medium.com

#LeetCode: Java Solution of Jump Game problem - Himanshu Shukla - Medium

Himanshu Shukla

4 minutes

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. Determine if you are able to reach the last index.

Example 1:
Input: [2,3,1,1,4]
Output: true
Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.

Example 2:
Input: [3,2,1,0,4]
Output: false
Explanation: You will always arrive at index 3 no matter what. Its maximum jump length is 0, which makes it impossible to reach the last index.

GIT URL: Java Solution of Leet Code's Jump Game problem

Java Solution 1 :

```java
public boolean canJump(int[] nums) {
    if (nums.length <= 1)
        return true;

    // the largest index that can be reached.
    int largestIndex = nums[0];

    for (int i = 0; i < nums.length; i++) {
        if (largestIndex <= i && nums[i] == 0)
            return false;

        if (i + nums[i] > largestIndex) {
            largestIndex = i + nums[i];
        }

        // is max is enough to reach the end?
        if (largestIndex >= nums.length - 1)
            return true;
    }
    return false;
}
```

Second Solution of Leet Code's Jump Game problem: Last Known Position

Algorithm:

1). We will maintain a variable lastAccuratePosition from which we can reach the last position. Since we can reach the last position from the last index we initialize lastAccuratePosition with the index of last element (i.e nums.length-1).

2). Now we will start iterating the input array from right (second last position) to the left.

3). In each iteration we will calculate furthestJump which is the summation of index and the value at that index (i.e nums[i]+i).

4). We will check if furthestJump is greater than or equal to lastAccuratePosition. If yes, then we will update the value of lastAccuratePosition with the current index.

5). After the iteration we will check if lastAccuratePosition is zero return true, else return false.

GIT URL: Java Solution of Leet Code's Jump Game problem

Java Solution 2 :

```
public boolean canJump(int[] nums) {
    int lastAccuratePosition=nums.length-1;
    int furthestJump=0;
    for(int i=nums.length-2;i>=0;i--) {
        furthestJump=nums[i]+i;
        if(furthestJump>=lastAccuratePosition)
            lastAccuratePosition=i;
    }
    return lastAccuratePosition==0;
}
```

Third Solution of Leet Code's Jump Game problem: Dynamic Programming Top-down

1). We will start marking each index with a certain value (let's say GOOD, BAD and DONTNO), initially all the index's (except the last one) will be marked as DONTNO. We will start computing if the index is GOOD or BAD. Once the index is marked as GOOD or BAD, we won't change it. Since the value is not going to change, that's means it will be computed only once. We store these marked values let's create a array with a name valNums.

2). We will write a recursive function jumpAllowed(). First we will check if the index is not DONTNO (i.e either GOOD or BAD) and if yes, then return true or false. Otherwise we will call jumpAllowed() recursively.

3). The furthestJump from any position could be value at that position or the length of input array (which ever is less)

4). Once we determine the value of the current index, we store it in valNums.

GIT URL: [Java Solution of Leet Code's Jump Game problem using Dynamic Programming Top-down]

Java Solution 3 :

```
public class JumpGameTopDown {
    public static void main(String[] args) {
        int[] nums= {2,3,1,1,4};
        //int[] nums= {3,2,1,0,4};
        System.out.println("Can jump?"+new JumpGameTopDown().canJump(nums));

    }
    /*
     * Second Solution of Leet Code's Jump Game problem: Dynamic Programming Top-down
     *
     * 1). We will start marking each index with a certain value (let's say GOOD, BAD and DONTNO),
     *     initially all the index's (except the last one) will be marked as DONTNO.
     *     We will start computing if the index is GOOD or BAD. Once the index is marked as GOOD or BAD, we won't change it.
```

```
     *     Since the value is not going to change, that's means it will be computed only once.
     *     We store these marked values let's create a array with a name valNums.
     *
     * 2). We will write a recursive function jumpAllowed(). First we will check if the index
     *     is not DONTNO (i.e either GOOD or BAD) and if yes, then return true or false.
     *     Otherwise we will call jumpAllowed() recursively.
     *
     * 3). The furthestJump from any position could be value at that position or the length of input array (which ever is less)
     *
     * 4). Once we determine the value of the current index, we store it in valNums.
     * */
    VALENUM[] valNums;
    public boolean canJump(int[] nums) {
        if(nums.length==0)
            return false;

        //create a VALENUM with length equal to input array
        valNums=new VALENUM[nums.length];

        //set all the values in VALENUM as DONTNO
        for(int i=0; i<nums.length;i++)
            valNums[i]=VALENUM.DONTNO;

        //set the last value of VALENUM as GOOD
        valNums[nums.length-1]=VALENUM.GOOD;

        return jumpAllowed(0, nums);
    }
     public boolean jumpAllowed(int position, int[] nums) {
        //If the current position is already marked as YES or NO, return true of false
         if(valNums[position]!=VALENUM.DONTNO)
             return valNums[position]==VALENUM.GOOD?true:false;

          int furthestJump=Math.min(nums[position]+position, nums.length-1);
         for (int nextPosition = position + 1; nextPosition <= furthestJump; nextPosition++) {
             if(jumpAllowed(nextPosition, nums))
             {
                 valNums[nextPosition]=VALENUM.GOOD;
                 return true;
             }
         }
         valNums[position]=VALENUM.BAD;
         return false;
     }
}
enum VALENUM {
    GOOD, BAD, DONTNO
}
```

Fourth Solution of Leet Code's Jump Game problem: Dynamic Programming Bottom-up

We will change our top-down approach to bottom-up, while doing this we will eliminate the recursion. Since we are not using recursive function jumpAllowed(), we no longer have the method stack overhead.

In this approach also, we will start marking each index with a certain value (let's say GOOD, BAD and DONTNO), initially all the index's (except the last one) will be marked as DONTNO. We will start computing if the index is GOOD or BAD. Once the index is marked as GOOD or BAD, we won't change it.

We start from the right of the array, every time we will query a position to our right. If that position has already be determined as

being GOOD or BAD, we won't recurse it anymore, as we will always get the value from valNums.

GIT URL: [Java Solution of Leet Code's Jump Game problem using Dynamic Programming Bottom-Up]

Java Solution 4 :

```
public boolean canJumpBottomUp(int[] nums) {
    int length=nums.length;
    if (length == 0)
        return false;

    // create a VALENUM with length equal to input array
    valNums = new VALENUM[length];

    // set all the values in VALENUM as DONTNO
    for (int i = 0; i < length; i++)
        valNums[i] = VALENUM.DONTNO;

    // set the last value of VALENUM as GOOD
    valNums[length - 1] = VALENUM.GOOD;

    int furthestJump;
    //we will start from right to left
    for (int i=length-2; i>=0; i--){
         furthestJump=Math.min(nums[i]+i, length-1);
         for(int j=i+1;j<=furthestJump; j++) {
             if(valNums[j]==VALENUM.GOOD) {
                 valNums[i]=VALENUM.GOOD;
                 break;
             }
         }
    }
    return valNums[0]==VALENUM.GOOD;
}
```

-Himanshu Shukla..