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

You are given a **0-indexed** array of integers nums of length n. You are initially positioned at nums[0].

Each element nums[i] represents the maximum length of a forward jump from index i. In other words, if you are at nums[i], you can jump to any nums[i + j] where:

* 0 <= j <= nums[i] and
* i + j < n

Return *the minimum number of jumps to reach* nums[n - 1]. The test cases are generated such that you can reach nums[n - 1].

**Example 1:**

Input: nums = [2,3,1,1,4]

Output: 2

Explanation: The minimum number of jumps to reach the last index is 2. Jump 1 step from index 0 to 1, then 3 steps to the last index.

**Example 2:**

Input: nums = [2,3,0,1,4]

Output: 2

**Constraints:**

* 1 <= nums.length <= 104
* 0 <= nums[i] <= 1000

**Attempt 1: 2023-01-22**

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

class Solution {

public int jump(int[] nums) {

return helper(nums, 0);

}

// Return remain steps required to get last position of array

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

// Able to reach, no more steps required

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

return 0;

}

// Not able to reach, infinite steps remained

// Note: actually not required, since minimum steps maximum value

// set as 10001, even return because of not approach last position

// but already stuck on certain index won't increase the maximum value

if(nums[index] == 0) {

return 10001;

}

int min\_steps = 10001;

// I can make jumps ranging from index + 1, till index + nums[index],

// and hence will run a loop to cover all those possbile jumps

for(int i = index + 1; i <= index + nums[index]; i++) {

if(i < nums.length) {

min\_steps = Math.min(min\_steps, 1 + helper(nums, i));

}

}

return min\_steps;

}

}

Time Complexity: O(N!)

At each index i we have N-i choices and we recursively explore each of them till end. So we require O(N\*(N-1)\*(N-2)...1) = O(N!).

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/jump-game-ii/solutions/1192401/easy-solutions-w-explanation-optimizations-from-brute-force-to-dp-to-greedy-bfs/>

❌ ***Solution - I (Brute Force) [Rejected]***

We start at index 0 and are required to reach index n - 1 (where n = nums.size()). We can't always do the maximum jump at each index. This can be easily verified by looking at the example test cases.

So, at each position, we can use a jump size of anywhere in the range *[1, nums[pos]]*. The final answer will be the minimum jumps required. We can recursively solve this problem as -

* If we reach index *n-1* return 0, signifying that we need 0 more jumps.
* Else recurse for each jump size possible from the current index and return the answer in which we require the minimum number of jumps

int jump(vector<int>& nums, int pos = 0) {

if(pos >= size(nums) - 1) return 0;

int minJumps = 10001; // initialising to max possible jumps + 1

for(int j = 1; j <= nums[pos]; j++) // explore all possible jump sizes from current position

minJumps = min(minJumps, 1 + jump(nums, pos + j));

return minJumps;

}

***Time Complexity :*** **O(N!)**. At each index i we have N-i choices and we recursively explore each of them till end. So we require O(N\*(N-1)\*(N-2)...1) = O(N!).

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

<https://leetcode.com/problems/jump-game-ii/solutions/2591598/recursion-memoization-tabulation-o-n-with-constant-space/>

int jump(vector<int>& nums) {

int n = nums.size();

return f(0, n, nums);

}

int f(int ind, int n, vector<int>& nums) {

if(ind == n-1) return 0;

int jump = nums[ind], mini = 1e9;

for(int i = ind + 1; i <= min(ind+jump, n-1); i++) {

int val = 1 + f(i, n, nums);

mini = min(mini, val);

}

return mini;

}

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

class Solution {

public int jump(int[] nums) {

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

return helper(nums, 0, memo);

}

// Return remain steps required to get last position of array

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

// Able to reach, no more steps required

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

return 0;

}

// Not able to reach, infinite steps remained

// Note: actually not required, since minimum steps maximum value

// set as 10001, even return because of not approach last position

// but already stuck on certain index won't increase the maximum value

if(nums[index] == 0) {

return 10001;

}

if(memo[index] != null) {

return memo[index];

}

int min\_steps = 10001;

for(int i = index + 1; i <= index + nums[index]; i++) {

if(i < nums.length) {

min\_steps = Math.min(min\_steps, 1 + helper(nums, i, memo));

}

}

memo[index] = min\_steps;

return min\_steps;

}

}

Time Complexity : O(N^2)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/jump-game-ii/solutions/1192401/easy-solutions-w-explanation-optimizations-from-brute-force-to-dp-to-greedy-bfs/>

✔️ ***Solution - II (Recursive Dynamic Programming - Memoization) [Accepted]***

We can see that for a given position, we are repeatedly calculating the same answer over and over again. The jumps required to reach for a given index on the path remains fixed and can be stored in *dp* array to avoid re-calculations.

The solution is similar as the brute force with just the change that we are storing the solutions for each *pos* and returning it if it is already calculated.

int jump(vector<int>& nums) {

vector<int> dp(size(nums), 10001); // initialise all to max possible jumps + 1 denoting dp[i] hasn't been computed yet

return solve(nums, dp, 0);

}

// recursive solver to find min jumps to reach end

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

if(pos >= size(nums) - 1) return 0; // when we reach end, return 0 denoting no more jumps required

if(dp[pos] != 10001) return dp[pos]; // number of jumps from pos to end is already calculated, so just return it

// explore all possible jump sizes from current position. Store & return min jumps required

for(int j = 1; j <= nums[pos]; j++)

dp[pos] = min(dp[pos], 1 + solve(nums, dp, pos + j));

return dp[pos];

}

***Time Complexity :*** **O(N^2)**

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

**Solution 3:  Bottom Up DP (10 min)**

**Style 1: Traverse backward**

class Solution {

public int jump(int[] nums) {

// dp[i] means minimum steps can reach last position from index 'i'

int n = nums.length;

int[] dp = new int[n];

Arrays.fill(dp, 10001);

// Since no step required for last position, minimum step as 0

dp[n - 1] = 0;

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

for(int j = 1; j <= nums[i]; j++) {

// 'i + j' should not over index boundary

if(i + j < n) {

// Scan backwards and that's why compare index 'i' status with

// index 'i + j' status, for index 'i + j' it requires index 'i'

// to move 1 more step to reach, just need to choose the less

// one between index 'i' status and index 'i + j' status

// plus 1 looply

dp[i] = Math.min(dp[i], 1 + dp[i + j]);

}

}

}

return dp[0];

}

}

Time Complexity : O(N^2)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/jump-game-ii/solutions/1192401/easy-solutions-w-explanation-optimizations-from-brute-force-to-dp-to-greedy-bfs/>

✔️ ***Solution - III (Iterative Dynamic Programming - Tabulation) [Accepted]***

We can solve this iteratively as well. For this, we start from the last index. We need 0 jumps from nums[n-1] to reach the end. We store this as *dp[n - 1] = 0* and then iteratively solve this for each previous index till the 0th index. Here *dp[i]* denotes minimum jumps required from current index to reach till the end.

1. For each index, we explore all the possible jump sizes available with us.
2. The minimum jumps required to reach the end from the current index would be - **min(dp[jumpLen])**, where *1 <= jumpLen <= nums[currentPostion]*

int jump(vector<int>& nums) {

int n = size(nums);

vector<int> dp(n, 10001);

dp[n - 1] = 0; // start from last index. No jumps required to reach end if we are already here

// same as above. For each index, explore all jump sizes and use the one requiring minimum jumps to reach end

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

for(int jumpLen = 1; jumpLen <= nums[i]; jumpLen++)

dp[i] = min(dp[i], 1 + dp[min(n - 1, i + jumpLen)]); // min(n-1, i + jumpLen) for bounds handling

return dp[0];

}

***Time Complexity :*** **O(N^2)**

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

**Style 2: Traverse forward**

class Solution {

public int jump(int[] nums) {

// dp[i] means minimum steps can reach position 'i'

// Note: definition changed than tarverse backwards

int n = nums.length;

int[] dp = new int[n];

Arrays.fill(dp, 10001);

// Since no step required to reach first position, minimum step as 0

dp[0] = 0;

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

for(int j = 1; j <= nums[i]; j++) {

// 'i + j' should not over index boundary

if(i + j < n) {

// Scan forwards and that's why compare index 'i + j' status with

// index 'i' status, for index 'i + j' it requires index 'i'

// to move 1 more step to reach, just need to choose the less

// one between index 'i + j' status and index 'i' status

// plus 1 looply

dp[i + j] = Math.min(dp[i + j], 1 + dp[i]);

}

}

}

return dp[n - 1];

}

}

Time Complexity : O(N^2)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/jump-game-ii/solutions/1197063/intuitive-explanation-w-solution-analysis-java-beginner-dp/>

This problem is one of the simplest and best problems for beginners to understand one of the most difficult yet rewarding algorithm - **Dynamic Programming**. The aim of the post is not to just throw away with a working solution, rather provide a new perspective for beginners and potential learners on how to deconstruct a problem statement to understand the crux of it and reconstruct back into a working algorithm. Let's quickly get into the process of understanding the problem and coming up with some intuitions and solutions.

**INTUITION**

* **It is mentioned in the problem that we start from the initial position and we can always reach the end position. We need to reach them with minimum possible steps.** This means that from the start position, there can be one or many possible ways of jumps (traversing through multiple sets of indices) to reach the end point. Hence it will be initially necessary to have a data structure that stores the optimum number of jumps to reach any particular index. Since the constraints states that the input array length will be less than 1000 *(1 <= nums.length <= 1000)*, we create an output array of length 1000 *(ideally length of input array)* and maintain the minimum jumps required to reach each position.
* **All indices are unreachable initially.** Since we start from the first position, it is impossible to reach all the other indices without making the first move. So all of them can be initialized with some maximum value *(1e9 or Integer.MAX\_VALUE)*. However, since we are standing in the first position, it is reachable by default without making any move. Hence we initialize the value of the first index in the output array as 0.

**CODE & EXPLANTION**

class Solution {

public int jump(int[] nums) {

int i,n,j;

n=nums.length;

int [] pre = new int[n];

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

{

// Initialize all the index position with some max value to denote it is unreachable

pre[i] = Integer.MAX\_VALUE;

}

// Make the first position value as 0 as it is the default place where we start

pre[0] = 0;

// Main condition check goes here.

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

{

for(j=i;j<=i+nums[i] && j<n;j++)

{

pre[j] = Math.min(pre[j], pre[i]+1);

}

}

return pre[n-1];

}

}

Let's try to understand the loop where the main condition check happens. Each value in the nums array specifies the jump that is possible to take from the current position. So if the value in the ith index is nums[i], then it is understandable that from **(ith) position to (i+nums[i])th position** *(i, i+1, i+2, ...., i+nums[i])*, it can be reached in a single jump. However as already mentioned, since a particular position can be reached from various other position, we need to store the minimum possible jump value to reach the position.*pre[j] = Math.min(pre[j], pre[i]+1)*.

By filling it this way, we can naturally fill the way the last element is reached also and return the pre[n-1] value directly.

There goes around a popular saying that *'When one learns things right, he learns them only once'*. There could be many people (including me :P) who would have done hundreds of problems and still finds it difficult to crack the next one because of a gap in the understanding and ability to convert that into code. Lets make use of this post to address some basic intuitions and approaches along with constructing a solution for this problem.

**Solution 4:  Greedy (60 min, similar to BFS)**

class Solution {

public int jump(int[] nums) {

int steps = 0;

int curLevelEnd = 0;

int nextLevelEnd = 0;

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

if(i > curLevelEnd) {

steps++;

curLevelEnd = nextLevelEnd;

}

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

}

return steps;

}

}

Time Complexity : O(N)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/jump-game-ii/solutions/18089/evolve-from-brute-force-to-optimal/>

BFS O(n) Time, O(1) Space. The problem asks for shortest path and it is naturally to think of BFS. Nodes are array elements. For an element, neighbors are the elements that are within its max jump length. Given nodes in current level, we generate all nodes in the next level. Nodes are traversed level by level.

public int jump(int[] nums) {

int steps = 0, curLevelEnd = 0, nxtLevelEnd = 0;

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

if(i>curLevelEnd) {

steps++;

curLevelEnd = nxtLevelEnd;

}

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

}

return steps;

}

<https://leetcode.com/problems/jump-game-ii/solutions/1192401/easy-solutions-w-explanation-optimizations-from-brute-force-to-dp-to-greedy-bfs/>

✔️ ***Solution IV (Greedy BFS)***

We can iterate over all indices maintaining the furthest reachable position from current index - *maxReachable* and currently furthest reached position - *lastJumpedPos*. Every time we will try to update *lastJumpedPos* to furthest possible reachable index - maxReachable.

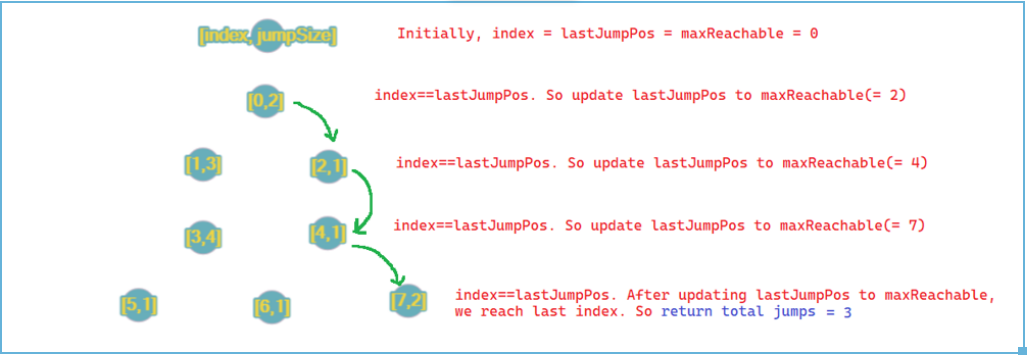
Updating the lastJumpedPos separately from *maxReachable* allows us to maintain track of minimum *jumps* required. Each time lastJumpedPos is updated, ***jumps*** will also be updated and store the minimum jumps required to reach lastJumpedPos (On the contrary, updating jumps with maxReachable won't give the optimal (minimum possible) value of jumps required).

We will just return it as soon as *lastJumpedPos* reaches(or exceeds) last index.

We can try to understand the steps in code below as analogous to those in BFS as -

* **maxReachable = max(maxReachable, i + nums[i])** : Updating the range of next level. Similar to *queue.push(node)* step of BFS but here we are only greedily storing the max reachable index on next level.
* **i == lastJumpedPos** : When it becomes true, current level iteration has been completed.
* **lastJumpedPos = maxReachable** : Set range till which we need to iterate the next level
* **jumps++** : Move on to the next level.
* **return jumps** : The final answer will be number of levels in BFS traversal.

For eg. Take the example : nums = [2,3,1,4,1,1,1,2]. This approach proceeds as illustrated in image below -



int jump(vector<int>& nums) {

int n = size(nums), i = 0, maxReachable = 0, lastJumpedPos = 0, jumps = 0;

while(lastJumpedPos < n - 1) { // loop till last jump hasn't taken us till the end

maxReachable = max(maxReachable, i + nums[i]); // furthest index reachable on the next level from current level

if(i == lastJumpedPos) { // current level has been iterated & maxReachable position on next level has been finalised

lastJumpedPos = maxReachable; // so just move to that maxReachable position

jumps++; // and increment the level

// NOTE: jump^ only gets updated after we iterate all possible jumps from previous level

// This ensures jumps will only store minimum jump required to reach lastJumpedPos

}

i++;

}

return jumps;

}

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

***Space Complexity :*** **O(1)**