Minimum Number of Jumps to Reach End

Difficulty: Medium; **Asked in:** Google, Amazon, Walmart, Adobe, eBay

Key takeaway: An excellent problem to learn performance optimization and problem-solving using dynamic programming. The efficient single-loop solution is intuitive and worth exploring.

Let's understand the problem!

Given an array of positive integers, write a program to reach the last index using the minimum number of jumps.

> We are initially at the first index of the array.
>
> Each element in the array represents the **maximum jump length** from that position. For example, if A[i] = x, this means that from the index **i**, we can jump to any one of the indices i + 1, i + 2, ..., i + x.
>
> Assume that we always reach the last index.
>
> There can be several ways to reach the end using the minimum number of jumps. Our aim is to return only the minimum number of jumps required.

Important note: Before moving on to the solutions, we recommend trying this problem on paper for at least 15 or 30 minutes. Enjoy problem-solving!

Example 1

Input: A[] = [1, 3, 5, 8, 10, 2, 6, 7, 6, 8, 9], Output: 3

Explanation: **1-> 3 -> 8 -> 9** or **1-> 3 -> 10 -> 9**

Here we have 11 elements in the array, so the start index is 0, and the end index is 10.

A[0] = 1: We can jump from the 0th index to the 1st index.

A[1] = 3: We can jump a maximum of 3 steps from index 1. In other words, we can reach indices 2, 3, and 4 from the 1st index.

A[2] = 5: We cannot reach the last index from index 2 because the maximum reachable index from index 2 would be 2 + 5 = 7.

A[3] = 8: We can easily reach the last index from index 3 because the maximum reachable index from index 3 would be 3 + 8 = 11. In other words, we can take a 7-step jump from index 3 to reach the last index 10.

A[4] = 10: We can easily reach the last index from index 4 because the maximum reachable index from index 4 would be 4 + 10 = 14. In other words, we can take a 6-step jump from index 4 to reach the last index 10.

Example 2

Input: A[] = [2, 3, 1, 1, 4], Output: 2

Explanation: **2->3->4**. The minimum number of jumps to reach the last index is 2. We jump 1 step from index 0 to 1, then 3 steps to the last index.

Discussed solution approaches

Brute force approach using recursion

Bottom-up approach using dynamic programming

Efficient in-place approach using single loop

Another efficient in-place approach using single loop

Brute force approach using recursion

Solution idea

There can be multiple possible ways to reach the end, but our objective is to find the path with the minimum number of jumps. So one basic solution is to calculate the jump count for all possible ways to reach the end and return the minimum value among them. The critical question is: How can we explore all the possible ways to reach the end? Let's think!

When we need to explore all possible ways, we can apply the concept of recursion, where we solve the larger problem by using the solutions to smaller subproblems.

Here is an idea for a recursive solution: We start from the 0th index and recursively call for all the indices reachable from the 0th index. In other words, we can calculate the minimum number of jumps needed to reach the last index by considering the minimum number of jumps required to reach the last index from an index that is reachable from the 0th index. (Think about it!)

Recursive structure

Suppose our initial function call is **minJumps(A[], start, end)**, where the end index of A[] is **end**, and the start index is **start**. From the **start** index, we can take a single jump of **i** number of steps, where **1 <= i <= A[start]**.

What would be the smaller subproblem if we take a single jump of i number of steps from the **start**? The answer is simple: Find the minimum number of jumps to reach the last index from the **(start + i)th** index, i.e., **minJumps(A[], start + i, end)**.

To find the overall minimum jump count, we recursively calculate the jump count for all possible values of **i (1 <= i <= A[start])** and find the minimum among them. We also add **1** because we take one jump to reach index i from the start index.

For all i reachable from start, i.e., i = 1 to A[start]:

minJumps(A[], start, end) = 1 + min(minJumps(A[], start + i, end))

Base case: This is a case of a single or zero-element array where no jump is required, i.e., **if (start >= end)**, we return **0**.

Solution code C++

```
int minJump(int A[], int start, int end)
{
    if (start >= end)
        return 0;

    int minJumpCount = INT_MAX;
    for (int i = 1; i <= A[start] && i < end; i = i + 1)
    {
        int jumpCount = 1 + minJump(A, start + i, end);
        if (jumpCount < minJumpCount)
            minJumpCount = jumpCount;
    }
    return minJumpCount;
}
```

Solution code Python

```
def minJump(A, start, end):
    if start >= end:
        return 0

    minJumpCount = sys.maxsize
    for i in range(1, A[start] + 1):
        if i < end:
            jumpCount = 1 + minJump(A, start + i, end)
            if jumpCount < minJumpCount:
                minJumpCount = jumpCount

    return minJumpCount
```

Solution analysis

We are recursively exploring all possible ways to reach the end. So there are two possibilities for each index from 1 to n - 1: Either we take a jump from that index or do not take a jump. So the total number of possible ways is 2^(n-1) or O(2^n), which is exponential.

We can also analyze it by writing a recurrence relation. Suppose the time complexity to reach the end in an n-size array is T(n). Now after taking i number of jumps at the start, the input size of the smaller sub-problem is n - i. So the time complexity of the input size n - i is **T(n - i)**.

After this, we are recursively calling for all indexes reachable from the first index. So the maximum possible jump is equal to the value of A[start]. In the worst case, the value of this jump can be n - 1.

So here is the recurrence relation: T(n) = c + ∑(i = 1 to A[start]) T(n - i). If we put A[start] = n - 1, we get the upper bound: **T(n) = c + ∑(i = 1 to n - 1) T(n - i)**. So term T(n) is the sum of the constant c and all previous terms T(n-1), T(n-2), ..., T(1).

Equation 1: T(n) = c + T(1) + T(2) + ... + T(n-1).

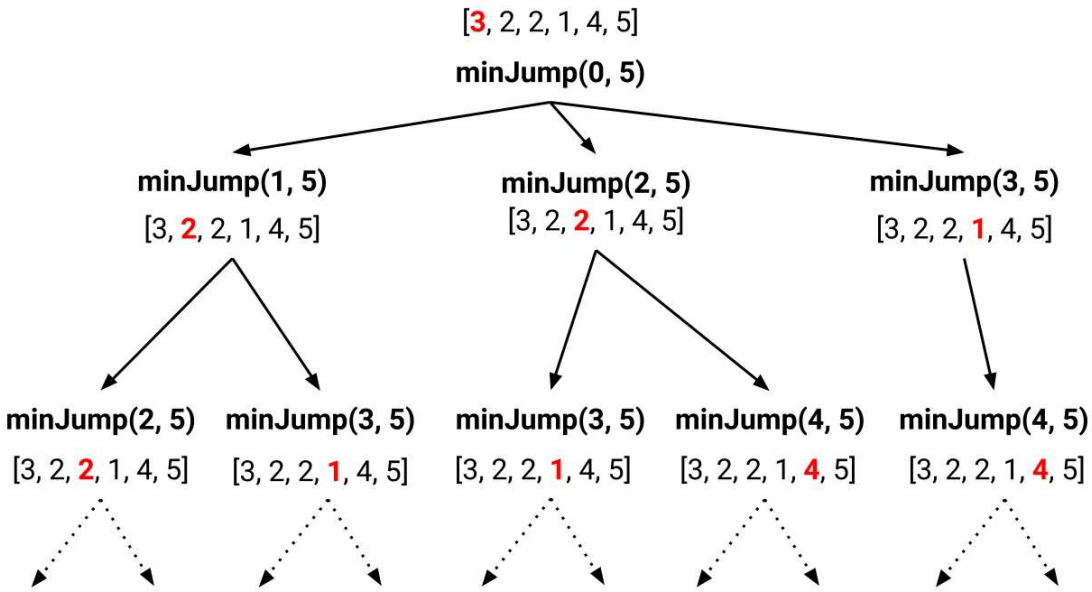
If we put n = n - 1 in the above equation, we will get:

Equation 2: T(n - 1) = c + T(1) + T(2) + ... + T(n-2)

If we subtract the equation 1 and 2, we will get =>T(n) - T(n - 1) = T(n - 1) => **T(n) = 2T(n - 1)**. Now we have a simple recurrence relation and we can easily solve it using the substitution method.

```
T(n)
= 2*T(n - 1)
= 2^2*T(n - 2)
= 2^3*T(n - 3)
... and so on
= 2^(n - 1)*T(1)
= 2^(n - 1)
=  O(2^n)
```

So finding the min jump is an exponential function of n, which is inefficient. In retrospect, this is not surprising because we are exploring all possibilities. If we create a recursion tree, we can notice overlapping subproblems. For example, if we observe the following picture, the sub-problem minJump(2, 5) is coming two times, minJump(3, 5) is coming three times, and so on.



Critical points to notice

This is an optimization problem with an exponentially large solution space

Sub-problems are overlapping i.e. we are solving the same subproblems again and again.

In the worst case, the total number of different subproblems is equal to n, i.e., sub-problems of size 1, sub-problem of size 2...up to sub-problem of size n. So time complexity is exponential because these n sub-problems are repeated during the recursion.

We can use dynamic programming to solve this problem efficiently.

Bottom-up approach using dynamic programming

Solution idea

Since we have identified this as a dynamic programming problem, we can efficiently solve it using a bottom-up approach. Here our goal is to calculate the solution of smaller sub-problems iteratively and store their results in a table.

1. **Table structure and size:** Only one variable decreases by 1 during the recursion, so we can take a one-dimensional table jumps[n] to store solutions of the n different sub-problems (Table size is equal to the total number of sub-problems). Here, jumps[i] indicate the minimum number of jumps needed to reach A[i] from A[0].
2. **Table initialization:** We initialize the table using the basic solution, i.e., jumps[0] = 0. There is no need to jump when there is a single element in the array. We also initialize the remaining positions of the jumps[] array with INT_MAX.
3. **Iterative structure to fill the table:** We can define an iterative structure to fill the table using the recursive solution. This approach of storing results in a table is similar to the longest-increasing subsequence problem.

   jumps[i] = For (j = 0 to i - 1) min (jumps [j] + 1), where j + A[j] >= i.
4. **Termination and returning final solution:** Our final answer will be stored at jumps[n-1], and we return this value.

Solution code C++

```
int minJump(int A[], int n)
{
    int jump[n];
    jump[0] = 0;
    for (int i = 1; i < n; i = i + 1)
        jump[i] = INT_MAX;
    for (int i = 1; i < n; i = i + 1)
    {
```

```
            for (int j = 0; j < i; j = i + 1)
            {
                if (i <= j + A[j] && jump[j] != INT_MAX)
                    jump[i] = min(jump[i], jump[j] + 1);
            }
        }
        return jump[n - 1];
    }
```

Solution code Python

```
def minJump(A, n):
    jump = [0] + [sys.maxsize] * (n - 1)
    for i in range(1, n):
        for j in range(i):
            if i <= j + A[j] and jump[j] != sys.maxsize:
                jump[i] = min(jump[i], jump[j] + 1)
    return jump[n -1]
```

Solution analysis

Time complexity = Time complexity of initializing jump[] array + Time complexity of nested loops to store values in jump[] array = O(n) + O(n^2) = O(n^2). Similarly, space complexity = O(n) for using n size jump[] array.

Efficient in-place approach using single loop

Solution idea

The above dynamic programming solution is much more efficient than brute force solution. But critical questions are: Can we further improve the time complexity, space complexity, or both? Can we try to remove inner loop of the above solution? Can we track the value of minimum jump using some variable?

Here is an improvement insight: From any jump point i, we can reach any index from **(i + 1)** to **(i + A[i])**, and there will be some value between range **A [i + 1]** to **A [i + A[i]]**, which can provide the farthest reach from that range. So we define lower and upper end of the current jump point and calculate the farthest reach in that range. Whenever we reach the upper endpoint of a range, we update the upper end with the value of farthest reach and increment jump count. We continue this process till the value of farthest reachable index is greater than n - 1.

Let's define four variables to simulate the above process:

jump: To store the value of jump count.

currStart and **currEnd:** To store the range of the current jump point.

currMaxReach: To store the farthest reach possible in the range [currStart, currEnd].

Solution steps

We initialize jump to 0, currStart to 0, currEnd to 0, and currMaxReach to -1.

Next, we traverse the array from currStart = 1 to n - 2, updating currMaxReach as we go. Note that currStart will be less than n - 1, because we don't need to jump again when we reach the last element of the array.

Once currStart equals currEnd, we trigger a jump and update the range of the next jump point. To update the range, we set the value of currEnd to currMaxReach.

During the update of currMaxReach, we also need to check whether currMaxReach is greater than n - 1. If it is true, we can directly reach the end of the array from the currStart by taking a single jump. So we increment the value of jump and break out of the loop.

At the end of the loop, we return the value stored in jump.

Solution code C++

```
int minJump(int A[], int n)
{
    if (n <= 1)
        return 0;

    int jump = 0, currStart = 0;
    int currEnd = 0, currMaxReach = -1;
    while (currStart < n - 1)
    {
        currMaxReach = max(currMaxReach, currStart + A[currStart])
        if (currMaxReach >= n - 1)
        {
            jump = jump + 1;
            break;
        }
        if (currStart == currEnd)
        {
            jump = jump + 1;
            currEnd = currMaxReach;
        }
        currStart = currStart + 1;
    }
    return jump;
}
```

Solution code Python

```
def minJump(A, n):
    if n <= 1:
        return 0

    jump = 0
    currStart = 0
    currEnd = 0
    currMaxReach = -1
    while currStart < n - 1:
        currMaxReach = max(currMaxReach, currStart + A[currStart])
        if currMaxReach >= n - 1:
```

```
            jump = jump + 1
            break
        if currStart == currEnd:
            jump = jump + 1
            currEnd = currMaxReach
        currStart = currStart + 1
    return jump
```

Another efficient in-place approach using single loop

Solution idea

In this approach, we define three variables: **currMaxReach**, **stepsCount**, and **jump**. We set both the jump and stepsCount variables to the value of the first index of the array. Here, currMaxReach is the maximum we can reach from that index, which is the index plus the value of the index (the jump value). So, we keep updating it in each iteration (starting from 1 to n - 2), so that whenever we move forward, the variable currMaxReach stores the max reach by using **currMaxReach = max(currMaxReach, A[start] + start)**.

Also, at each iteration, we reduce our stepsCount variable by 1. As we move forward, we consume 1 step each time. So, whenever we run out of steps, it means we need to take one jump. So, we increase the **jump** variable and update the **stepsCount** variable to the value **(currMaxReach - start)**, which is the maximum reach possible from the current index. This means that we can take those steps, and then we need to jump again.

So, we return **jump + 1** as our output in this solution since we only jump after running out of steps. Also, we need to note that: we moved only till the second last element and not the last element since at the last step, we do not need to consume one more step as we are already there and no need to jump more.

Solution code C++

```
int minJump(int A[], int n)
{
    if (n <= 1)
        return 0;

    int currMaxReach = A[0];
    int stepsCount = A[0];
    int jump = 0;
    for (int start = 1; start < n - 1; start = start + 1)
    {
        currMaxReach = max(currMaxReach, start + A[start]);
        stepsCount = stepsCount - 1;
        if (stepsCount == 0)
        {
            jump = jump + 1;
            stepsCount = currMaxReach - start;
        }
    }
    return jump + 1;
}
```

Solution code Python

```
def minJump(A, n):
    if n <= 1:
        return 0

    currMaxReach = A[0]
    stepsCount = A[0]
    jump = 0
    for start in range(1, n - 1):
        currMaxReach = max(currMaxReach, start + A[start])
        stepsCount = stepsCount - 1
        if stepsCount == 0:
            jump = jump + 1
            stepsCount = currMaxReach - start

    return jump + 1
```

Solution analysis

In both the single loop solution, we are doing constant operations at each iteration. So time complexity = O(n). We are using constant extra space, so space complexity = O(1)

Critical ideas to think!

Can we think of implementing the single loop solution by traversing the array from right to left instead of left to right?

How would we modify the above approach if there is no guarantee to reach the end? In other words, return true if we can reach the last index, or false otherwise.

Does the single solution look like a greedy approach?

What would be the space complexity of the recursive solution?

How do we modify the above approaches to output the elements involved in reaching the end-point using the minimum number of jumps?

Comparison of time and space complexities

Using recursion: Time = O(2^n) , Space = O(n).

Using bottom-up approach: Time = O(n^2), Space = O(n).

Using a single loop : Time = O(n) , Space = O(1).

Another solution using a single loop: Time = O(n), Space = O(1).

Suggested coding problems to practice

Jump Game

Jump Game III

Jump Game VII

Minimum number of Fibonacci jumps to reach the end

Paths requiring a minimum number of jumps to reach the end of an array

If you have any queries or feedback, please write us at contact@enjoyalgorithms.com. Enjoy learning, Enjoy algorithms!

Prev Chapter Next Chapter

Author: Shubham Gautam Reviewer: EnjoyAlgorithms Team

More from EnjoyAlgorithms

Count Total Number of Unique Binary Search Trees with n Keys

Write a program to find the number of structurally unique binary search trees (BSTs) that have exactly n nodes, where each node has a unique integer key ranging from 1 to n. In other words, we need to determine the count of all possible BSTs that can be formed using n distinct keys.

Read More

Inorder Successor of a Node in Binary Search Tree (BST)

Given a node x and the root of a binary search tree, write a program to find the in-order successor of node x. The in-order successor of node x is a node that will be visited just after node x during an in-order traversal. In other words, the in-order successor of a node x is the node with the smallest key greater than x->key.

Read More

Equilibrium Index of an Array

Write a program to find the equilibrium index of an array. An array's equilibrium index is an index such that the sum of elements at lower indexes equals the sum of elements at higher indexes. Note: This is an excellent coding question to learn time and space complexity optimization using a prefix array and a single loop using variables.

Read More

Sort an Array of 0s, 1s and 2s (Dutch National Flag Problem)

Given an array consisting of 0s, 1s, and 2s, write a program to sort this array of 0, 1, and 2 in ascending order. We need to sort the array in O(n) time complexity without using sorting algorithms or extra space. Note: This is a variation of the Dutch national flag problem and an excellent problem to learn problem solving using three pointers.

Read More

Insertion in Binary Search Tree (BST)

The root of the binary search tree and a key k is given. Write a program to insert key k into the binary search tree. Note: BST structure will change after the insertion. So we need to perform insertion in such a way that the BST property continues to hold. In this blog, we have discussed recursive and iterative implementations of insertion in BST.

Read More

Generating all K-combinations

Given two numbers, n and K, write a program to find all possible combinations of K numbers from 1 to n. You may return the answer in any order. Note: This is an excellent problem for learning problem-solving using the inclusion and exclusion principle of combinatorics and backtracking. We can apply similar ideas to solve other problems.

Read More

Self-paced Courses and Blogs

Coding Interview

DSA Course

DSA Blogs

Machine Learning

ML Course

ML Blogs

System Design

SD Course

SD Blogs

OOP Concepts

OOP Course

OOP Blogs

Our Newsletter

Subscribe to get well designed content on data structure and algorithms, machine learning, system design, object orientd programming and math.

Email address

Subscribe