**Minimum jumps to reach the end**

**WE'LL COVER THE FOLLOWING**

* + - [Problem Statement](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r#problem-statement)
    - [Basic Solution](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r#basic-solution)
    - [Top-down Dynamic Programming with Memoization](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r#top-down-dynamic-programming-with-memoization)
    - [Bottom-up Dynamic Programming](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r#bottom-up-dynamic-programming)
    - [Fibonacci number pattern](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r#fibonacci-number-pattern)

**Problem Statement**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r#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**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r#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](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r)
* [JS](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r)
* [Python3](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r)
* [C++](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r)

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

    return this.countMinJumpsRecursive(jumps, 0);

  }

  private int countMinJumpsRecursive(int[] jumps, int currentIndex) {

    // if we have reached the last index, we don't need any more jumps

    if( currentIndex == jumps.length - 1)

      return 0;

    if (jumps[currentIndex] == 0)

      return Integer.MAX\_VALUE;

    int totalJumps = Integer.MAX\_VALUE;

    int start = currentIndex + 1;

    int end = currentIndex + jumps[currentIndex];

    while(start < jumps.length && start <= end) {

      // jump one step and recurse for the remaining array

      int minJumps = countMinJumpsRecursive(jumps, start++);

      if(minJumps != Integer.MAX\_VALUE)

        totalJumps = Math.min(totalJumps, minJumps + 1);

    }

    return totalJumps;

  }

  public static void main(String[] args) {

    ArrayJump aj = new ArrayJump();

    int[] jumps = {2, 1, 1, 1, 4};

    System.out.println(aj.countMinJumps(jumps));

    jumps = new int[]{1, 1, 3, 6, 9, 3, 0, 1, 3};

    System.out.println(aj.countMinJumps(jumps));

  }

}





RUN

SAVERESET

The time complexity of the above algorithm is O(2^n)*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)*O*(2​*n*​​). The space complexity is O(n)*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**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r#top-down-dynamic-programming-with-memoization)

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

* [Java](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r)
* [JS](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r)
* [Python3](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r)
* [C++](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r)

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

    // If an element is 0, then we cannot move through that element

    if (jumps[currentIndex] == 0)

      return Integer.MAX\_VALUE;

    // if we have already solved this problem, return the result

    if(dp[currentIndex] != 0)

      return dp[currentIndex];

    int totalJumps = Integer.MAX\_VALUE;

    int start = currentIndex + 1;

    int end = currentIndex + jumps[currentIndex];

    while(start < jumps.length && start <= end) {

      // jump one step and recurse for the remaining array

      int minJumps = countMinJumpsRecursive(dp, jumps, start++);

      if(minJumps != Integer.MAX\_VALUE)

        totalJumps = Math.min(totalJumps, minJumps + 1);

    }

    dp[currentIndex] = totalJumps;

    return dp[currentIndex];

  }

  public static void main(String[] args) {

    ArrayJump aj = new ArrayJump();

    int[] jumps = {2, 1, 1, 1, 4};

    System.out.println(aj.countMinJumps(jumps));

    jumps = new int[]{1, 1, 3, 6, 9, 3, 0, 1, 3};

    System.out.println(aj.countMinJumps(jumps));

  }

}





RUN

SAVERESET

**Bottom-up Dynamic Programming**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r#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](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r)
* [JS](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r)
* [Python3](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r)
* [C++](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

class ArrayJump {

  public int countMinJumps(int[] jumps) {

    int[] dp = new int[jumps.length];

    //initialize with infinity, except the first index which should be zero as we start from there

    for(int i=1; i<jumps.length; i++)

      dp[i] = Integer.MAX\_VALUE;

    for(int start=0; start < jumps.length-1; start++) {

      for(int end=start+1; end <= start+jumps[start] && end < jumps.length; end++)

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

    }

    return dp[jumps.length-1];

  }

  public static void main(String[] args) {

    ArrayJump aj = new ArrayJump();

    int[] jumps = {2, 1, 1, 1, 4};

    System.out.println(aj.countMinJumps(jumps));

    jumps = new int[]{1, 1, 3, 6, 9, 3, 0, 1, 3};

    System.out.println(aj.countMinJumps(jumps));

  }

}





RUN

SAVERESET

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

**Fibonacci number pattern**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r#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);

**MARK AS COMPLETED**

[**←    Back**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/NE52PnMY376)

Number factors

[**Next    →**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G)

Minimum jumps with fee

Stuck? Get help on

[**DISCUSS**](https://discuss.educative.io/c/grokking-dynamic-programming-patterns-for-coding-interviews-design-gurus/pattern-4-fibonacci-numbers-minimum-jumps-to-reach-the-end)

Send feedback

17 Recommendations