Minimum jumps to reach the end

We'll cover the following

Problem Statement

Given an array of positive numbers, where each element represents the max number of jumps that can be made forward from that element, write a program to find the minimum number of jumps needed to reach the end of the array (starting from the first element). If an element is 0, then we cannot move through that element.

Example 1:

```
Input = {2,1,1,1,4}
Output = 3
Explanation: Starting from index '0', we can reach the last index through: 0->2->3
->4
```

Example 2:

```
Input = {1,1,3,6,9,3,0,1,3}
Output = 4
Explanation: Starting from index '0', we can reach the last index through: 0->1->2
->3->8
```

Let's first start with a recursive brute-force solution.

Basic Solution

We will start with the '0'th index and try all options. So, if the value at the current index is 'p', we will try every jump in the range (1 to 'p') from that index. After taking a jump, we recursively try all option from that index.

Here is the code:

Java | JS | Python3 | C++

```
1  import math
```

```
  .
2
3
4  def count_min_jumps(jumps):
5    return count_min_jumps_recursive(jumps, 0)
6
7
8  def count_min_jumps_recursive(jumps, currentIndex):
9    n = len(jumps)
10   # if we have reached the last index, we don't need any more jumps
11   if currentIndex == n - 1:
12     return 0
13
14   if jumps[currentIndex] == 0:
15     return math.inf
16
17   totalJumps = math.inf
18   start, end = currentIndex + 1, currentIndex + jumps[currentIndex]
19   while start < n and start <= end:
20     # jump one step and recurse for the remaining array
21     minJumps = count_min_jumps_recursive(jumps, start)
22     start += 1
23     if minJumps != math.inf:
24       totalJumps = min(totalJumps, minJumps + 1)
25
26   return totalJumps
27
28
29 def main():
30
31   print(count_min_jumps([2, 1, 1, 1, 4]))
32   print(count_min_jumps([1, 1, 3, 6, 9, 3, 0, 1, 3]))
33
34
35 main()
```

The time complexity of the above algorithm is $O(2^n)$, where 'n' is the size of the input array. The 'while loop' can execute a maximum of 'n' times (for the case where we can jump to all the steps ahead) and since in each iteration, the function recursively calls itself, therefore, the time complexity is $O(2^n)$. The space complexity is $O(n)$ which is used to store the recursion stack.

We can clearly see the overlapping subproblem pattern. We can optimize this using memoization to store the results for subproblems.

Top-down Dynamic Programming with Memoization

We can use an array to store the already solved subproblems. Here is the code for this:

Java | JS | Python3 | C++

```
1  import math
2
3
4  def count_min_jumps(jumps):
5    dp = [0 for x in range(len(jumps))]
```

```
  return count_min_jumps_recursive(dp, jumps, 0)


def count_min_jumps_recursive(dp, jumps, currentIndex):
  n = len(jumps)
  # if we have reached the last index, we don't need any more jumps
  if currentIndex == n - 1:
    return 0

  if jumps[currentIndex] == 0:
    return math.inf

  # if we have already solved this problem, return the result
  if dp[currentIndex] != 0:
    return dp[currentIndex]

  totalJumps = math.inf
  start, end = currentIndex + 1, currentIndex + jumps[currentIndex]
  while start < n and start <= end:
    # jump one step and recurse for the remaining array
    minJumps = count_min_jumps_recursive(dp, jumps, start)
    start += 1
    if minJumps != math.inf:
      totalJumps = min(totalJumps, minJumps + 1)

  dp[currentIndex] = totalJumps
  return dp[currentIndex]


def main():

  print(count_min_jumps([2, 1, 1, 1, 4]))
  print(count_min_jumps([1, 1, 3, 6, 9, 3, 0, 1, 3]))


main()
```

Bottom-up Dynamic Programming

Let's try to populate our `dp[]` array from the above solution, working in the bottom-up fashion. As we saw in the above code, we were trying to find the minimum jumps needed to reach every index (if it is within the range) from the current index. We can use this fact to populate our array.

As we know, every index within the range of current index can be reached in one jump. Therefore, we can say that we can reach every index (within the range of current index) in:

```
'jumps to reach current index' + 1
```

So, while going through all the indexes, we will take the minimum value between the current jump-count and the jumps needed to reach the current index + 1.

Here is the code for our bottom-up dynamic programming approach:

Java JS Python3 C++

```python
import math


def count_min_jumps(jumps):
  n = len(jumps)
  # initialize with infinity, except the first index which should be zero as we
  # start from there
  dp = [math.inf for _ in range(n)]
  dp[0] = 0

  for start in range(n - 1):
    end = start + 1
    while end <= start + jumps[start] and end < n:
      dp[end] = min(dp[end], dp[start] + 1)
      end += 1

  return dp[n - 1]


def main():

  print(count_min_jumps([2, 1, 1, 1, 4]))
  print(count_min_jumps([1, 1, 3, 6, 9, 3, 0, 1, 3]))


main()

```

The above solution has a time complexity of $O(n^2)$ (because of the two `for` loops) and space complexity of $O(n)$ to store `dp[]`.

Fibonacci number pattern

We can clearly see that this problem follows the Fibonacci number pattern. The only difference is that every Fibonacci number is a sum of the two preceding numbers, whereas in this problem every number is the minimum of two numbers (start and end):

```
    dp[end] = Math.min(dp[end], dp[start]+1);
```

Back

Number factors

Next

Minimum jumps with fee

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/minimum-jumps-to-reach-the-end__pattern-3-fibonacci-numbers__grokking-dynamic-programming-patterns-for-coding-interviews)