {

dp[i] = Math.max(dp[i], dp[i + j] + nums[i]);

}

}

}

return dp[0];

}

}

Time Complexity : O(k\*N)

Space Complexity : O(N)

**Refer to**

<https://leetcode.com/problems/jump-game-vi/solutions/2257293/java-recursive-memoized-tabulated-optimized-dp/>

**TABULATED APROACH**

Throws TLE.

class Solution {

private int MIN = Integer.MIN\_VALUE / 2;

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

int n = nums.length;

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

dp[n - 1] = nums[n - 1];

for(int index = n - 2; index >= 0; index--){

int maximumScore = MIN;

for(int i = index + 1; i <= Math.min(n - 1, index + k); i++)

maximumScore = Math.max(maximumScore, dp[i] + nums[index]);

dp[index] = maximumScore;

}

return dp[0];

}

}

**Style 2: Traverse forward (Relation between i, j can either be 'i - j' or 'i + j' as L45. Jump Game II)**

**To understand relate how dp[i] denotes maximum achievable score to reach dp[i] starting from 0th index, the best way is debugging to monitor how dp value updated, and during debugging, for correct answer till last position, dp = {1, 0, -1, 4, -3, 7}, and 7 is the answer**

// i - j style

class Solution {

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

int n = nums.length;

// dp[i] denotes maximum achievable score to reach dp[i] starting from 0th index

int[] dp = new int[n];

Arrays.fill(dp, Integer.MIN\_VALUE);

dp[0] = nums[0];

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

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

if(i - j >= 0) {

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

}

}

}

return dp[n - 1];

}

}

=========================================================================================

// i + j style, don't forget "+ nums[i]" update to "+ nums[i + j]"

class Solution {

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

int n = nums.length;

// dp[i] denotes maximum achievable score to reach dp[i] starting from 0th index

int[] dp = new int[n];

Arrays.fill(dp, Integer.MIN\_VALUE);

dp[0] = nums[0];

for(int i = 0; i < n; i++) {

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

if(i + j < n) {

dp[i + j] = Math.max(dp[i + j], dp[i] + nums[i + j]);

}

}

}

return dp[n - 1];

}

}

Time Complexity : O(k\*N)

Space Complexity : O(N)

**Refer to**

<https://leetcode.com/problems/jump-game-vi/solutions/1260737/optimizations-from-brute-force-to-dynamic-programming-w-explanation/>

❌ ***Solution - III (Dynamic Programming (Tabulation) - Bottom-Up Approach)***

The above solution still leads to **TLE** with the given constraints. Let's see the iterative (bottom-up) version of the above code before moving to the optimized solution.

Here, *dp[i]* denotes maximum achievable score to reach *dp[i]* starting from 0th index. We start with *dp[0] = nums[0]* and make our way till the end. Finally dp[n-1] will give the maximum score we can obtain starting from *i=0*.

int maxResult(vector<int>& nums, int k) {

vector<int> dp(size(nums), INT\_MIN);

dp[0] = nums[0];

for(int i = 1; i < size(nums); i++)

for(int j = 1; j <= k && i - j >= 0; j++) // try all jumps length

dp[i] = max(dp[i], dp[i - j] + nums[i]); // choose the jump from previous index which maximises score

return dp.back();

}

***Time Complexity :*** **O(k\*N)**

***Space Complexity :*** **O(N)**

**Solution 4:  Optimized Bottom Up DP + Sliding Window + Priority Queue (120 min)**

class Solution {

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

int n = nums.length;

// dp[i] denotes maximum achievable score to reach dp[i] starting from 0th index

int[] dp = new int[n];

Arrays.fill(dp, Integer.MIN\_VALUE);

dp[0] = nums[0];

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

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

if(i - j >= 0) {

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

}

}

}

return dp[n - 1];

}

}

===========================================================

Evolution to below:

class Solution {

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

int n = nums.length;

// int[] -> {index, value}

PriorityQueue<int[]> maxPQ = new PriorityQueue<int[]>(k, (a, b) -> b[1] - a[1]);

maxPQ.offer(new int[]{0, nums[0]});

int max = nums[0];

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

// The optimization here is that, in the previous solution,

// we are always taking that dp[i-j], j going from 1 to k

// and i-j should be greater than 0, which is the maximum,

// so instead of always looping from 1 to k, we are storing

// the values of dp[i]'s in the max heap and taking the top

// element satisfying the condition that it is not exceeding

// i by more than k

while(i - maxPQ.peek()[0] > k) {

maxPQ.poll(); // We just compare the top node and see if it is outside of window or not by index check

}

// Since always pick "dp[i-j] + nums[i]" rather than "dp[i]"

// as larger value for position 'i', no need comparison as

// "dp[i] = Math.max(dp[i], dp[i - j] + nums[i])", just

// assign "dp[i-j] + nums[i]" to 'max', and dp[i-j] replace

// by "maxPQ.peek()[1]"

max = nums[i] + maxPQ.peek()[1];

maxPQ.offer(new int[]{i, max});

}

return max;

}

}

Time Complexity : O(N\*log(k))

Space Complexity : O(N)

**Refer to**

<https://leetcode.com/problems/jump-game-vi/solutions/1260737/optimizations-from-brute-force-to-dynamic-programming-w-explanation/>

✔️ ***Solution - IV (Optimized Dynamic Programming)***

In the above dynamic programming approach, we can observe that in the equation *dp[i] = max(dp[i], dp[i - j] + nums[i])*, **we are always choosing the dp[i-j] which has the maximum score**.

**We can observe why always choosing dp[i - j] by printing the logs below:**

public class Solution {

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

int n = nums.length;

int[] dp = new int[n];

Arrays.fill(dp, Integer.MIN\_VALUE);

dp[0] = nums[0];

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

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

if(i - j >= 0) {

// Add printing to find: In the above dynamic programming approach, we can

// observe that in the equation dp[i] = max(dp[i], dp[i - j] + nums[i]),

// we are always choosing the dp[i-j] which has the maximum score.

if(dp[i] < dp[i - j] + nums[i]) {

System.out.println("i=" + i + ", j=" + j + ", pick \"dp[i - j] + nums[i]\"=" + (dp[i - j] + nums[i]));

}

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

}

}

}

return dp[n - 1];

}

public static void main(String[] args) {

Solution s = new Solution();

int result = s.maxResult(new int[]{1,-1,-2,4,-7,3}, 2);

System.out.println(result);

}

}

==================================================================

For all index 'i', no matter j=1 or j=2, the equation dp[i] = max(dp[i], dp[i - j] + nums[i]), we are always choosing the dp[i-j] which has the maximum score.

i=1, j=1, pick "dp[i - j] + nums[i]"=0

i=2, j=1, pick "dp[i - j] + nums[i]"=-2

i=2, j=2, pick "dp[i - j] + nums[i]"=-1

i=3, j=1, pick "dp[i - j] + nums[i]"=3

i=3, j=2, pick "dp[i - j] + nums[i]"=4

i=4, j=1, pick "dp[i - j] + nums[i]"=-3

i=5, j=1, pick "dp[i - j] + nums[i]"=0

i=5, j=2, pick "dp[i - j] + nums[i]"=7

**So, instead of iterating k times, we can just store the maximum amongst the previous *dp[i-1], dp[i-2], ...dp[i-k]* and use it directly in the equation.** For this, we can make use of a **multiset** to maintain the results of max score for previous k indices in a sorted order. **That is, we are storing dp[i-1], dp[i-2],..., dp[i-k] in a sorted** **order**in the multiset.

Whenever we reach *i > k*, the dp[i-k - 1] will be useless to us, since there's no way we can reach current index i from i-k - 1th index. So, we will just remove it from our set. Finally, the above equation - *dp[i] = max(dp[i], dp[i - j] + nums[i])*, can now be solved in **O(logk)** instead of O(k).

int maxResult(vector<int>& nums, int k) {

vector<int> dp(size(nums), INT\_MIN);

multiset<int> s ({ dp[0] = nums[0] }); // set dp[0] = nums[0] and insert it into set

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

if(i > k) s.erase(s.find(dp[i - k - 1])); // erase elements from which we cant jump to current index

s.insert(dp[i] = \*rbegin(s) + nums[i]); // choose element with max score and jump from that to the current index

}

return dp.back();

}

***Time Complexity :*** **O(N\*log(k))**

***Space Complexity :*** **O(N)**

**Note: Java has no multiset equally data structure, but we can directly spin up to PriorityQueue**

**Refer to**

<https://leetcode.com/problems/jump-game-vi/solutions/2259706/c-recursion-memoization-tabulation-tabulation-priority-queue/>

**The optimization here is that, in the previous solution, we are always taking that dp[i-j], j going from 1 to k and i-j should be greater than 0, which is the maximum, so instead of always looping from 1 to k, we are storing the values of dp[i]'s in the max heap and taking the top element satisfying the condition that it is not exceeding i by more than k.**

class Solution {

public:

int maxResult(vector<int>& nums, int k) {

vector<int>dp(nums.size(),INT\_MIN);

dp[0] = nums[0];

priority\_queue<pair<int,int>>pq;

pq.push({dp[0],0});

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

while(pq.size() && i-pq.top().second >k){

pq.pop();

}

auto top = pq.top();

// Since proof earlier, we don't have to compare dp[i] and nums[i] + dp[top.second] again,

// as we always choose to use nums[i] + dp[top.second]

dp[i] = max(dp[i], nums[i] + dp[top.second]);

pq.push({dp[i],i});

}

return dp[nums.size()-1];

}

};

**Refer to**

<https://leetcode.com/problems/jump-game-vi/solutions/1261213/java-journey-from-brute-force-to-most-optimized-dp-sliding-window-algorithm/>

If we closely look at the problem we can understand that we need to get the maximum score within a given window ( k steps) preceding current node(inclusive) and it would give us the maximum score for the current node. At a given node we only need the nearest k nodes data to process. For this we use a priority queue(PQ) to fetch maximum score in O(1) with overall cost of O(nlog(n)) for interval sorting of PQ. Also, once we have moved forward we need to remove the element which is beyond the k step window( even if it is the maximum scoring).

To clear the last statement, we might have some values in Priority Queue which are outside of K step window but when we process a node we make sure we are using the score of the node within the acceptable window.

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

PriorityQueue<Pair<Integer, Integer>> pq = new PriorityQueue<>(k,

Comparator.comparing(o -> -o.getValue())) {{

offer(new Pair<>(0, nums[0]));

}};

int max = nums[0], ans;

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

while (pq.peek().getKey() < i - k) { // We just compare the top node and see if it is outside of window or not.

pq.poll();

}

max = nums[i] + pq.peek().getValue();

pq.offer(new Pair<>(i, max));

}

return max;

}

*Time Complexity* - O(nlog(n)) - Although it seem that we have multiple while loops but if we think about we are only processing the nodes at most twice once for processing that and second time while removing from the window. So, overall complexity depends on the sorting in PQ which is O(nlog(n)) due to internal heap.

**Solution 5:  Optimized Bottom Up DP + Sliding Window + Deque / Monotonic Queue (360 min, the Sliding Window + Deque / Monotonic Queue solution go with similar route as how L239/P2.1.Sliding Window Maximum works)**

class Solution {

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

int n = nums.length;

Deque<int[]> deque = new LinkedList<int[]>();

// int[] -> {index, value}

deque.offer(new int[]{0, nums[0]});

int max = nums[0];

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

// Remove index(represent corresponding number) out of range k

// from deque's left end(front end)

if(!deque.isEmpty() && i - deque.peekFirst()[0] > k) {

deque.pollFirst();

}

// Based on monotonic deque theory, the left most element is the

// largest element

max = nums[i] + deque.peekFirst()[1];

// Add new index(represent corresponding number) onto deque's

// right end(rear end), but since it requires O(n) time complexity,

// we could not implement additional sort algorithem or use auto

// sort data structure like Priority Queue which is O(nlogn), finally

// comes to Montonic Deque which guarantee O(n)

// Before add a new index onto deque's rear end, we have to compare new

// element's value(max) against all previous rear end elements'

// values(deque.peekLast()[1]), if previous rear end elements'

// values less than new element value, we have to looply remove them

// till find a larger or equal element to maintain a strictly decreasing

// order of elements' values on monotonic deque from left to right(front

// to rear)

// In short, the elements(indexes represented) stored on deque must

// monotonically decrease from left to right, e.g 1st > 2nd > 3rd...

// and we can easily find the maximum element value at the left end(front

// end) of deque

while(!deque.isEmpty() && deque.peekLast()[1] < max) {

deque.pollLast();

}

deque.offerLast(new int[]{i, max});

}

return max;

}

}

Time Complexity : O(N)

Space Complexity : O(N)

**Note: Java has no multiset equally data structure, but we can directly spin up to Deque**

**Refer to**

<https://leetcode.com/problems/jump-game-vi/solutions/1261213/java-journey-from-brute-force-to-most-optimized-dp-sliding-window-algorithm/>

**DP + Sliding Window -- Deque (Accepted) 70ms ✅**

Although the above PQ based solution is accepted, we can improve the overall complexity if we can handle getting maximum value for the window without the need to sort them. This can be achieved by maintaining the maximum score element at the front of the queue always.

Once the front element is out of window we can simply remove the element. Also, if we see once we get a score greater than the previous elements in the window we don't need th earlier elements as the current would be always selected for future elements due to it's score greater than earlier ones and also it is latest processed so would remain in the window for the last.

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

Deque<Pair<Integer, Integer>> deque = new LinkedList<>() {{

offer(new Pair<>(0, nums[0]));

}};

int max = nums[0];

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

while (!deque.isEmpty() && deque.peekFirst().getKey() < i - k) {

deque.pollFirst();

}

max = nums[i] + (deque.isEmpty() ? 0 : deque.peekFirst().getValue());

while (!deque.isEmpty() && deque.peekLast().getValue() <= max) {

deque.pollLast();

}

deque.offerLast(new Pair<>(i, max));

}

return max;

}

*Time Complexity* - O(n) - At initial glance it might seem that we have multiple loops that might lead to complexity greater than O(n); however, if we take a close look, we can see that each node would be processed maximum of 2 times. Once while the processing of itself and the second time either while removing out of window elements of the onces which are less than the current maximum score.

**Refer to**

<https://leetcode.com/problems/jump-game-vi/solutions/1260737/optimizations-from-brute-force-to-dynamic-programming-w-explanation/>

✔️ ***Solution - V (Further Optimized DP)***

We can maintain a simple double-side queue in a sorted order to reduce the max previous score lookup from O(logk) down to O(1). Here we will store the indices instead of dp[i] in the queue.

Just as in above approach, we will pop i-k-1th index from queue since it will be useless to us. Along with that, we will also pop those indices which will never have any chance of being chosen in the future. So for eg., if the score for current index - dp[i] is greater than some indices stored in the queue, it will always be optimal to choose dp[i] instead of those other indices. So, we will just pop those indices from queue since they won't ever be used.

int maxResult(vector<int>& nums, int k) {

vector<int> dp(size(nums));

dp[0] = nums[0];

deque<int> q{ 0 };

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

if(q.front() < i - k) q.pop\_front(); // can't reach current index from index stored in q

dp[i] = nums[i] + dp[q.front()]; // update max score for current index

while(!q.empty() && dp[q.back()] <= dp[i]) // pop indices which won't be ever chosen in the future

q.pop\_back();

q.push\_back(i); // insert current index

}

return dp.back();

}

***Time Complexity :*** **O(N)**

***Space Complexity :*** **O(N)** required for dp and q. You can optimize it down to O(K) if you use *nums* itself as the dp array (if you are allowed to modify the given input)