Solution: Jump Game I

Let's solve the Jump Game I challenge using the Greedy pattern.

We'll cover the following

Statement

In a single-player jump game, the player starts at one end of a series of squares, with the goal of reaching the last square.

At each turn, the player can take up to s steps towards the last square, where s is the value of the current square.

For example, if the value of the current square is 3, the player can take either 3 steps, or 2 steps, or 1 step in the direction of the last square. The player cannot move in the opposite direction, that is, away from the last square.

You have been tasked with writing a function to validate whether a player can win a given game or not.

You've been provided with the `nums` integer array, representing the series of squares. The player starts at the first index and, following the rules of the game, tries to reach the last index.

If the player *can* reach the last index, your function returns TRUE; otherwise, it returns FALSE.

Constraints:

- $1 \leq$ `nums.length` $\leq 10^3$
- $0 \leq$ `nums[i]` $\leq 10^3$

Solution

You may have already brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and implementation constraints.

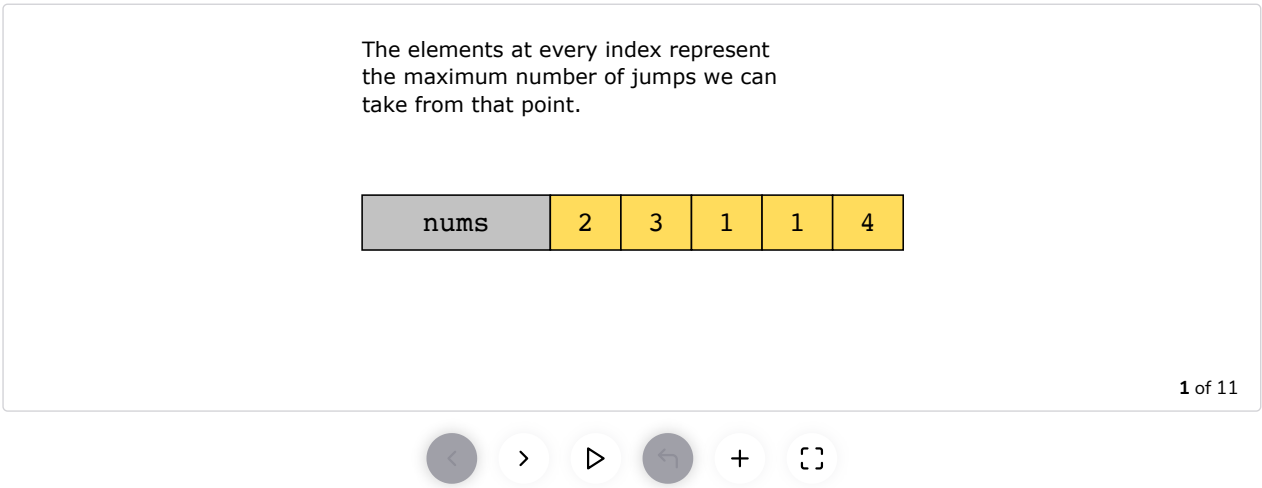
Naive approach

The naive approach is to attempt all possible jump patterns to traverse from the initial position to the final position. The process begins at the starting position and involves jumping to every reachable index. This process is repeated until the last index is reached. In case we are unable to proceed further, a backtrack is

performed to explore alternative paths. This method, although inefficient, ensures that all potential paths are explored to reach the final position. However, the time complexity of the backtracking approach will be exponential.

Optimized approach using the greedy pattern

Alternatively, an optimized approach to solve this problem is using the greedy technique by traversing our array backward. We check the elements one by one from the last element of our array. We keep track of the elements that can reach the ending point directly and verify if there are any possible paths to these indexes. We're "greedy" because we always pick the nearest preceding element that provides a path to the ending point. We can afford to be "greedy", since there is no danger of overshooting the target. The value at each index specifies the longest jump we can make and doesn't restrict us from making a smaller jump if that suits us better.



> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

Setting the last element of the array as our target is the cornerstone of our strategy. In simple terms, this target is the number we're trying to reach from the start of our array. The first step is to actually set the target. While doing this, we can also learn how to change the target, if needed.

Java

```java
import java.util.*;

class JumpGame {

    public static String printArrayWithMarker(int[] arr, int pValue) {
        String out = "[";
        for (int i = 0; i < arr.length - 1; i++) {
            if (i == pValue) {
                out += "«" + String.valueOf(arr[i]) + "»" + ", ";
            } else {
                out += String.valueOf(arr[i]) + ", ";
            }
        }
        if (arr.length - 1 == pValue) {
            out += "«" + String.valueOf(arr[arr.length - 1]) + "»]";
        } else {
            out += String.valueOf(arr[arr.length - 1]) + "]";
        }
        return out;
    }
```

```
21
22      public static boolean jumpGame(int[] nums) {
23          System.out.print("\tSetting the last element in our array as our initial target: ");
24          int targetNumIndex = nums.length - 1;
25          System.out.println(printArrayWithMarker(nums, targetNumIndex));
26          System.out.println("\t\tTarget index: " + targetNumIndex);
27          System.out.println("\n\tChanging the target as we traverse the array backwards:");
28          // Traversing the array from the back to the first element in the array
```

Moving the target backwards

We've learned how to move backward and change our target along the way. Now, let's take our solution one step further. We can use the value at each index to see how many indexes we can jump forward from it.

Java

```
1   import java.util.*;
2
3   class JumpGame {
4
5       public static String printArrayWithMarker(int[] arr, int pValue) {
6           String out = "[";
7           for (int i = 0; i < arr.length - 1; i++) {
8               if (i == pValue) {
9                   out += "«";
10                  out += String.valueOf(arr[i]) + "»" + ", ";
11              } else {
12                  out += String.valueOf(arr[i]) + ", ";
13              }
14          }
15          if (arr.length - 1 == pValue) {
16              out += "«";
17              out += String.valueOf(arr[arr.length - 1]) + "»]";
18          } else {
19              out += String.valueOf(arr[arr.length - 1]) + "]";
20          }
21          return out;
22      }
23
24      public static boolean jumpGame(int[] nums) {
25          System.out.println("\tSetting the last element in our array as our initial target: ");
26          int targetNumIndex = nums.length - 1;
27          System.out.println("\t\t" + printArrayWithMarker(nums, targetNumIndex));
28          System.out.println("\t\tTarget index: " + targetNumIndex);
```

Identifying the longest jump at each index

Next, we can check whether the current target is reachable from the preceding index.

If it is, that preceding index becomes the new target. If it isn't, we check the index before that, and so on, until we reach the start of the array.

If, at any point, we get all the way to the start of the array without finding a preceding index from which the current target is reachable, then there is no way to win the game. On the other hand, if each index is reachable from some preceding index, there is at least one way to win the game.

Java

```
1   import java.util.*;
2
3   class JumpGame {
4
5       public static String printArrayWithMarker(int[] arr, int pValue1, String mrk1a, String mrk1b, int pVa
6           String out = "[";
7           for (int i = 0; i < arr.length - 1; i++) {
8               if (i == pValue1) {
```

```
            if (i == pValue1) {
                out += mrk1a;
                out += String.valueOf(arr[i]) + mrk1b + ", ";
            } else if (i == pValue2) {
                out += mrk2a;
                out += String.valueOf(arr[i]) + mrk2b + ", ";
            } else {
                out += String.valueOf(arr[i]) + ", ";
            }
        }
        if (arr.length - 1 == pValue1) {
            out += mrk1a;
            out += String.valueOf(arr[arr.length - 1]) + mrk1b + "]";
        } else if (arr.length - 1 == pValue2) {
            out += mrk2a;
            out += String.valueOf(arr[arr.length - 1]) + mrk2b + "]";
        } else {
            out += String.valueOf(arr[arr.length - 1]) + "]";
        }
        return out;
    }
```

Jump Game I

Just the code

Here's the complete solution to this problem:

Java

```java
import java.util.*;

class JumpGame {
    public static boolean jumpGame(int[] nums) {
        int targetNumIndex = nums.length - 1;
        for (int i = nums.length - 2; i >= 0; i--) {
            if (targetNumIndex <= (i + nums[i])) {
                targetNumIndex = i;
            }
        }
        if (targetNumIndex == 0)
            return true;
        return false;
    }

    public static void main(String[] args) {
        int[][] nums = {
            {3, 2, 2, 0, 1, 4},
            {2, 3, 1, 1, 9},
            {3, 2, 1, 0, 4},
            {0},
            {1},
            {4, 3, 2, 1, 0},
            {1, 1, 1, 1, 1},
            {4, 0, 0, 0, 1},
            {3, 3, 3, 3, 3},
            {1, 2, 3, 4, 5, 6, 7}
        };
```

Solution summary

1. Set the last index of the array as the target index.
2. Traverse the array backward and verify if we can reach the target index from any of the previous indexes.

- If we can reach it, we update the target index with the index that allows us to jump to the target index.
- We repeat this process until we've traversed the entire array.

3. Return TRUE if, through this process, we can reach the first index of the array. Otherwise, return FALSE.

Time complexity

The time complexity of the above solution is $O(n)$, since we traverse the array only once, where n is the number of elements in the array.

Space complexity

The space complexity of the above solution is $O(1)$, because we do not use any extra space.

Back
Jump Game I

Next
Boats to Save People

Mark as Completed