Previous Next

45. Jump Game II

Feb. 27, 2020 | 10.4K views

Average Rating: 3.69 (29 votes)

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

Example:

```
Input: [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.
```

Note:

You can assume that you can always reach the last index.

Solution

Approach 1: Greedy, Moving Forward

Optimal solution

There are several ways to solve this problem:

- Backtracking, $\mathcal{O}(2^N)$ time.
- Dynamic programming, $\mathcal{O}(N)$ time, $\mathcal{O}(N)$ space.
- Greedy, $\mathcal{O}(N)$ time and $\mathcal{O}(1)$ space.

In this article we will consider in details the optimal greedy approach.

Greedy algorithms

Greedy problems usually look like "Find minimum number of *something* to do *something*" or "Find maximum number of *something* to fit in *some conditions*".

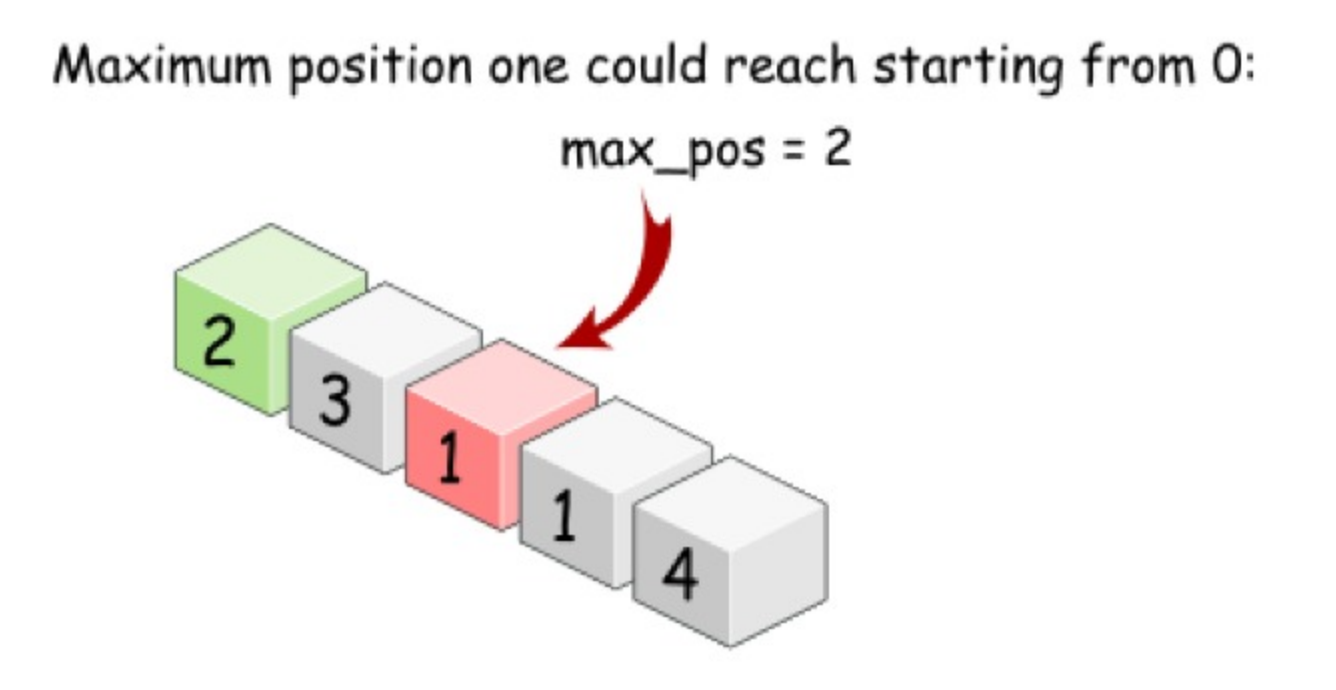
> The idea of greedy algorithm is to pick *locally* optimal move at each step, that will lead to *globally* optimal solution.

Jump Game

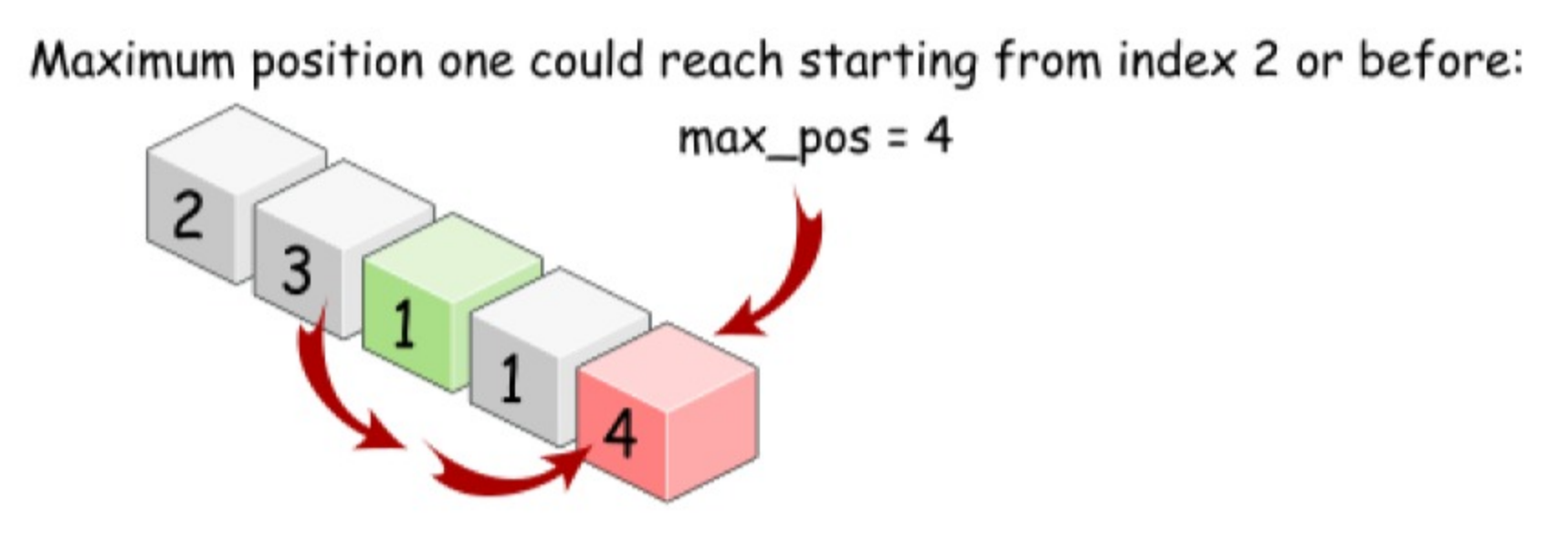
Let us start from the predecessor problem - Jump Game, which is to determine if it's possible to reach the last index.

> The most straightforward solution is to compute at each point the *maximum position* `max_pos` that one could reach starting from the current index `i` or *before*.

The minimum value for `max_pos` is `i + nums[i]` = current position + jump length. For example, for index 0, `max_pos = nums[0]`.

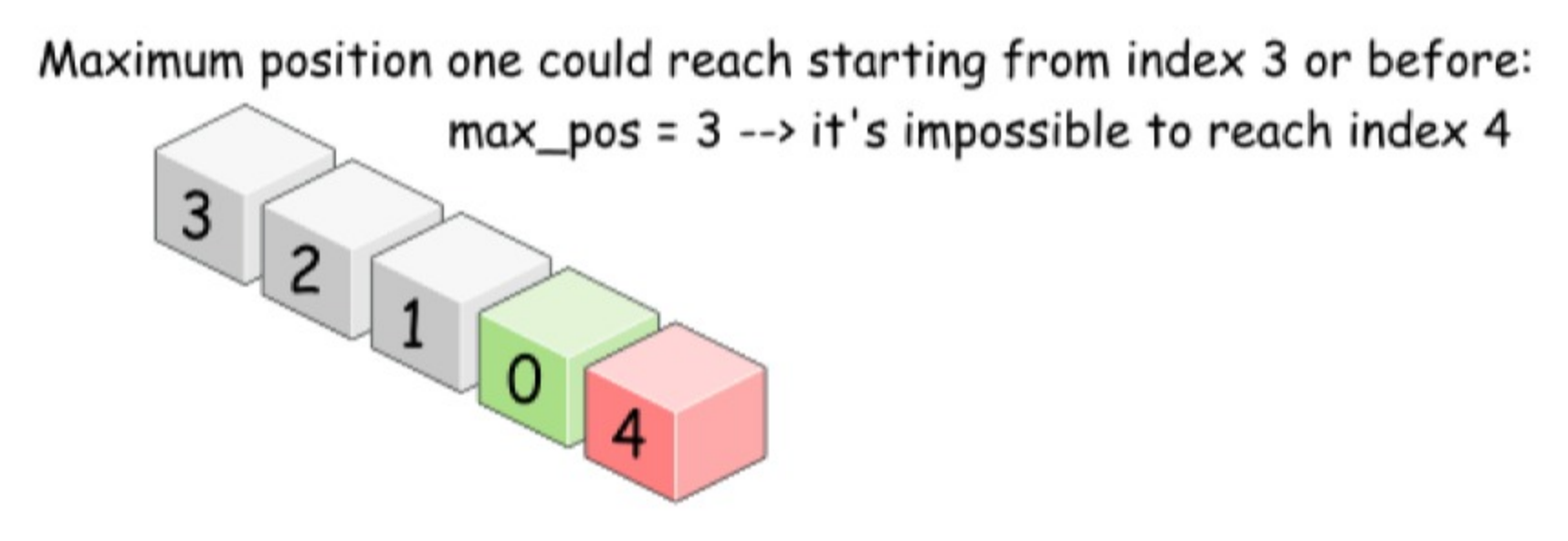


Maximum position could be greater than `i + nums[i]` if one of the previous cells allows longer jump. For example, cell 2 offers jump length equal to 1, *i.e.* starting from index 2, one could reach cell 3. Although, starting *before*, at cell 1, one could reach cell 4, and hence the maximum position at index 2 is 4.



Now the solution is simple: to compare for each index `max_pos[i - 1]` and `i`.

> One couldn't reach index `i` if the maximum position that one could reach starting from the previous cells is less than `i`. Unreachable index `i <= n -1` means that the last index is unreachable as well.



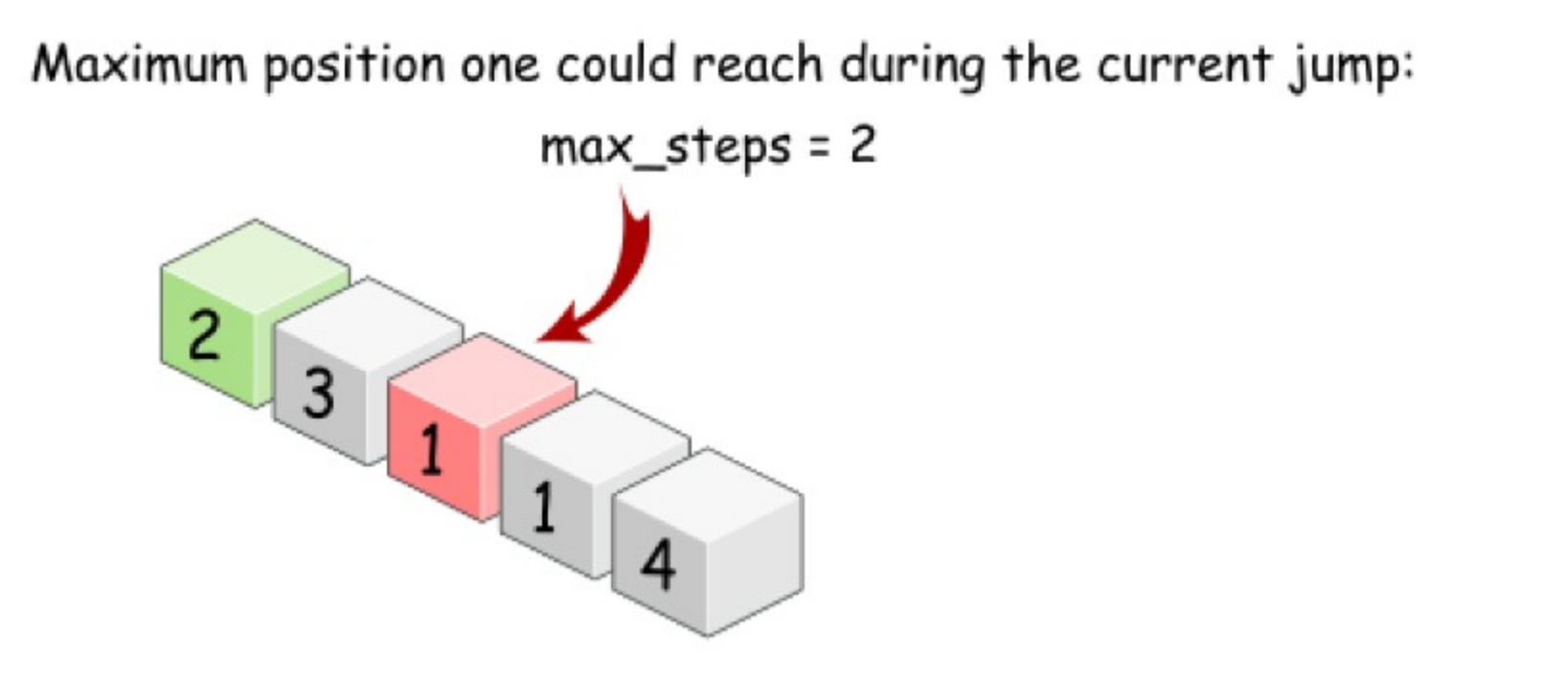
C++ Java Python Copy

```python
class Solution:
    def canJump(self, nums: List[int]) -> bool:
        n = len(nums)

        # max position one could reach
        # starting from index <= i
        max_pos = nums[0]

        for i in range(1, n):
            # if one could't reach this point
            if max_pos < i:
                return False
            max_pos = max(max_pos, nums[i] + i)

        return True
```

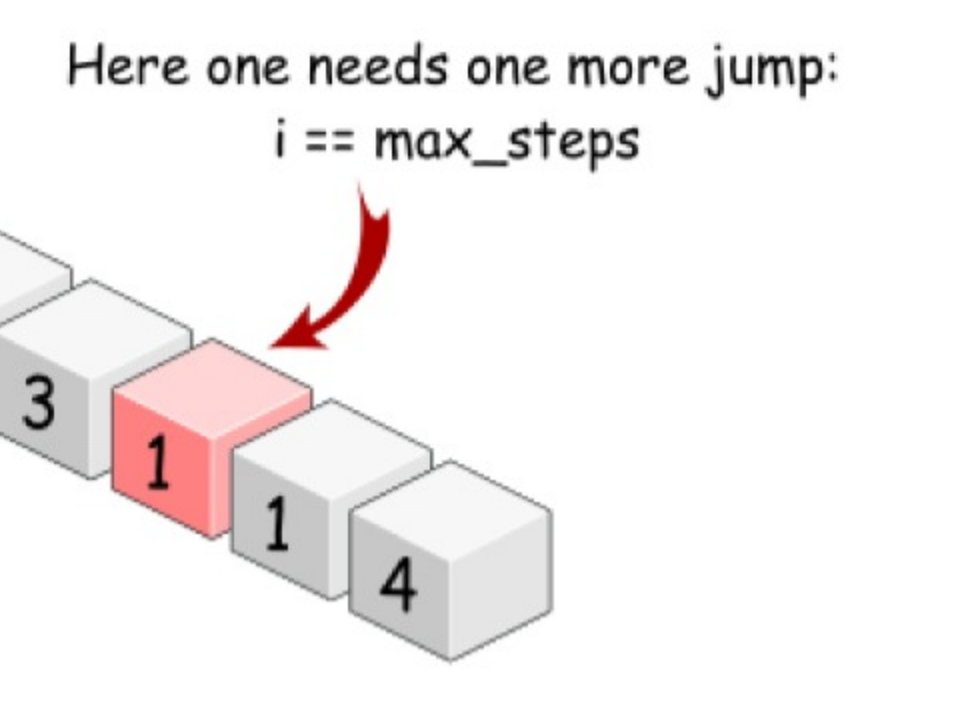
Jump Game II

To solve the current problem, let's use the same *maximum position* `max_pos` that one could reach starting from the current index `i` or *before*. Here it's guaranteed that one can *always* reach the last index, but what is the *minimum* number of jumps needed for that?

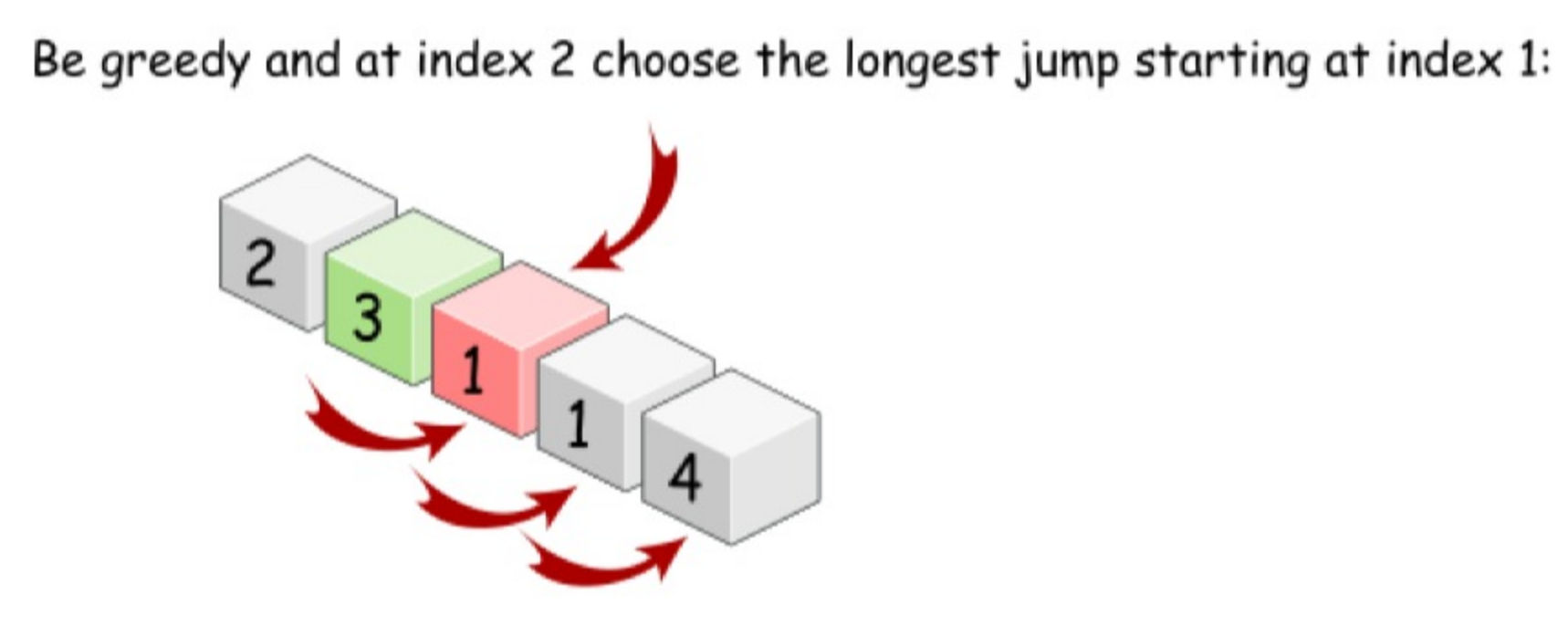
Let's use `max_steps` variable to track the maximum position reachable *during* the current jump. For index 0, `max_steps = nums[0]`.



One could continue to the index `i = max_steps`. At this index one needs one more jump, and this jump has to be performed from the index `i = max_step` or before.



> To minimize the number of jumps, let's follow the "greedy" strategy and choose the longest possible jump. The longest jump is defined by the maximum reachable position that we've computed just above: `max_steps = max_pos`.

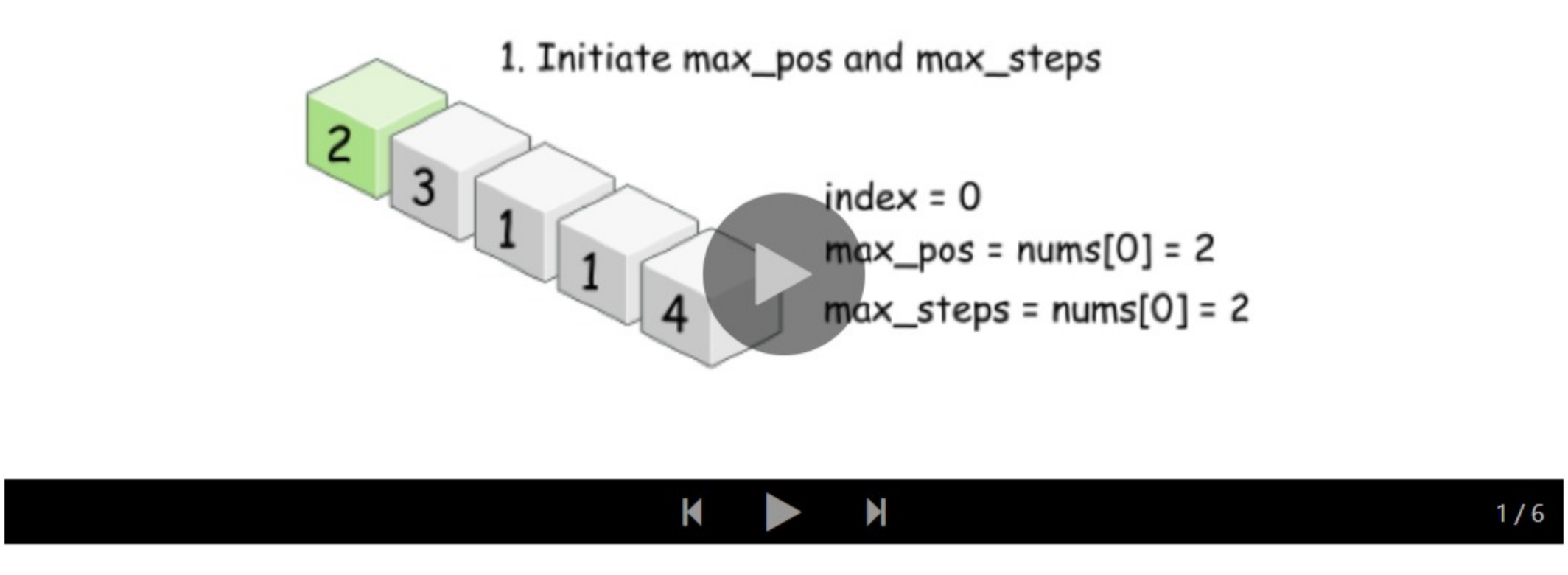


Now we could repeat the above process again and again, till we reach the last index.

Algorithm

- Initiate the maximum position that one could reach starting from the current index `i` or before: `max_pos = nums[0]`.
- Initiate the maximum position reachable *during* the current jump: `max_steps = nums[0]`.
- Initiate number of steps: at least one, if array has more than 1 cell.
- Iterate over number of elements in the input array:
    - If `max_step < i`, one needs one more jump: `jumps += 1`. To minimize the number of jumps, choose the longest possible one: `max_steps = max_pos`.
    - Update `max_pos = max(max_pos, i + nums[i])`.
- Return the number of jumps.

Implementation



C++ Java Python Copy

```python
1  class Solution:
2      def jump(self, nums: List[int]) -> int:
3          n = len(nums)
4          if n < 2:
5              return 0
6  
7          # max position one could reach
8          # starting from index <= i
9          max_pos = nums[0]
10         # max number of steps one could do
11         # inside this jump
12         max_steps = nums[0]
13 
14         jumps = 1
15         for i in range(1, n):
16             # if to reach this point
17             # one needs one more jump
18             if max_steps < i:
19                 jumps += 1
20                 max_steps = max_pos
21             max_pos = max(max_pos, nums[i] + i)
22 
23         return jumps
```

Complexity Analysis

- Time complexity : $\mathcal{O}(N)$, it's one pass along the input array.
- Space complexity : $\mathcal{O}(1)$ since it's a constant space solution.

Rate this article:

Previous Next

Comments: 9 Sort By

Type comment here... (Markdown is supported)

Preview Post

LeiH 54 March 20, 2020 4:44 AM

Shouldn't the Dynamic programming solution of this problem be O(N^2) instead of O(N)?

42 Share Reply

SHOW 4 REPLIES

ats13 41 April 26, 2020 9:19 PM Report

With all due respect, the solution should have been more straightforward. Dear LC admins, and fellow LC users, please endeavour to make the solutions more easily understandable. I took many reads, but could not understand it thoroughly. In such a case, people understand portions of it only. Piece of advice: please understand the solutions, so as to learn the APPROACH behind the solution. That is most important. After going through more than 10 different users' submissions, I found one most concise

Read More

11 Share Reply

suanmeiguo 41 April 23, 2020 3:14 AM

I've spend hours and read the approach multiple times, still don't understand.. Sad..

7 Share Reply

SHOW 7 REPLIES

svm14 6 May 14, 2020 5:38 AM

This was the hardest explanation ever.

3 Share Reply

calvinchankf 2917 April 30, 2020 6:59 PM Report

Actually, you can also do it with binary search and got accepted. My first thought was do a caching approach in O(N^2), and then optimize it to O(NlogN)

Details

Read More

1 Share Reply

__thalaiva__ 82 February 28, 2020 1:49 PM

How is the backtracking solution complexity O(N!)?

1 Share Reply

SHOW 1 REPLY

wonwoo9762 0 May 28, 2020 10:50 AM Report

easier way of formatting the greedy solution

```
class Solution:
    def jump(self, nums: List[int]) -> int:
        start, end = 0, 0
```

Read More

0 Share Reply

Prashanth_123 1 April 26, 2020 5:01 PM

```
class Solution {
    public int jump(int[] nums) {
        if (nums == null || nums.length <= 1) return 0;
```

Read More

0 Share Reply

mishuman 78 March 30, 2020 10:47 PM Report

why are we not updating max_pos and then checking max_steps (line 17). max_steps we can take from index 3 is 4 because we found that in index 1. however, if say, index 3 had a 2 then max_pos would have been 5 and we would still make max_step 4. in other words my question is why max_step is the maximum jump we can make in prior steps?

0 Share Reply

SHOW 1 REPLY