**Recursion I**

**Introduction**

Recursion is an important concept in computer science. It is a foundation for many other algorithms and data structures. However, the concept of recursion can be tricky to grasp for many beginners.

Before getting started with this card, we strongly recommend that you complete the [binary tree](https://leetcode.com/explore/learn/card/data-structure-tree/) and the [stack](https://leetcode.com/explore/learn/card/queue-stack/) Explore cards first.

In this Explore card, we answer the following questions:

1. What is recursion? How does it work?
2. How to solve a problem recursively?
3. How to analyze the time and space complexity of a recursive algorithm?
4. How can we apply recursion in a better way?

After completing this card, you will feel more confident in solving problems recursively and analyzing the complexity on your own.

Before you start, bear in mind that should you have any questions or comments, you can always post them on the [Discussion](https://leetcode.com/discuss/explore/recursion-i) forum that is located at the end of this card. We'll do our best to respond to you as soon as we can.

Principle of Recursion

 Principle of Recursion

 Reverse String

 Solution - Reverse String

 Recursion Function

 Swap Nodes in Pairs

Recurrence Relation

 Recurrence Relation

 Reverse Linked List

 Search in a Binary Search Tree

 Pascal's Triangle II

Memoization

 Duplicate Calculation in Recursion

 Fibonacci Number

 Climbing Stairs

Complexity Analysis

 Time Complexity - Recursion

 Space Complexity - Recursion

 Tail Recursion

 Maximum Depth of Binary Tree

 Pow(x, n)

Conclusion

 Conclusion - Recursion I

 Merge Two Sorted Lists

 K-th Symbol in Grammar

 Unique Binary Search Trees II

Principle of Recursion

In this chapter, we will:

* Explain the basic concept of recursion;
* Demonstrate how to apply the recursion to solve certain problems;
* Finally provide some exercises for you to practice recursion.

**Principle of Recursion**

Recursion is an approach to solving problems using a function that calls itself as a subroutine.

You might wonder how we can implement a function that calls itself. The trick is that each time a recursive function calls itself, it reduces the given problem into subproblems. The recursion call continues until it reaches a point where the subproblem can be solved without further recursion.

A recursive function should have the following properties so that it does not result in an infinite loop:

1. A simple base case (or cases) — a terminating scenario that does not use recursion to produce an answer.
2. A set of rules, also known as recurrence relation that reduces all other cases towards the base case.

Note that there could be multiple places where the function may call itself.

### **Example**

Let's start with a simple programming problem:

Print a string in reverse order.

You can easily solve this problem iteratively, i.e. looping through the string starting from its last character. But how about solving it recursively?

First, we can define the desired function as printReverse(str[0...n-1]), where str[0] represents the first character in the string. Then we can accomplish the given task in two steps:

1. printReverse(str[1...n-1]): print the substring str[1...n-1] in reverse order.
2. print(str[0]): print the first character in the string.

Notice that we call the function itself in the first step, which by definition makes the function recursive.

Here is the code snippet:

|  |
| --- |
| private static void printReverse(char [] str) {  helper(0, str);  }  private static void helper(int index, char [] str) {  if (str == null || index >= str.length) {  return;  }  helper(index + 1, str);  System.out.print(str[index]);  } |

Next, you will find an exercise that is slightly different from the above example. You should try to solve it using recursion.

*Note:* For this exercise, we also provide a detailed solution in this Explore chapter.

**Reverse String**

Write a function that reverses a string. The input string is given as an array of characters char[].

Do not allocate extra space for another array, you must do this by **modifying the input array**[**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) with O(1) extra memory.

You may assume all the characters consist of [printable ascii characters](https://en.wikipedia.org/wiki/ASCII#Printable_characters).

**Example 1:**

**Input:** ["h","e","l","l","o"]

**Output:** ["o","l","l","e","h"]

**Example 2:**

**Input:** ["H","a","n","n","a","h"]

**Output:** ["h","a","n","n","a","H"]

   Hide Hint #1

The entire logic for reversing a string is based on using the opposite directional two-pointer approach!

## Solution

#### **Overview**

Life is short, use Python. (c)

|  |
| --- |
| class Solution:  def reverseString(self, s):  s.reverse() |

Speaking seriously, let's use this problem to discuss two things:

* Does in-place mean constant space complexity?
* Two pointers approach.

#### **Approach 1: Recursion, In-Place, O(N) Space**

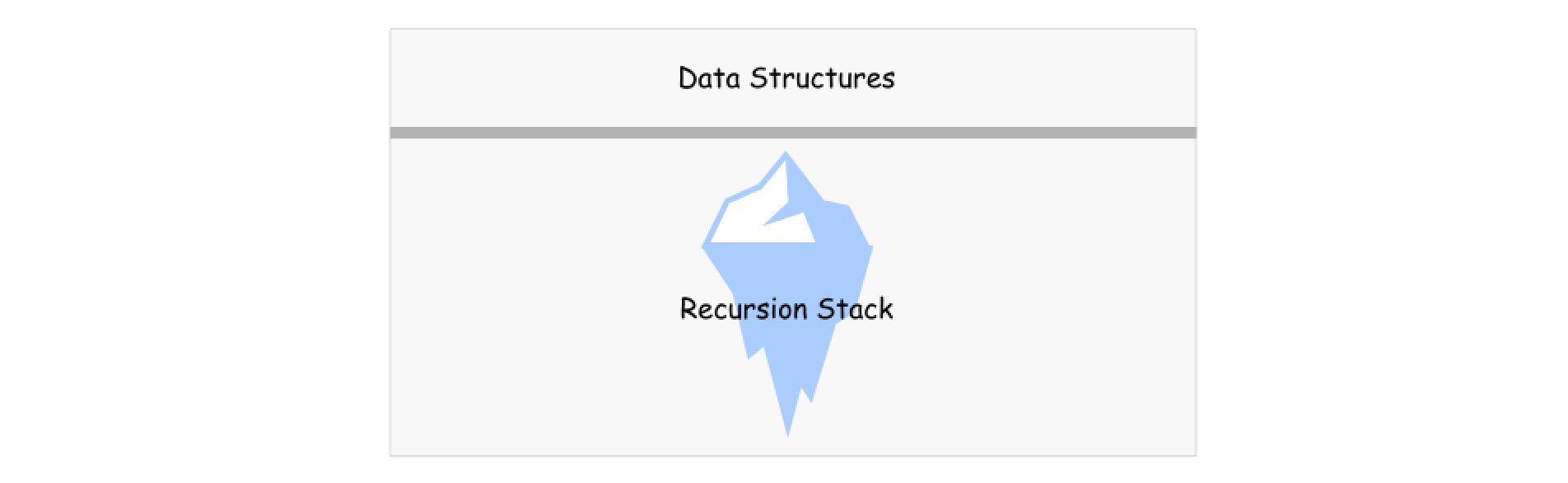
**Does in-place mean constant space complexity?**

No. [By definition](https://en.wikipedia.org/wiki/In-place_algorithm), an in-place algorithm is an algorithm which transforms input using no auxiliary data structure.

The tricky part is that space is used by many actors, not only by data structures. The classical example is to use recursive function without any auxiliary data structures.

Is it in-place? Yes.

Is it constant space? No, because of recursion stack.



**Algorithm**

Here is an example. Let's implement recursive function helper which receives two pointers, left and right, as arguments.

* Base case: if left >= right, do nothing.
* Otherwise, swap s[left] and s[right] and call helper(left + 1, right - 1).

To solve the problem, call helper function passing the head and tail indexes as arguments: return helper(0, len(s) - 1).

|  |
| --- |
| class Solution {  public void helper(char[] s, int left, int right) {  if (left >= right) return;  char tmp = s[left];  s[left++] = s[right];  s[right--] = tmp;  helper(s, left, right);  }  public void reverseString(char[] s) {  helper(s, 0, s.length - 1);  }  } |

**Complexity Analysis**

* Time complexity : O(*N*) time to perform *N*/2 swaps.
* Space complexity : O(*N*) to keep the recursion stack.

#### **Approach 2: Two Pointers, Iteration, O(1) Space**

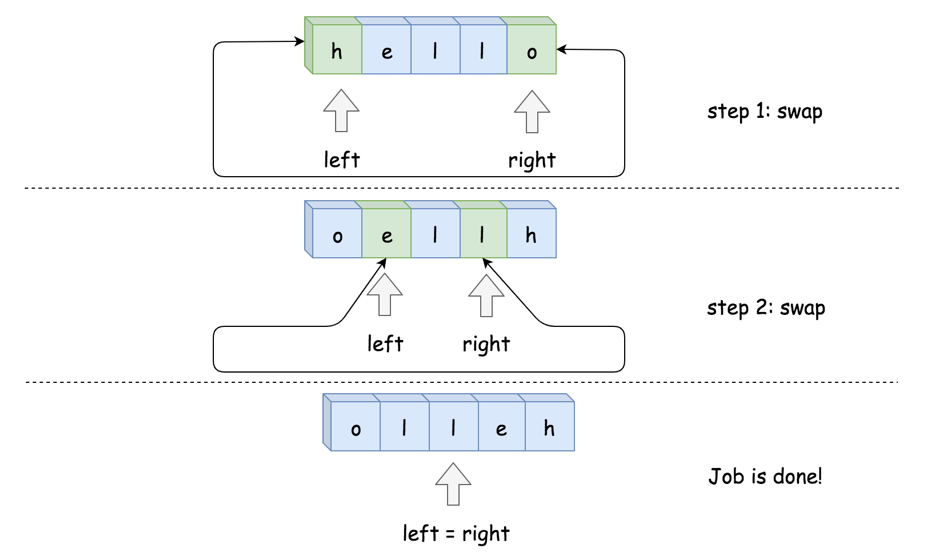
**Two Pointers Approach**

In this approach, two pointers are used to process two array elements at the same time. Usual implementation is to set one pointer in the beginning and one at the end and then to move them until they both meet.

Sometimes one needs to generalize this approach in order to use three pointers, like for classical [Sort Colors problem](https://leetcode.com/articles/sort-colors/).

**Algorithm**

* Set pointer left at index 0, and pointer right at index n - 1, where n is a number of elements in the array.
* While left < right:
  + Swap s[left] and s[right].
  + Move left pointer one step right, and right pointer one step left.



**Implementation**

|  |
| --- |
| class Solution {  public void reverseString(char[] s) {  int left = 0, right = s.length - 1;  while (left < right) {  char tmp = s[left];  s[left++] = s[right];  s[right--] = tmp;  }  }  } |

**Complexity Analysis**

* Time complexity : O(*N*) to swap *N*/2 element.
* Space complexity : O(1), it's a constant space solution.

**Solution - Reverse String**

In this article, we present a sample solution for the problem of [Reverse String](https://leetcode.com/explore/learn/card/recursion-i/250/principle-of-recursion/1440/).

The problem is not difficult, yet the trick part is that we have an additional **constraint** for the problem, i.e. one must modify the string with O(1) extra space.

Let's define the problem as the function reverseString(str[0...n-1]), where str[0...n-1] is a list of characters with the first character denoted as str[0].

Below, we will discuss how we can solve this problem with recursion.

### ***First Attempt***

If we follow the idea of the problem of printing a string in reversed order, as we presented in [the first article](https://leetcode.com/explore/learn/card/recursion-i/250/principle-of-recursion/1439/) of this card, we might come up with the following algorithm:

1. take the leading character str[0] from the input string.
2. call the function itself on the remaining substring, i.e. reverseString(str[1...n-1]).
3. then append the leading character to the result returned in the step (2).

The above algorithm could work, except that it does not meet the constraint imposed on the problem. This is because one would need to keep the intermediate result in step **(2)** which is proportional to the input string (i.e. with at least O(*N*) space complexity), which in no case could satisfy the constraint (use O(1) space to modify the string).

### ***Another Divide-and-Conquer Solution***

Looking closer at the constraint imposed by the problem, if we put it into the context of recursion, we could interpret it as not having additional space consumption between two consecutive recursive calls, i.e. we should divide the problem into independent subproblems.

So one of the ideas about how to divide the problem would be reducing the input string at each step into two components: 1). the leading and trailing characters. 2). the remaining substring without the leading and trailing characters. We then can solve the two components independently from each other.

Following the above idea, we could come up the algorithm as follows:

1. Take the leading and trailing characters from the input string, i.e. str[0] and str[n-1].
2. Swap the leading and trailing characters in place.
3. Call the function recursively to reverse the remaining substring, i.e. reverseString(str[1...n-2]).

Note that you can actually swap the order of steps (2) and (3), since they are independent tasks. Yet, it is better to keep them in this order, since this way we can use the optimization called [tail recursion](https://en.wikipedia.org/wiki/Tail_call). We'll shed more light on tail recursion in later chapters.

Here is an implementation of the above algorithm.

|  |
| --- |
| class Solution {  public void reverseString(char[] s) {  helper(0, s.length - 1, s);  }  private void helper(int start, int end, char [] s) {  if (start >= end) {  return;  }  // swap between the first and the last elements.  char tmp = s[start];  s[start] = s[end];  s[end] = tmp;    helper(start + 1, end - 1, s);  }  } |

Given the input string ["h", "e", "l", "l", "o"], we illustrate how it can be divided and solved:

Diagram

Description automatically generated

As one can see, we only need a constant memory in each recursive call in order to swap the leading and trailing characters. As a result, it meets the constraint of the problem.

**Recursion Function**

For a problem, if there exists a recursive solution, we can follow the guidelines below to implement it.

For instance, we define the problem as the function *F*(*X*) to implement, where *X* is the input of the function which also defines the scope of the problem.

Then, in the function *F*(*X*), we will:

1. Break the problem down into smaller scopes, such as *x*0​∈*X*,*x*1​∈*X*,...,*xn*​∈*X*;
2. Call function *F*(*x*0​),*F*(*x*1​),...,*F*(*xn*​) ***recursively*** to solve the subproblems of *X*;
3. Finally, process the results from the recursive function calls to solve the problem corresponding to *X*.

### ***Example***

To showcase the above guidelines, we give another example on how to solve a problem recursively.

Given a linked list, swap every two adjacent nodes and return its head.

e.g.  for a list 1-> 2 -> 3 -> 4, one should return the head of list as 2 -> 1 -> 4 -> 3.

We define the function to implement as swap(head), where the input parameter head refers to the head of a linked list. The function should return the head of the new linked list that has any adjacent nodes swapped.

Following the guidelines we lay out above, we can implement the function as follows:

1. First, we swap the first two nodes in the list, i.e. head and head.next;
2. Then, we call the function self as swap(head.next.next) to swap the rest of the list following the first two nodes.
3. Finally, we attach the returned head of the sub-list in step (2) with the two nodes swapped in step (1) to form a new linked list.

As an exercise, you can try to implement the solution using the steps we provided above. Click on "Swap Nodes in Pairs" to try to implement the solution yourself.

**Swap Nodes in Pairs**

Given a linked list, swap every two adjacent nodes and return its head.

**Example 1:**

Diagram

Description automatically generated

**Input:** head = [1,2,3,4]

**Output:** [2,1,4,3]

**Example 2:**

**Input:** head = []

**Output:** []

**Example 3:**

**Input:** head = [1]

**Output:** [1]

**Constraints:**

* The number of nodes in the list is in the range [0, 100].
* 0 <= Node.val <= 100

**Follow up:** Can you solve the problem without modifying the values in the list's nodes? (i.e., Only nodes themselves may be changed.)

## Solution

#### **Approach 1: Recursive Approach**

**Intuition**

The problem doesn't ask for entire reversal of linked list. It's rather asking us to swap every two adjacent nodes of a linked list starting at the very first node.

Diagram

Description automatically generated

The basic intuition is to reach to the end of the linked list in steps of two using recursion.

A screenshot of a computer

Description automatically generated with medium confidence

and while back tracking the nodes can be swapped.

Diagram

Description automatically generated

In every function call we take out two nodes which would be swapped and the remaining nodes are passed to the next recursive call. The reason we are adopting a recursive approach here is because a sub-list of the original list would still be a linked list and hence, it would adapt to our recursive strategy. Assuming the recursion would return the swapped remaining list of nodes, we just swap the current two nodes and attach the remaining list we get from recursion to these two swapped pairs.

Diagram

Description automatically generated

**Algorithm**

1. Start the recursion with head node of the original linked list.
2. Every recursion call is responsible for swapping a pair of nodes. Let's represent the two nodes to be swapped by firstNode and secondNode.
3. Next recursion is made by calling the function with head of the next pair of nodes. This call would swap the next two nodes and make further recursive calls if there are nodes left in the linked list.
4. Once we get the pointer to the remaining swapped list from the recursion call, we can swap the firstNode and secondNode i.e. the nodes in the current recursive call and then return the pointer to the secondNode since it will be the new head after swapping.

Diagram

Description automatically generated

1. Once all the pairs are swapped in the backtracking step, we would eventually be returning the pointer to the head of the now swapped list. This head will essentially be the second node in the original linked list.

|  |
| --- |
| /\*\*  \* Definition for singly-linked list.  \* public class ListNode {  \* int val;  \* ListNode next;  \* ListNode(int x) { val = x; }  \* }  \*/  class Solution {  public ListNode swapPairs(ListNode head) {  // If the list has no node or has only one node left.  if ((head == null) || (head.next == null)) {  return head;  }  // Nodes to be swapped  ListNode firstNode = head;  ListNode secondNode = head.next;  // Swapping  firstNode.next = swapPairs(secondNode.next);  secondNode.next = firstNode;  // Now the head is the second node  return secondNode;  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*N*) where *N* is the size of the linked list.
* Space Complexity: *O*(*N*) stack space utilized for recursion.

#### **Approach 2: Iterative Approach**

**Intuition**

The concept here is similar to the recursive approach. We break the linked list into pairs by jumping in steps of two. The only difference is, unlike recursion, we swap the nodes on the go. After swapping a pair of nodes, say A and B, we need to link the node B to the node that was right before A. To establish this linkage we save the previous node of node A in prevNode.

Diagram

Description automatically generated

**Algorithm**

1. We iterate the linked list with jumps in steps of two.
2. Swap the pair of nodes as we go, before we jump to the next pair. Let's represent the two nodes to be swapped by firstNode and secondNode.

A picture containing diagram

Description automatically generated

1. Swap the two nodes. The swap step is
2. firstNode.next = secondNode.next
3. secondNode.next = firstNode

Diagram

Description automatically generated

1. We also need to assign the prevNode's next to the head of the swapped pair. This step would ensure the currently swapped pair is linked correctly to the end of the previously swapped list.
2. prevNode.next = secondNode

Diagram

Description automatically generated

This is an iterative step, so the nodes are swapped on the go and attached to the previously swapped list. And in the end we get the final swapped list.

|  |
| --- |
| /\*\*  \* Definition for singly-linked list.  \* public class ListNode {  \* int val;  \* ListNode next;  \* ListNode(int x) { val = x; }  \* }  \*/  class Solution {  public ListNode swapPairs(ListNode head) {  // Dummy node acts as the prevNode for the head node  // of the list and hence stores pointer to the head node.  ListNode dummy = new ListNode(-1);  dummy.next = head;  ListNode prevNode = dummy;  while ((head != null) && (head.next != null)) {  // Nodes to be swapped  ListNode firstNode = head;  ListNode secondNode = head.next;  // Swapping  prevNode.next = secondNode;  firstNode.next = secondNode.next;  secondNode.next = firstNode;  // Reinitializing the head and prevNode for next swap  prevNode = firstNode;  head = firstNode.next; // jump  }  // Return the new head node.  return dummy.next;  }  } |

**Complexity Analysis**

* Time Complexity : *O*(*N*) where N is the size of the linked list.
* Space Complexity : *O*(1).

**Recurrence Relation**

In the previous chapter, we learned the basic concept of recursion.

There are two important things that you need to figure out before implementing a recursion function:base case and recurrence relation.

In this chapter, we will:

* Go through a detailed example on how to define the base case and recurrence relation;
* Then, we will have some exercises for you to practice with.

**Recurrence Relation**

There are two important things that one needs to figure out before implementing a recursive function:

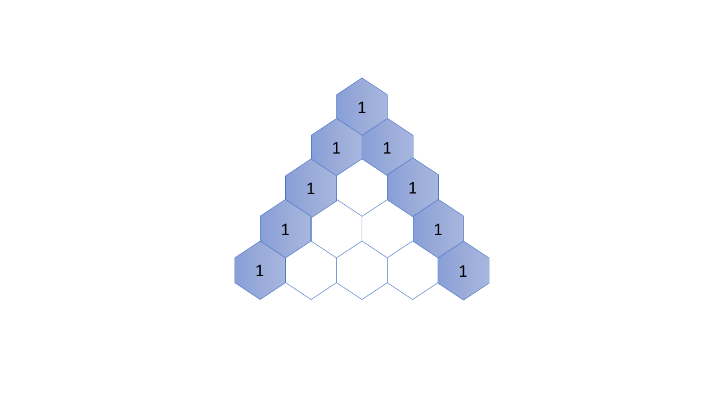
* recurrence relation: the relationship between the result of a problem and the result of its subproblems.
* base case: the case where one can compute the answer directly without any further recursion calls. Sometimes, the base cases are also called *bottom cases*, since they are often the cases where the problem has been reduced to the minimal scale, *i.e.* the bottom, if we consider that dividing the problem into subproblems is in a top-down manner.

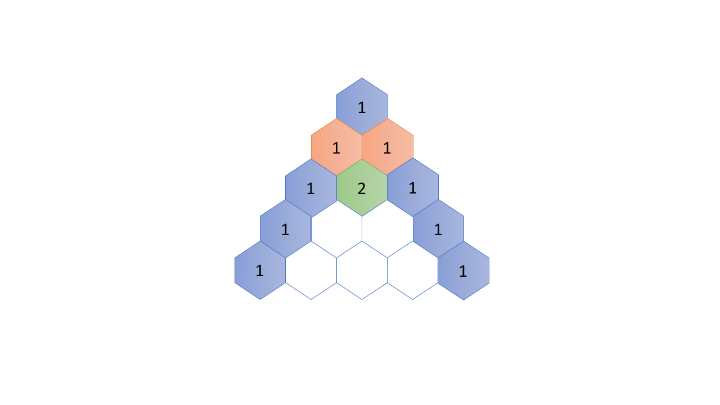
Once we figure out the above two elements, to implement a recursive function we simply call the function itself according to the recurrence relation until we reach the base case.

To explain the above points, let's look at a classic problem, Pascal's Triangle:

Pascal's triangle are a series of numbers arranged in the shape of triangle. In Pascal's triangle, the leftmost and the rightmost numbers of each row are always 1. For the rest, each number is the sum of the two numbers directly above it in the previous row.

Here's the illustration of the Pascal's Triangle with 5 rows:





A picture containing vector graphics

Description automatically generated

Given the above definition, one is asked to generate the Pascal's Triangle up to a certain number of rows.

### **Recurrence Relation**

Let's start with the recurrence relation within the Pascal's Triangle.

First of all, we define a function *f*(*i*,*j*) which returns the number in the Pascal's Triangle in the i-th row and j-th column.

We then can represent the recurrence relation with the following formula:

*f*(*i*,*j*)=*f*(*i*−1,*j*−1)+*f*(*i*−1,*j*)

### **Base Case**

As one can see, the leftmost and rightmost numbers of each row are the base cases in this problem, which are always equal to 1.

As a result, we can define the base case as follows:

*f*(*i*,*j*)=1*where j*=1 *or j*=*i*

### **Demo**

As one can see, once we define the recurrence relation and the base case, it becomes much more intuitive to implement the recursive function, especially when we formulate these two elements in terms of mathematical formulas.

Here is an example of how we can apply the formula to recursively calculate *f*(5,3), i.e. the 3rd number in the 5th row of the Pascal Triangle:

A picture containing vector graphics

Description automatically generated

A picture containing vector graphics

Description automatically generated

A picture containing vector graphics

Description automatically generated

A picture containing bubble chart

Description automatically generated

A picture containing bubble chart

Description automatically generated

A picture containing bubble chart

Description automatically generated

A picture containing bubble chart

Description automatically generated

A picture containing bubble chart

Description automatically generated

A picture containing vector graphics

Description automatically generated

A picture containing vector graphics

Description automatically generated

A picture containing vector graphics

Description automatically generated

A picture containing vector graphics

Description automatically generated

A picture containing chart

Description automatically generated

A picture containing vector graphics

Description automatically generated

A picture containing vector graphics

Description automatically generated

A picture containing vector graphics

Description automatically generated

A picture containing vector graphics

Description automatically generated

Starting from *f*(5,3), we can break it down as *f*(5,3)=*f*(4,2)+*f*(4,3), we then call *f*(4,2) and *f*(4,3) recursively:

* For the call of f(4, 2)*f*(4,2), we could extend it further until we reach the base cases, as follows:

(4,2)=*f*(3,1)+*f*(3,2)=*f*(3,1)+(*f*(2,1)+*f*(2,2))=1+(1+1)=3

* For the call of *f*(4,3), similarly we break it down as:

(4,3)=*f*(3,2)+*f*(3,3)=(*f*(2,1)+*f*(2,2))+*f*(3,3)=(1+1)+1=3

* Finally we combine the results of the above subproblems:

*f*(5,3)=*f*(4,2)+*f*(4,3)=3+3=6

### **Next**

In the above example, you might have noticed that the recursive solution can incur some duplicate calculations, i.e. we compute the same intermediate numbers repeatedly in order to obtain numbers in the last row. For instance, in order to obtain the result for the number *f*(5,3), we calculate the number *f*(3,2) twice both in the calls of *f*(4,2) and *f*(4,3).

We will discuss how to avoid these duplicate calculations in the next chapter of this Explore card.

Following this article, you will find exercises for problems related to Pascal's Triangle.

**Reverse Linked List**

Reverse a singly linked list.

**Example:**

**Input:** 1->2->3->4->5->NULL

**Output:** 5->4->3->2->1->NULL

**Follow up:**

A linked list can be reversed either iteratively or recursively. Could you implement both?

## Solution

#### **Approach #1 (Iterative) [Accepted]**

Assume that we have linked list 1 → 2 → 3 → Ø, we would like to change it to Ø ← 1 ← 2 ← 3.

While you are traversing the list, change the current node's next pointer to point to its previous element. Since a node does not have reference to its previous node, you must store its previous element beforehand. You also need another pointer to store the next node before changing the reference. Do not forget to return the new head reference at the end!

|  |
| --- |
| public ListNode reverseList(ListNode head) {  ListNode prev = null;  ListNode curr = head;  while (curr != null) {  ListNode nextTemp = curr.next;  curr.next = prev;  prev = curr;  curr = nextTemp;  }  return prev;  } |

**Complexity analysis**

* Time complexity : *O*(*n*). Assume that *n* is the list's length, the time complexity is *O*(*n*).
* Space complexity : *O*(1).

#### **Approach #2 (Recursive) [Accepted]**

The recursive version is slightly trickier and the key is to work backwards. Assume that the rest of the list had already been reversed, now how do I reverse the front part? Let's assume the list is: n1 → … → nk-1 → nk → nk+1 → … → nm → Ø

Assume from node nk+1 to nm had been reversed and you are at node nk.

n1 → … → nk-1 → **nk** → nk+1 ← … ← nm

We want nk+1’s next node to point to nk.

So,

nk.next.next = nk;

Be very careful that n1's next must point to Ø. If you forget about this, your linked list has a cycle in it. This bug could be caught if you test your code with a linked list of size 2.

|  |
| --- |
| public ListNode reverseList(ListNode head) {  if (head == null || head.next == null) return head;  ListNode p = reverseList(head.next);  head.next.next = head;  head.next = null;  return p;  } |

**Complexity analysis**

* Time complexity : *O*(*n*). Assume that *n* is the list's length, the time complexity is *O*(*n*).
* Space complexity : *O*(*n*). The extra space comes from implicit stack space due to recursion. The recursion could go up to *n* levels deep.

**Search in a Binary Search Tree**

You are given the root of a binary search tree (BST) and an integer val.

Find the node in the BST that the node's value equals val and return the subtree rooted with that node. If such a node does not exist, return null.

**Example 1:**

Shape

Description automatically generated with medium confidence

**Input:** root = [4,2,7,1,3], val = 2

**Output:** [2,1,3]

**Example 2:**

Shape, arrow

Description automatically generated

**Input:** root = [4,2,7,1,3], val = 5

**Output:** []

**Constraints:**

* The number of nodes in the tree is in the range [1, 5000].
* 1 <= Node.val <= 107
* root is a binary search tree.
* 1 <= val <= 107

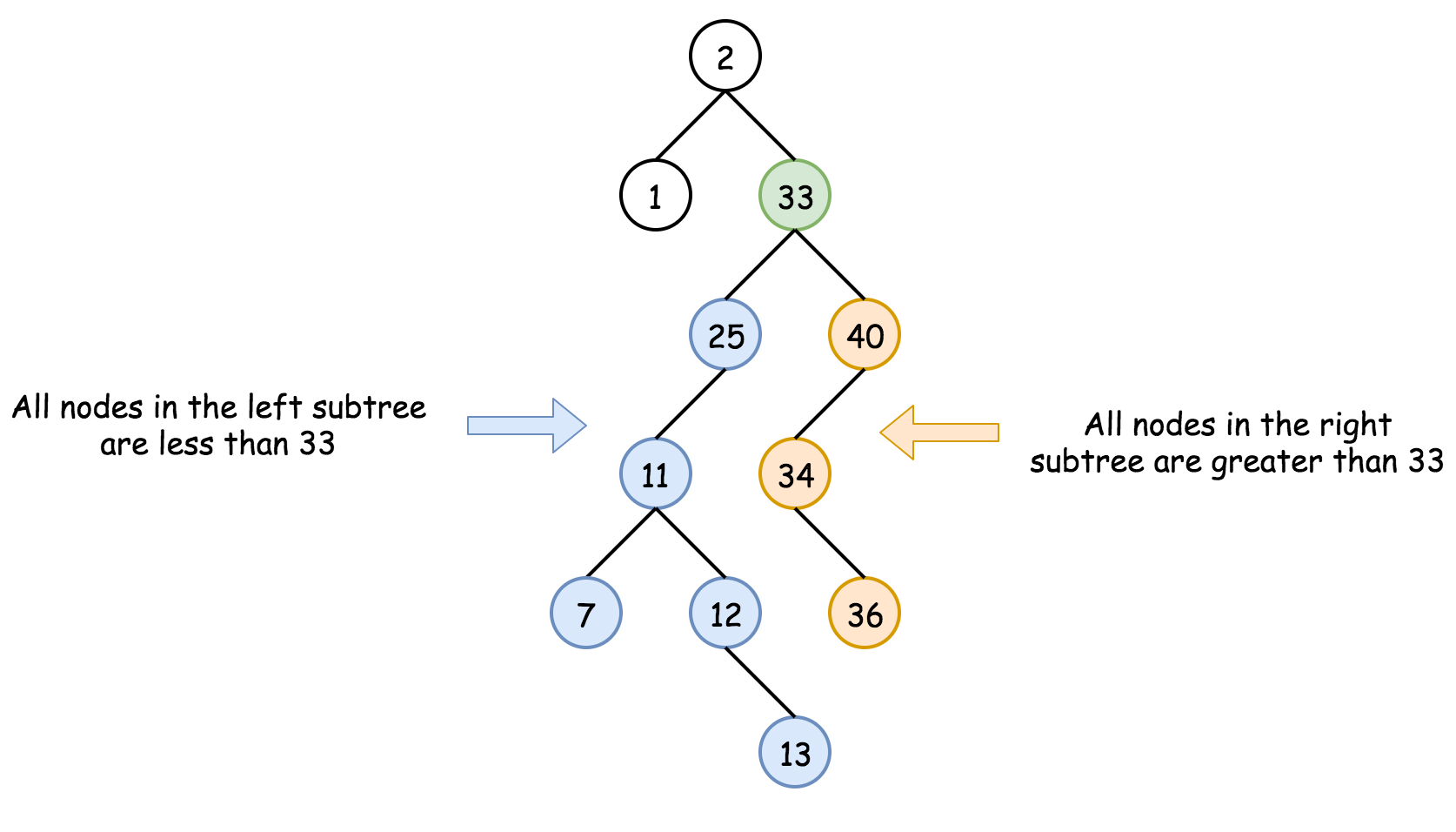
## Solution

#### **Binary Search Tree.**

Binary Search Tree is a binary tree where the key in each node

* is greater than any key stored in the left sub-tree,
* and less than any key stored in the right sub-tree.

Here is an example:



Such data structure provides the following operations in a logarithmic time:

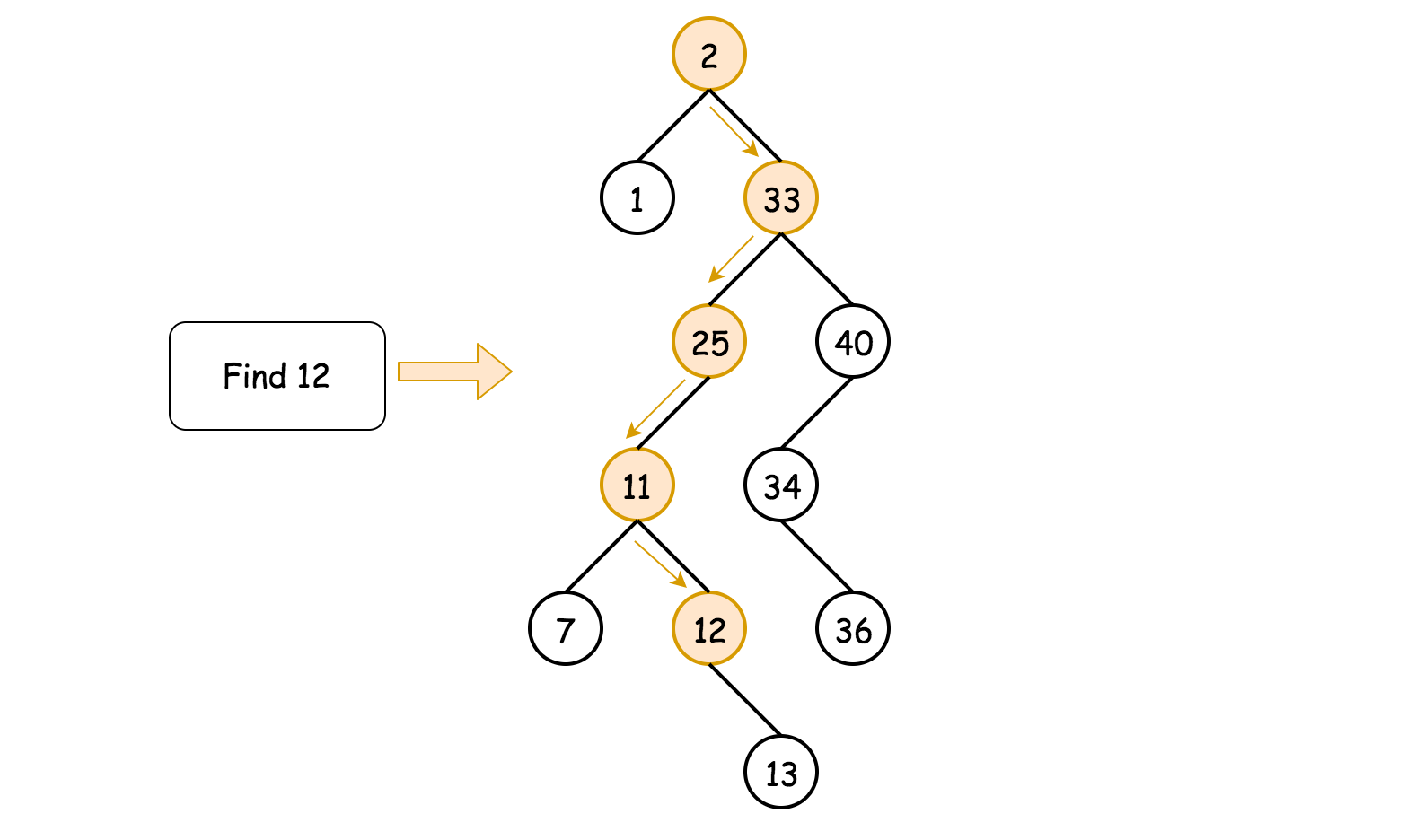
* Search.
* [Insert](https://leetcode.com/articles/insert-into-a-bst/).
* [Delete](https://leetcode.com/articles/delete-node-in-a-bst/).

#### **Approach 1: Recursion**

**Algorithm**

The recursion implementation is very straightforward:

* If the tree is empty root == null or the value to find is here val == root.val - return root.
* If val < root.val - go to search into the left subtree searchBST(root.left, val).
* If val > root.val - go to search into the right subtree searchBST(root.right, val).
* Return root.



**Implementation**

|  |
| --- |
| class Solution {  public TreeNode searchBST(TreeNode root, int val) {  if (root == null || val == root.val) return root;  return val < root.val ? searchBST(root.left, val) : searchBST(root.right, val);  }  } |

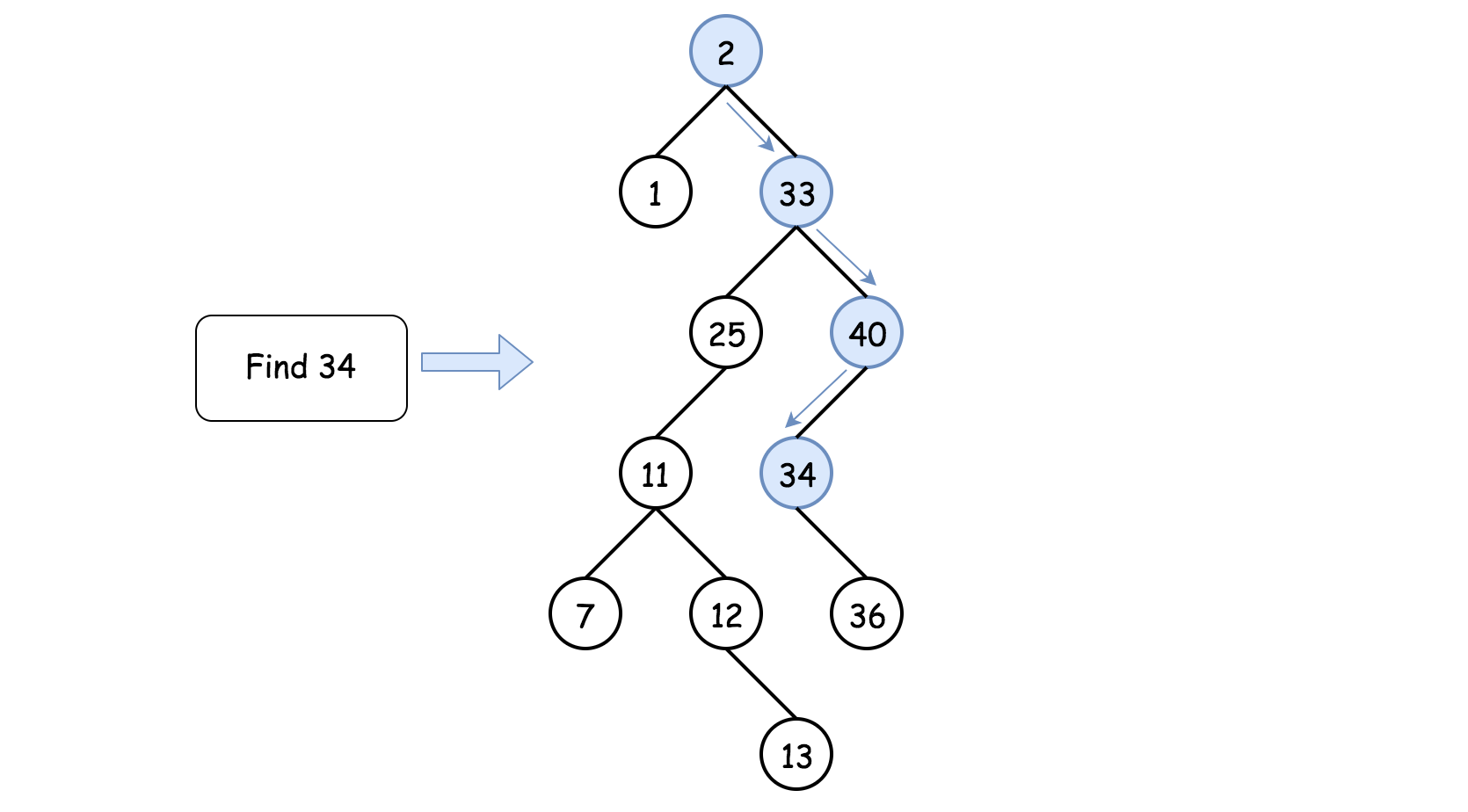
**Complexity Analysis**

* Time complexity : O(*H*), where *H* is a tree height. That results in O(log*N*) in the average case, and O(*N*) in the worst case.
* Space complexity : O(*H*) to keep the recursion stack, i.e. O(log*N*) in the average case, and O(*N*) in the worst case.

#### **Approach 2: Iteration**

To reduce the space complexity, one could convert recursive approach into the iterative one:

* While the tree is not empty root != null and the value to find is not here val != root.val:
  + If val < root.val - go to search into the left subtree root = root.left.
  + If val > root.val - go to search into the right subtree root = root.right.
* Return root.



**Implementation**

|  |
| --- |
| class Solution {  public TreeNode searchBST(TreeNode root, int val) {  while (root != null && val != root.val)  root = val < root.val ? root.left : root.right;  return root;  }  } |

**Complexity Analysis**

* Time complexity : O(*H*), where *H* is a tree height. That results in O(log*N*) in the average case, and O(*N*) in the worst case.
* Space complexity : O(1) since it's a constant space solution.

**Pascal's Triangle II**

Given an integer rowIndex, return the rowIndexth row of the Pascal's triangle.

Notice that the row index starts from **0**.

  
In Pascal's triangle, each number is the sum of the two numbers directly above it.

**Follow up:**

Could you optimize your algorithm to use only *O*(*k*) extra space?

**Example 1:**

**Input:** rowIndex = 3

**Output:** [1,3,3,1]

**Example 2:**

**Input:** rowIndex = 0

**Output:** [1]

**Example 3:**

**Input:** rowIndex = 1

**Output:** [1,1]

**Constraints:**

* 0 <= rowIndex <= 33

See Array part

## Memoization

In the previous chapter, we talked about the duplicate calculation problem in a recursion algorithm. In the best case, duplicate calculations would increase the time complexity of the algorithm, and in the worst case, it would lead to an infinite loop.

Therefore, in this chapter, we will:

* Start with an example and show you how duplicate calculations can occur;
* Show you how to avoid duplicate calculations using a technique called memoization.

**Duplicate Calculation in Recursion**

Recursion is often an intuitive and powerful way to implement an algorithm. However, it might bring some undesired penalty to the performance, *e.g.* duplicate calculations, if we do not use it wisely. For instance, at the end of the previous chapter, we have encountered the duplicate calculations problem in Pascal's Triangle, where some intermediate results are calculated multiple times.

In this article we will look closer into the duplicate calculations problem that could happen with recursion. We will then propose a common technique called memoization that can be used to avoid this problem.

To demonstrate another problem with duplicate calculations, let's look at an example that most people might be familiar with, the [Fibonacci number](https://en.wikipedia.org/wiki/Fibonacci_number). If we define the function F(n) to represent the Fibonacci number at the index of n, then you can derive the following recurrence relation:

F(n) = F(n - 1) + F(n - 2)

with the base cases:

F(0) = 0, F(1) = 1

Given the definition of a Fibonacci number, one can implement the function as follows:

|  |
| --- |
| public static int fibonacci(int n) {  if (n < 2) {  return n;  } else {  return fibonacci(n-1) + fibonacci(n-2);  }  } |

Now, if you would like to know the number of F(4), you can apply and extend the above formulas as follows:

F(4) = F(3) + F(2) = (F(2) + F(1)) + F(2)

As you can see, in order to obtain the result for F(4), we would need to calculate the number F(2) twice following the above deduction: the first time in the first extension of F(4) and the second time for the intermediate result F(3).

Here is the tree that shows all the duplicate calculations (grouped by colors) that occur during the calculation of F(4).

Diagram

Description automatically generated

### **Memoization**

To eliminate the duplicate calculation in the above case, as many of you would have figured out, one of the ideas would be to **store** the intermediate results in the cache so that we could reuse them later without re-calculation.

This idea is also known as memoization, which is a technique that is frequently used together with recursion.

[Memoization](https://en.wikipedia.org/wiki/Memoization) is an optimization technique used primarily to **speed up** computer programs by **storing** the results of expensive function calls and returning the cached result when the same inputs occur again. (Source: wikipedia)

Back to our Fibonacci function F(n). We could use a hash table to keep track of the result of each F(n) with n as the key. The hash table serves as a cache that saves us from duplicate calculations. The memoization technique is a good example that demonstrates how one can reduce compute time in exchange for some additional space.

For the sake of comparison, we provide the implementation of Fibonacci number solution with memoization below.

As an exercise, you could try to make memoization more general and non-intrusive, i.e. applying memoization without changing the original function. (Hint: one can refer to a design pattern called **decorator**).

|  |
| --- |
| import java.util.HashMap;  public class Main {  HashMap<Integer, Integer> cache = new HashMap<Integer, Integer>();  private int fib(int N) {  if (cache.containsKey(N)) {  return cache.get(N);  }  int result;  if (N < 2) {  result = N;  } else {  result = fib(N-1) + fib(N-2);  }  // keep the result in cache.  cache.put(N, result);  return result;  }  } |

Following this article, we provide the [Fibonacci number problem](https://leetcode.com/explore/learn/card/recursion-i/255/recursion-memoization/1661/) and another classic problem called [climbing stairs](https://leetcode.com/explore/learn/card/recursion-i/255/recursion-memoization/1662/), which could be really fun and challenging to solve.

In the next chapter, we will dive a bit into the complexity analysis of recursion algorithms.

**Fibonacci Number**

The **Fibonacci numbers**, commonly denoted F(n) form a sequence, called the **Fibonacci sequence**, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

F(0) = 0, F(1) = 1

F(n) = F(n - 1) + F(n - 2), for n > 1.

Given n, calculate F(n).

**Example 1:**

**Input:** n = 2

**Output:** 1

**Explanation:** F(2) = F(1) + F(0) = 1 + 0 = 1.

**Example 2:**

**Input:** n = 3

**Output:** 2

**Explanation:** F(3) = F(2) + F(1) = 1 + 1 = 2.

**Example 3:**

**Input:** n = 4

**Output:** 3

**Explanation:** F(4) = F(3) + F(2) = 2 + 1 = 3.

**Constraints:**

* 0 <= n <= 30

## Solution

#### **Approach 1: Recursion**

**Intuition**

Use recursion to compute the Fibonacci number of a given integer.

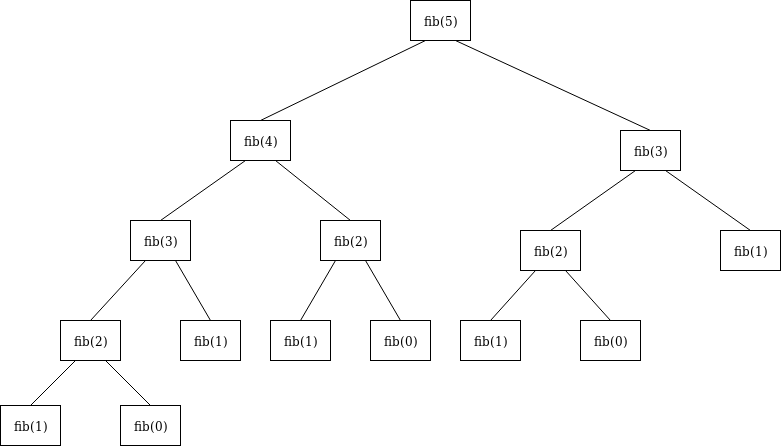


Figure 1. An example tree representing what *fib(5)* would look like

**Algorithm**

* Check if the provided input value, N, is less than or equal to 1. If true, return N.
* Otherwise, the function fib(int N) calls itself, with the result of the 2 previous numbers being added to each other, passed in as the argument. This is derived directly from the recurrence relation: *Fn*​=*Fn*−1​+*Fn*−2​
* Do this until all numbers have been computed, then return the resulting answer.

|  |
| --- |
| public class Solution {  public int fib(int N) {  if (N <= 1) {  return N;  }  return fib(N-1) + fib(N-2);  }  } |

**Complexity Analysis**

* Time complexity : O(2^N). This is the slowest way to solve the Fibonacci Sequence because it takes exponential time. The amount of operations needed, for each level of recursion, grows exponentially as the depth approaches N.
* Space complexity : O(N). We need space proportionate to N to account for the max size of the stack, in memory. This stack keeps track of the function calls to fib(N). This has the potential to be bad in cases that there isn't enough physical memory to handle the increasingly growing stack, leading to a StackOverflowError. The [Java docs](https://docs.oracle.com/javase/7/docs/api/java/lang/StackOverflowError.html) have a good explanation of this, describing it as an error that occurs because an application recurses too deeply.

#### **Approach 2: Bottom-Up Approach using Memoization**

**Intuition**

Improve upon the recursive option by using iteration, still solving for all of the sub-problems and returning the answer for N, using already computed Fibonacci values. In using a bottom-up approach, we can iteratively compute and store the values, only returning once we reach the result.

**Algorithm**

* If N is less than or equal to 1, return N
* Otherwise, iterate through N, storing each computed answer in an array along the way.
* Use this array as a reference to the 2 previous numbers to calculate the current Fibonacci number.
* Once we've reached the last number, return it's Fibonacci number.

|  |
| --- |
| class Solution {  public int fib(int N) {  if (N <= 1) {  return N;  }  return memoize(N);  }  public int memoize(int N) {  int[] cache = new int[N + 1];  cache[1] = 1;  for (int i = 2; i <= N; i++) {  cache[i] = cache[i-1] + cache[i-2];  }  return cache[N];  }  } |

**Complexity Analysis**

* Time complexity : *O*(*N*). Each number, starting at 2 up to and including N, is visited, computed and then stored for *O*(1) access later on.
* Space complexity : *O*(*N*). The size of the data structure is proportionate to N.

#### **Approach 3: Top-Down Approach using Memoization**

**Intuition**

Solve for all of the sub-problems, use memoization to store the pre-computed answers, then return the answer for N. We will leverage recursion, but in a smarter way by not repeating the work to calculate existing values.

**Algorithm**

* Check if N <= 1. If it is, return N.
* Call and return memoize(N)
* If N exists in the map, return the cached value for N
* Otherwise set the value of N, in our mapping, to the value of memoize(N-1) + memoize(N-2)

|  |
| --- |
| class Solution {  private Integer[] cache = new Integer[31];  public int fib(int N) {  if (N <= 1) {  return N;  }  cache[0] = 0;  cache[1] = 1;  return memoize(N);  }  public int memoize(int N) {  if (cache[N] != null) {  return cache[N];  }  cache[N] = memoize(N-1) + memoize(N-2);  return memoize(N);  }  } |

**Complexity Analysis**

* Time complexity : *O*(*N*). Each number, starting at 2 up to and including N, is visited, computed and then stored for *O*(1) access later on.
* Space complexity : *O*(*N*). The size of the stack in memory is proportionate to N.

#### **Approach 4: Iterative Top-Down Approach**

**Intuition**

Let's get rid of the need to use all of that space and instead use the minimum amount of space required. We can achieve *O*(1) space complexity by only storing the value of the two previous numbers and updating them as we iterate to N.

**Algorithm**

* Check if N <= 1, if it is then we should return N.
* Check if N == 2, if it is then we should return 1 since N is 2 and fib(2-1) + fib(2-2) equals 1 + 0 = 1.
* To use an iterative approach, we need at least 3 variables to store each state fib(N), fib(N-1) and fib(N-2).
* Preset the initial values:
  + Initialize current with 0.
  + Initialize prev1 with 1, since this will represent fib(N-1) when computing the current value.
  + Initialize prev2 with 1, since this will represent fib(N-2) when computing the current value.
* Iterate, incrementally by 1, all the way up to and including N. Starting at 3, since 0, 1 and 2 are pre-computed.
* Set the current value to fib(N-1) + fib(N-2) because that is the value we are currently computing.
* Set the prev2 value to fib(N-1).
* Set the prev1 value to current\_value.
* When we reach N+1, we will exit the loop and return the previously set current value.

|  |
| --- |
| class Solution {  public int fib(int N) {  if (N <= 1) {  return N;  }  if (N == 2) {  return 1;  }  int current = 0;  int prev1 = 1;  int prev2 = 1;  for (int i = 3; i <= N; i++) {  current = prev1 + prev2;  prev2 = prev1;  prev1 = current;  }  return current;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*N*). Each value from 2 to N will be visited at least once. The time it takes to do this is directly proportionate to N where N is the Fibonacci Number we are looking to compute.
* Space complexity : *O*(1). This requires 1 unit of Space for the integer N and 3 units of Space to store the computed values (curr, prev1 and prev2) for every loop iteration. The amount of Space doesn't change so this is constant Space complexity.

#### **Approach 5: Matrix Exponentiation**

**Intuition**

Use Matrix Exponentiation to get the Fibonacci number from the element at (0, 0) in the resultant matrix.

In order to do this we can rely on the matrix equation for the Fibonacci sequence, to find the Nth Fibonacci number: Text

Description automatically generated with low confidence

**Algorithm**

* Check if N is less than or equal to 1. If it is, return N.
* Use a recursive function, matrixPower, to calculate the power of a given matrix A. The power will be N-1, where N is the Nth Fibonacci number.
* The matrixPower function will be performed for N/2 of the Fibonacci numbers.
* Within matrixPower, call the multiply function to multiply 2 matrices.
* Once we finish doing the calculations, return A[0][0] to get the Nth Fibonacci number.

|  |
| --- |
| class Solution {  int fib(int N) {  if (N <= 1) {  return N;  }  int[][] A = new int[][]{{1, 1}, {1, 0}};  matrixPower(A, N-1);  return A[0][0];  }  void matrixPower(int[][] A, int N) {  if (N <= 1) {  return;  }  matrixPower(A, N/2);  multiply(A, A);  int[][] B = new int[][]{{1, 1}, {1, 0}};  if (N%2 != 0) {  multiply(A, B);  }  }  void multiply(int[][] A, int[][] B) {  int x = A[0][0] \* B[0][0] + A[0][1] \* B[1][0];  int y = A[0][0] \* B[0][1] + A[0][1] \* B[1][1];  int z = A[1][0] \* B[0][0] + A[1][1] \* B[1][0];  int w = A[1][0] \* B[0][1] + A[1][1] \* B[1][1];  A[0][0] = x;  A[0][1] = y;  A[1][0] = z;  A[1][1] = w;  }  } |

**Complexity Analysis**

* Time complexity : *O*(log*N*). By halving the N value in every matrixPower's call to itself, we are halving the work needed to be done.
* Space complexity : *O*(log*N*). The size of the stack in memory is proportionate to the function calls to matrixPower plus the memory used to account for the matrices which takes up constant space.

#### **Approach 6: Math**

**Intuition** Using the golden ratio, a.k.a Binet's forumula: 

Here's a [link](http://demonstrations.wolfram.com/GeneralizedFibonacciSequenceAndTheGoldenRatio/) to find out more about how the Fibonacci sequence and the golden ratio work.

We can derive the most efficient solution to this problem using only constant time and constant space!

**Algorithm**

* Use the golden ratio formula to calculate the Nth Fibonacci number.

|  |
| --- |
| class Solution {  public int fib(int N) {  double goldenRatio = (1 + Math.sqrt(5)) / 2;  return (int)Math.round(Math.pow(goldenRatio, N)/ Math.sqrt(5));  }  } |

**Complexity Analysis**

* Time complexity : *O*(1). Constant time complexity since we are using no loops or recursion and the time is based on the result of performing the calculation using Binet's formula.
* Space complexity : *O*(1). The space used is the space needed to create the variable to store the golden ratio formula.

**Climbing Stairs**

You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

**Example 1:**

**Input:** n = 2

**Output:** 2

**Explanation:** There are two ways to climb to the top.

1. 1 step + 1 step

2. 2 steps

**Example 2:**

**Input:** n = 3

**Output:** 3

**Explanation:** There are three ways to climb to the top.

1. 1 step + 1 step + 1 step

2. 1 step + 2 steps

3. 2 steps + 1 step

**Constraints:**

* 1 <= n <= 45

To reach nth step, what could have been your previous steps? (Think about the step sizes)

## Summary

You are climbing a stair case. It takes n steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

## Solution

#### **Approach 1: Brute Force**

**Algorithm**

In this brute force approach we take all possible step combinations i.e. 1 and 2, at every step. At every step we are calling the function *climbStairs* for step 1 and 2, and return the sum of returned values of both functions.

*climbStairs*(*i*,*n*)=(*i*+1,*n*)+*climbStairs*(*i*+2,*n*)

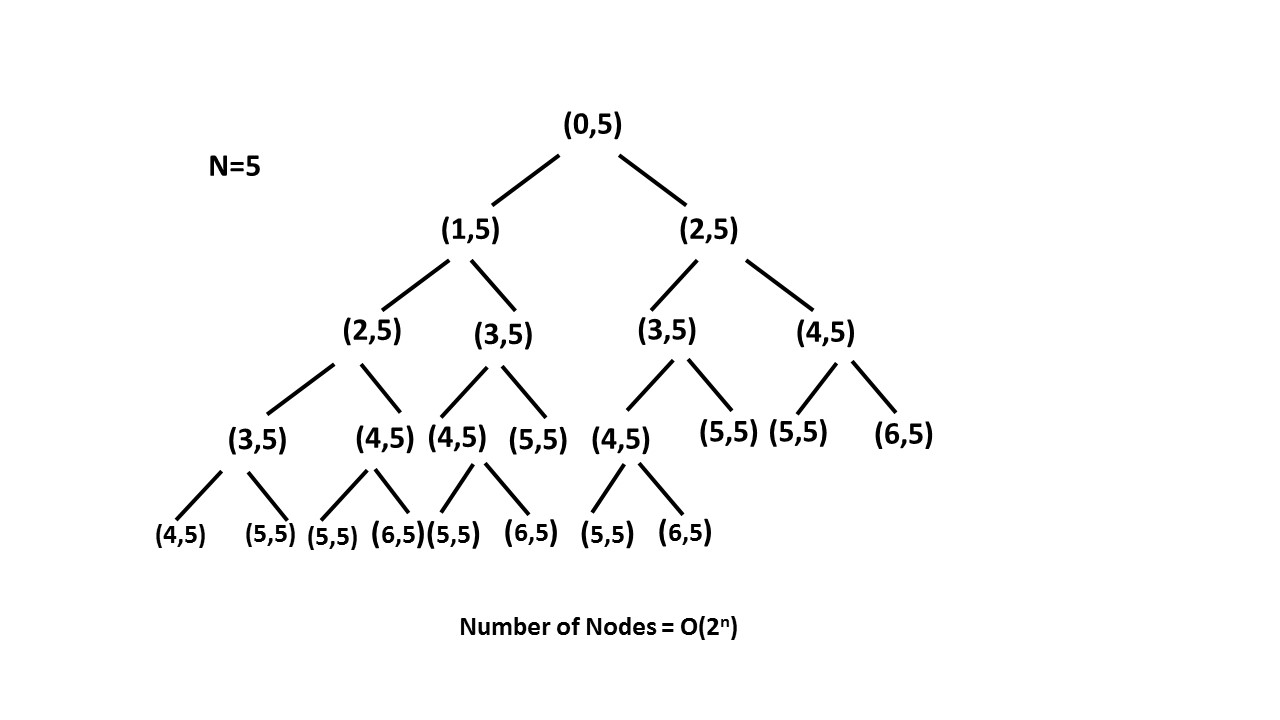
where *i* defines the current step and *n* defines the destination step.

|  |
| --- |
| public class Solution {  public int climbStairs(int n) {  return climb\_Stairs(0, n);  }  public int climb\_Stairs(int i, int n) {  if (i > n) {  return 0;  }  if (i == n) {  return 1;  }  return climb\_Stairs(i + 1, n) + climb\_Stairs(i + 2, n);  }  } |

**Complexity Analysis**

* Time complexity : O(2^n). Size of recursion tree will be 2^n.

Recursion tree for n=5 would be like this:



* Space complexity : *O*(*n*). The depth of the recursion tree can go upto *n*.

#### **Approach 2: Recursion with Memoization**

**Algorithm**

In the previous approach we are redundantly calculating the result for every step. Instead, we can store the result at each step in *memo* array and directly returning the result from the memo array whenever that function is called again.

In this way we are pruning recursion tree with the help of *memo* array and reducing the size of recursion tree upto *n*.

|  |
| --- |
| public class Solution {  public int climbStairs(int n) {  int memo[] = new int[n + 1];  return climb\_Stairs(0, n, memo);  }  public int climb\_Stairs(int i, int n, int memo[]) {  if (i > n) {  return 0;  }  if (i == n) {  return 1;  }  if (memo[i] > 0) {  return memo[i];  }  memo[i] = climb\_Stairs(i + 1, n, memo) + climb\_Stairs(i + 2, n, memo);  return memo[i];  }  } |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). Size of recursion tree can go upto n*n*.
* Space complexity : O(n)*O*(*n*). The depth of recursion tree can go upto n*n*.

#### **Approach 3: Dynamic Programming**

**Algorithm**

As we can see this problem can be broken into subproblems, and it contains the optimal substructure property i.e. its optimal solution can be constructed efficiently from optimal solutions of its subproblems, we can use dynamic programming to solve this problem.

One can reach *ith* step in one of the two ways:

1. Taking a single step from (*i*−1)*th* step.
2. Taking a step of 2 from (*i*−2)*th* step.

So, the total number of ways to reach *ith* is equal to sum of ways of reaching (*i*−1)*th* step and ways of reaching (*i*−2)*th* step.

Let *dp*[*i*] denotes the number of ways to reach on *ith* step:

*dp*[*i*]=*dp*[*i*−1]+*dp*[*i*−2]

Timeline

Description automatically generated

Timeline

Description automatically generated

Timeline

Description automatically generated

Timeline

Description automatically generated

Timeline

Description automatically generated

Timeline

Description automatically generated

Timeline

Description automatically generated

|  |
| --- |
| public class Solution {  public int climbStairs(int n) {  if (n == 1) {  return 1;  }  int[] dp = new int[n + 1];  dp[1] = 1;  dp[2] = 2;  for (int i = 3; i <= n; i++) {  dp[i] = dp[i - 1] + dp[i - 2];  }  return dp[n];  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*). Single loop upto *n*.
* Space complexity : *O*(*n*). *dp* array of size *n* is used.

#### **Approach 4: Fibonacci Number**

**Algorithm**

In the above approach we have used *dp* array where *dp*[*i*]=*dp*[*i*−1]+*dp*[*i*−2]. It can be easily analysed that *dp*[*i*] is nothing but *ith* fibonacci number.

*Fib*(*n*)=*Fib*(*n*−1)+*Fib*(*n*−2)

Now we just have to find *nth* number of the fibonacci series having 1 and 2 their first and second term respectively, i.e. *Fib*(1)=1 and *Fib*(2)=2.

|  |
| --- |
| public class Solution {  public int climbStairs(int n) {  if (n == 1) {  return 1;  }  int first = 1;  int second = 2;  for (int i = 3; i <= n; i++) {  int third = first + second;  first = second;  second = third;  }  return second;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*). Single loop upto *n* is required to calculate *nth* fibonacci number.
* Space complexity : *O*(1). Constant space is used.

## Complexity Analysis

In this chapter, we will talk about how to estimate the time and space complexity of recursion algorithms.

In particular, we will present you a useful technique called Tail Recursion, which can be applied to optimize the space complexity of some recursion problems, and more importantly to avoid the problem of stack overflow.

**Time Complexity - Recursion**

In this article, we will focus on how to calculate the time complexity for recursion algorithms.

Given a recursion algorithm, its time complexity O(*T*) is typically the product of **the number of recursion** **invocations** (denoted as *R*) and **the time complexity of calculation** (denoted as O(*s*)) that incurs along with each recursion call:

O(*T*)=*R*∗O(*s*)

Let's take a look at some examples below.

### ***Example***

As you might recall, in the problem of [printReverse](https://leetcode.com/explore/learn/card/recursion-i/250/principle-of-recursion/1439/" \t "_blank), we are asked to print the string in the reverse order. A recurrence relation to solve the problem can be expressed as follows:

printReverse(str) = printReverse(str[1...n]) + print(str[0])

where str[1...n] is the substring of the input string str, without the leading character str[0].

As you can see, the function would be recursively invoked n times, where n is the size of the input string. At the end of each recursion, we simply print the leading character, therefore the time complexity of this particular operation is constant, i.e. O(1).

To sum up, the overall time complexity of our recursive function printReverse(str) would be O(*printReverse*)=*n*∗O(1)=O(*n*).

### ***Execution Tree***

For recursive functions, it is rarely the case that the number of recursion calls happens to be linear to the size of input. For example, one might recall the example of [Fibonacci number](https://leetcode.com/explore/learn/card/recursion-i/255/recursion-memoization/1661/) that we discussed in the previous chapter, whose recurrence relation is defined as f(n) = f(n-1) + f(n-2). At first glance, it does not seem straightforward to calculate the number of recursion invocations during the execution of the Fibonacci function.

In this case, it is better resort to the ***execution tree***, which is a tree that is used to denote the execution flow of a recursive function in particular. Each node in the tree represents an invocation of the recursive function. Therefore, the total number of nodes in the tree corresponds to the number of recursion calls during the execution.

The execution tree of a recursive function would form an n-ary tree, with n as the number of times recursion appears in the recurrence relation. For instance, the execution of the Fibonacci function would form a ***binary tree***, as one can see from the following graph which shows the execution tree for the calculation of Fibonacci number f(4).

Diagram

Description automatically generated

In a full binary tree with n levels, the total number of nodes would be 2n - 1. Therefore, the upper bound (though not tight) for the number of recursion in f(n) would be 2n -1, as well. As a result, we can estimate that the time complexity for f(n) would be O(2n).

### ***Memoization***

In the previous chapter, we discussed the technique of memoization that is often applied to optimize the time complexity of recursion algorithms. By caching and reusing the intermediate results, memoization can greatly reduce the number of recursion calls, i.e. reducing the number of branches in the execution tree. One should take this reduction into account when analyzing the time complexity of recursion algorithms with memoization.

Let's get back to our example of Fibonacci number. With memoization, we save the result of Fibonacci number for each index n. We are assured that the calculation for each Fibonacci number would occur only once. And we know, from the recurrence relation, the Fibonacci number f(n) would depend on all n-1 precedent Fibonacci numbers. As a result, the recursion to calculate f(n) would be invoked n-1 times to calculate all the precedent numbers that it depends on.

Now, we can simply apply the formula we introduced in the beginning of this chapter to calculate the time complexity, which is O(1)∗*n*=O(*n*). Memoization not only optimizes the time complexity of algorithm, but also simplifies the calculation of time complexity.

In the next article, we will talk about how to evaluate the space complexity of recursion algorithms.

**Space Complexity – Recursion**In this article, we will talk about how to analyze the space complexity of a recursive algorithm.

There are mainly two parts of the space consumption that one should bear in mind when calculating the space complexity of a recursive algorithm: recursion related and non-recursion related space.

### ***Recursion Related Space***

The recursion related space refers to the memory cost that is incurred directly by the recursion, i.e. the stack to keep track of recursive function calls. In order to complete a typical function call, the system allocates some space in the stack to hold three important pieces of information:

1. The returning address of the function call. Once the function call is completed, the program must know where to return to, i.e. the line of code after the function call.
2. The parameters that are passed to the function call.
3. The local variables within the function call.

This space in the stack is the minimal cost that is incurred during a function call. However, once the function call is done, this space is freed.

For recursive algorithms, the function calls chain up successively until they reach a base case (a.k.a. bottom case). This implies that the space that is used for each function call is accumulated.

For a recursive algorithm, if there is no other memory consumption, then this recursion incurred space will be the space upper-bound of the algorithm.

For example, in the exercise of [printReverse](https://leetcode.com/explore/learn/card/recursion-i/250/principle-of-recursion/1439/" \t "_blank), we don't have extra memory usage outside the recursive call, since we simply print a character. For each recursive call, let's assume it can use space up to a constant value. And the recursive calls will chain up to n times, where n is the size of the input string. So the space complexity of this recursive algorithm is O(*n*).

To illustrate this, for a sequence of recursive calls f(x1) -> f(x2) -> f(x3), we show the sequence of execution steps along with the layout of the stack:

Diagram

Description automatically generated

A space in the stack will be allocated for f(x1) in order to call f(x2). Similarly in f(x2), the system will allocate another space for the call to f(x3). Finally in f(x3), we reach the base case, therefore there is no further recursive call within f(x3).

It is due to recursion-related space consumption that sometimes one might run into a situation called [stack overflow](https://en.wikipedia.org/wiki/Stack_overflow), where the stack allocated for a program reaches its maximum space limit and the program crashes. Therefore, when designing a recursive algorithm, one should carefully check if there is a possibility of stack overflow when the input scales up.

### ***Non-Recursion Related Space***

As suggested by the name, the non-recursion related space refers to the memory space that is not directly related to recursion, which typically includes the space (normally in heap) that is allocated for the global variables.

Recursion or not, you might need to store the input of the problem as global variables, before any subsequent function calls. And you might need to save the intermediate results from the recursive calls as well. The latter is also known as ***memoization*** as we saw in the previous chapters. For example, in the recursive algorithm with memoization to solve the Fibonacci number problem, we used a map to keep track of all intermediate Fibonacci numbers that occurred during the recursive calls. Therefore, in the space complexity analysis, we must take the space cost incurred by the memoization into consideration.

**Tail Recursion**

In the previous article, we talked about the implicit extra space incurred on the system stack due to recursion calls. However, you should learn to identify a special case of recursion called [tail recursion](https://en.wikipedia.org/wiki/Tail_call), which is **exempted** from this space overhead.

**Tail recursion** is a recursion where the recursive call is the final instruction in the recursion function. And there should be only **one** recursive call in the function.

We have already seen an example of tail recursion in the solution of [Reverse String](https://leetcode.com/explore/learn/card/recursion-i/250/principle-of-recursion/1679/). Here is another example that shows the difference between non-tail-recursion and tail-recursion. Notice that in the non-tail-recursion example there is an extra computation after the very last recursive call.

|  |
| --- |
| public class Main {    private static int helper\_non\_tail\_recursion(int start, int [] ls) {  if (start >= ls.length) {  return 0;  }  // not a tail recursion because it does some computation after the recursive call returned.  return ls[start] + helper\_non\_tail\_recursion(start+1, ls);  }  public static int sum\_non\_tail\_recursion(int [] ls) {  if (ls == null || ls.length == 0) {  return 0;  }  return helper\_non\_tail\_recursion(0, ls);  }  //---------------------------------------------  private static int helper\_tail\_recursion(int start, int [] ls, int acc) {  if (start >= ls.length) {  return acc;  }  // this is a tail recursion because the final instruction is the recursive call.  return helper\_tail\_recursion(start+1, ls, acc+ls[start]);  }    public static int sum\_tail\_recursion(int [] ls) {  if (ls == null || ls.length == 0) {  return 0;  }  return helper\_tail\_recursion(0, ls, 0);  }  } |

The benefit of having tail recursion is that it could avoid the accumulation of stack overheads during the recursive calls, since the system could reuse a fixed amount space in the stack for each recursive call.

For example, for the sequence of recursion calls f(x1) -> f(x2) -> f(x3), if the function f(x) is implemented as tail recursion, then here is the sequence of execution steps along with the layout of the stack:

Diagram

Description automatically generated

Note that in tail recursion, we know that as soon as we return from the recursive call we are going to immediately return as well, so we can skip the entire chain of recursive calls returning and return straight to the original caller. That means we don't need a call stack at all for all of the recursive calls, which saves us space.

For example, in step (1), a space in the stack would be allocated for f(x1) in order to call f(x2). Then in step (2), the function f(x2) would recursively call f(x3). However, instead of allocating new space on the stack, the system could simply reuse the space allocated earlier for this second recursion call. Finally, in the function f(x3), we reach the base case, and the function could simply return the result to the original caller without going back to the previous function calls.

A tail recursion function can be executed as non-tail-recursion functions, *i.e.* with piles of call stacks, without impact on the result. Often, the compiler recognizes tail recursion pattern, and optimizes its execution. However, not all programming languages support this optimization. For instance, C, C++ support the optimization of tail recursion functions. On the other hand, Java and Python do not support tail recursion optimization.

**Maximum Depth of Binary Tree**

Given the root of a binary tree, return its maximum depth.

A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

**Example 1:**

Diagram, shape

Description automatically generated

**Input:** root = [3,9,20,null,null,15,7]

**Output:** 3

**Example 2:**

**Input:** root = [1,null,2]

**Output:** 2

**Example 3:**

**Input:** root = []

**Output:** 0

**Example 4:**

**Input:** root = [0]

**Output:** 1

**Constraints:**

* The number of nodes in the tree is in the range [0, 104].
* -100 <= Node.val <= 100

## Solution

**Tree definition**

First of all, here is the definition of the TreeNode which we would use.

|  |
| --- |
| /\* Definition for a binary tree node. \*/  public class TreeNode {  int val;  TreeNode left;  TreeNode right;  TreeNode(int x) {  val = x;  }  } |

#### **Approach 1: Recursion**

**Intuition** By definition, the maximum depth of a binary tree is the maximum number of steps to reach a leaf node from the root node.

From the definition, an intuitive idea would be to traverse the tree and record the maximum depth during the traversal.

**Algorithm**

A picture containing text, clock, watch

Description automatically generated

A picture containing text, clock, watch

Description automatically generated

A picture containing text, watch, clock

Description automatically generated

A picture containing text, clock, watch

Description automatically generated

A picture containing text, clock, watch

Description automatically generated

A picture containing text, clock, watch

Description automatically generated

A picture containing text, clock, watch

Description automatically generated

A picture containing text, clock, watch

Description automatically generated

A picture containing text, clock, watch

Description automatically generated

A picture containing text, clock, watch

Description automatically generated

One could traverse the tree either by Depth-First Search (DFS) strategy or Breadth-First Search (BFS) strategy. For this problem, either way would do. Here we demonstrate a solution that is implemented with the **DFS** strategy and **recursion**.

|  |
| --- |
| class Solution {  public int maxDepth(TreeNode root) {  if (root == null) {  return 0;  } else {  int left\_height = maxDepth(root.left);  int right\_height = maxDepth(root.right);  return java.lang.Math.max(left\_height, right\_height) + 1;  }  }  } |

**Complexity analysis**

* Time complexity : we visit each node exactly once, thus the time complexity is O(*N*), where *N* is the number of nodes.
* Space complexity : in the worst case, the tree is completely unbalanced, e.g. each node has only left child node, the recursion call would occur *N* times (the height of the tree), therefore the storage to keep the call stack would be O(*N*). But in the best case (the tree is completely balanced), the height of the tree would be log(*N*). Therefore, the space complexity in this case would be O(log(*N*)).

#### **Approach 2: Tail Recursion + BFS**

One might have noticed that the above recursion solution is probably not the most optimal one in terms of the space complexity, and in the extreme case the overhead of call stack might even lead to stack overflow.

To address the issue, one can tweak the solution a bit to make it **tail recursion**, which is a specific form of recursion where the recursive call is the last action in the function.

The benefit of having tail recursion, is that for certain programming languages (e.g. C++) the compiler could optimize the memory allocation of call stack by reusing the same space for every recursive call, rather than creating the space for each one. As a result, one could obtain the constant space complexity O(1) for the overhead of the recursive calls.

Here is a sample solution. Note that the optimization of tail recursion is not supported by Python or Java.

|  |
| --- |
| class Solution {  private:  // The queue that contains the next nodes to visit,  // along with the level/depth that each node is located.  queue<pair<TreeNode\*, int>> next\_items;  int max\_depth = 0;    /\*\*  \* A tail recursion function to calculate the max depth  \* of the binary tree.  \*/  int next\_maxDepth() {    if (next\_items.size() == 0) {  return max\_depth;  }    auto next\_item = next\_items.front();  next\_items.pop();  auto next\_node = next\_item.first;  auto next\_level = next\_item.second + 1;    max\_depth = max(max\_depth, next\_level);  // Add the nodes to visit in the following recursive calls.  if (next\_node->left != NULL) {  next\_items.push(make\_pair(next\_node->left, next\_level));  }  if (next\_node->right != NULL) {  next\_items.push(make\_pair(next\_node->right, next\_level));  }    // The last action should be the ONLY recursive call  // in the tail-recursion function.  return next\_maxDepth();  }    public:  int maxDepth(TreeNode\* root) {  if (root == NULL) return 0;    // clear the previous queue.  std::queue<pair<TreeNode\*, int>> empty;  std::swap(next\_items, empty);  max\_depth = 0;    // push the root node into the queue to kick off the next visit.  next\_items.push(make\_pair(root, 0));    return next\_maxDepth();  }  }; |

**Complexity analysis**

* Time complexity : O(*N*), still we visit each node once and only once.
* Space complexity :  i.e. the maximum number of nodes at the same level (the number of leaf nodes in a full binary tree), since we traverse the tree in the **BFS** manner.

As one can see, this probably is not the best example to apply the tail recursion technique. Because though we did gain the constant space complexity for the recursive calls, we pay the price of O(*N*) complexity to maintain the state information for recursive calls. This defeats the purpose of applying tail recursion.

However, we would like to stress on the point that tail recursion is a useful form of recursion that could eliminate the space overhead incurred by the recursive function calls.

Note: a function cannot be tail recursion if there are multiple occurrences of recursive calls in the function, even if the last action is the recursive call. Because the system has to maintain the function call stack for the sub-function calls that occur within the same function.

#### **Approach 3: Iteration**

**Intuition**

We could also convert the above recursion into iteration, with the help of the stack data structure. Similar with the behaviors of the function call stack, the stack data structure follows the pattern of FILO (First-In-Last-Out), i.e. the last element that is added to a stack would come out first.

With the help of the stack data structure, one could mimic the behaviors of function call stack that is involved in the recursion, to convert a recursive function to a function with iteration.

**Algorithm**

The idea is to keep the next nodes to visit in a stack. Due to the FILO behavior of stack, one would get the order of visit same as the one in recursion.

We start from a stack which contains the root node and the corresponding depth which is 1. Then we proceed to the iterations: pop the current node out of the stack and push the child nodes. The depth is updated at each step.

|  |
| --- |
| class Solution {  public int maxDepth(TreeNode root) {  LinkedList<TreeNode> stack = new LinkedList<>();  LinkedList<Integer> depths = new LinkedList<>();  if (root == null) return 0;  stack.add(root);  depths.add(1);  int depth = 0, current\_depth = 0;  while(!stack.isEmpty()) {  root = stack.pollLast();  current\_depth = depths.pollLast();  if (root != null) {  depth = Math.max(depth, current\_depth);  stack.add(root.left);  stack.add(root.right);  depths.add(current\_depth + 1);  depths.add(current\_depth + 1);  }  }  return depth;  }  }; |

**Complexity analysis**

* Time complexity : O(*N*).
* Space complexity : in the worst case, the tree is completely unbalanced, e.g. each node has only left child node, the recursion call would occur *N* times (the height of the tree), therefore the storage to keep the call stack would be O(*N*). But in the average case (the tree is balanced), the height of the tree would be log(*N*). Therefore, the space complexity in this case would be O(log(*N*)).

**Pow(x, n)**

Implement [pow(x, n)](http://www.cplusplus.com/reference/valarray/pow/), which calculates x raised to the power n (i.e. xn).

**Example 1:**

**Input:** x = 2.00000, n = 10

**Output:** 1024.00000

**Example 2:**

**Input:** x = 2.10000, n = 3

**Output:** 9.26100

**Example 3:**

**Input:** x = 2.00000, n = -2

**Output:** 0.25000

**Explanation:** 2-2 = 1/22 = 1/4 = 0.25

**Constraints:**

* -100.0 < x < 100.0
* -231 <= n <= 231-1
* -104 <= xn <= 104

#### **Approach 1: Brute Force**

**Intuition**

Just simulate the process, multiply x for n times.

If *n*<0, we can substitute *x*,*n* with 1/*x*​,−*n* to make sure *n*≥0. This restriction can simplify our further discussion.

But we need to take care of the corner cases, especially different range limits for negative and positive integers.

**Algorithm**

We can use a straightforward loop to compute the result.

|  |
| --- |
| class Solution {  public double myPow(double x, int n) {  long N = n;  if (N < 0) {  x = 1 / x;  N = -N;  }  double ans = 1;  for (long i = 0; i < N; i++)  ans = ans \* x;  return ans;  }  }; |

**Complexity Analysis**

* Time complexity : *O*(*n*). We will multiply x for n times.
* Space complexity : *O*(1). We only need one variable to store the final product of x.

#### **Approach 2: Fast Power Algorithm Recursive**

**Intuition**

Assuming we have got the result of x ^ n, how can we get x ^ {2 \* n} ? Obviously we do not need to multiply x for another n times. Using the formula (x ^ n) ^ 2 = x ^ {2 \* n}, we can get x ^ {2 \* n} at the cost of only one computation. Using this optimization, we can reduce the time complexity of our algorithm.

**Algorithm**

Assume we have got the result of x ^ {n / 2}, and now we want to get the result of x ^ n. Let A be result of x ^ {n / 2}, we can talk about x ^ n based on the parity of n respectively. If n is even, we can use the formula (x ^ n) ^ 2 = x ^ {2 \* n}(*xn*) to get x ^ n = A \* A. If n is odd, then A \* A = x ^ {n - 1}. Intuitively, We need to multiply another x*x* to the result, so x ^ n = A \* A \* x. This approach can be easily implemented using recursion. We call this method "**Fast Power**", because we only need at most *O*(log*n*) computations to get x ^ n.

|  |
| --- |
| class Solution {  private double fastPow(double x, long n) {  if (n == 0) {  return 1.0;  }  double half = fastPow(x, n / 2);  if (n % 2 == 0) {  return half \* half;  } else {  return half \* half \* x;  }  }  public double myPow(double x, int n) {  long N = n;  if (N < 0) {  x = 1 / x;  N = -N;  }  return fastPow(x, N);  }  }; |

**Complexity Analysis**

* Time complexity : *O*(log*n*). Each time we apply the formula (x ^ n) ^ 2 = x ^ {2 \* n}(*xn*), *n* is reduced by half. Thus we need at most *O*(log*n*) computations to get the result.
* Space complexity : *O*(log*n*). For each computation, we need to store the result of x ^ {n / 2}. We need to do the computation for *O*(log*n*) times, so the space complexity is *O*(log*n*).

#### **Approach 3: Fast Power Algorithm Iterative**

**Intuition**

Using the formula x ^ {a + b} = x ^ a \* x ^ b*xa*+*b*=*xa*∗*xb*, we can write n as a sum of positive integers, n = A picture containing graphical user interface

Description automatically generated. If we can get the result of x(Bi)​ quickly, the total time for computing x ^ n. will be reduced.

**Algorithm**

Text, letter

Description automatically generated

Using fast power recursively or iteratively are actually taking different paths towards the same goal. For more information about fast power algorithm, you can visit its wiki[[1]](https://leetcode.com/problems/powx-n/solution/#fn1).

|  |
| --- |
| class Solution {  public double myPow(double x, int n) {  long N = n;  if (N < 0) {  x = 1 / x;  N = -N;  }  double ans = 1;  double current\_product = x;  for (long i = N; i > 0; i /= 2) {  if ((i % 2) == 1) {  ans = ans \* current\_product;  }  current\_product = current\_product \* current\_product;  }  return ans;  }  }; |

**Complexity Analysis**

* Time complexity : *O*(log*n*). For each bit of n 's binary representation, we will at most multiply once. So the total time complexity is *O*(log*n*).
* Space complexity : *O*(1). We only need two variables for the current product and the final result of x.

## Conclusion

In the previous chapters, we went through the concept and the principles of recursion.

As a reminder, here is the general workflow to solve a recursion problem:

1. Define the recursion function;
2. Write down the recurrence relation and base case;
3. Use memoization to eliminate the duplicate calculation problem, if it exists.
4. Whenever possible, implement the function as tail recursion, to optimize the space complexity.

In this chapter, we conclude on the recursion algorithms and provide you with more tips on how to solve some problems with recursion.

**Conclusion - Recursion I**

Now, you might be convinced that recursion is indeed a powerful technique that allows us to solve many problems in an elegant and efficient way. But still, it is no silver bullet. Not every problem can be solved with recursion, due to the time or space constraints. And recursion itself might come with some undesired side effects such as stack overflow.

In this chapter we would like to share a few more tips on how to better apply recursion to solve problems in the real world.

When in doubt, write down the **recurrence relationship**.

Sometimes, at a first glance it is not evident that a recursion algorithm can be applied to solve a problem. However, it is always helpful to deduct some relationships with the help of mathematical formulas, since the recurrence nature in recursion is quite close to the mathematics that we are familiar with. Often, they can clarify the ideas and uncover the hidden recurrence relationship. Within this chapter, you can find a fun example named [Unique Binary Search Trees II](https://leetcode.com/explore/featured/card/recursion-i/253/conclusion/2384/), which can be solved by recursion, with the help of mathematical formulas.

Whenever possible, apply **memoization**.

When drafting a recursion algorithm, one could start with the most naive strategy. Sometimes, one might end up with the situation where there might be duplicate calculation during the recursion, e.g. Fibonacci numbers. In this case, you can try to apply the memoization technique, which stores the intermediate results in cache for later reuse. Memoization could greatly improve the time complexity with a bit of trade on space complexity, since it could avoid the expensive duplicate calculation.

When stack overflows, **tail recursion** might come to help.

There are often several ways to implement an algorithm with recursion. Tail recursion is a specific form of recursion that we could implement. Different from the memoization technique, tail recursion could optimize the space complexity of the algorithm, by eliminating the stack overhead incurred by recursion. More importantly, with tail recursion, one could avoid the problem of stack overflow that comes often with recursion. Another advantage about tail recursion is that often times it is easier to read and understand, compared to non-tail-recursion. Because there is no post-call dependency in tail recursion (i.e. the recursive call is the final action in the function), unlike non-tail-recursion. Therefore, whenever possible, one should strive to apply the tail recursion.

A picture containing shape

Description automatically generated

### ***Next***

Now, with everything that you've learned so far about recursion, you can carry on to solve many more problems on LeetCode! We provide a few more classic exercises in this chapter for you to practise your newly acquired hammer — recursion !

Enjoy the exercises! If you have any questions, you can always go back to review previous chapters, or simply make a post in the [Discuss forum](https://leetcode.com/discuss/explore/recursion-i) at the end of this Explore card.

**Merge Two Sorted Lists**

Merge two sorted linked lists and return it as a **sorted** list. The list should be made by splicing together the nodes of the first two lists.

**Example 1:**

A picture containing text, clipart, clock

Description automatically generated

**Input:** l1 = [1,2,4], l2 = [1,3,4]

**Output:** [1,1,2,3,4,4]

**Example 2:**

**Input:** l1 = [], l2 = []

**Output:** []

**Example 3:**

**Input:** l1 = [], l2 = [0]

**Output:** [0]

**Constraints:**

* The number of nodes in both lists is in the range [0, 50].
* -100 <= Node.val <= 100
* Both l1 and l2 are sorted in **non-decreasing** order.

See LinkedList

**K-th Symbol in Grammar**

On the first row, we write a 0. Now in every subsequent row, we look at the previous row and replace each occurrence of 0 with 01, and each occurrence of 1 with 10.

Given row N and index K, return the K-th indexed symbol in row N. (The values of K are 1-indexed.) (1 indexed).

**Examples:**

**Input:** N = 1, K = 1

**Output:** 0

**Input:** N = 2, K = 1

**Output:** 0

**Input:** N = 2, K = 2

**Output:** 1

**Input:** N = 4, K = 5

**Output:** 1

**Explanation:**

row 1: 0

row 2: 01

row 3: 0110

row 4: 01101001

**Note:**

1. N will be an integer in the range [1, 30].
2. K will be an integer in the range [1, 2^(N-1)].

   Hide Hint #1

Try to represent the current (N, K) in terms of some (N-1, prevK). What is prevK ?

## Solution

#### **Approach 1: Brute Force**

**Intuition and Algorithm**

We'll make each row exactly as directed in the problem statement. We only need to remember the last row.

Unfortunately, the strings could have length around 1 billion, as they double on each row, so this approach is not efficient enough.

|  |
| --- |
| class Solution {  public int kthGrammar(int N, int K) {  int[] lastrow = new int[1 << N];  for (int i = 1; i < N; ++i) {  for (int j = (1 << (i-1)) - 1; j >= 0; --j) {  lastrow[2\*j] = lastrow[j];  lastrow[2\*j+1] = 1 - lastrow[j];  }  }  return lastrow[K-1];  }  } |

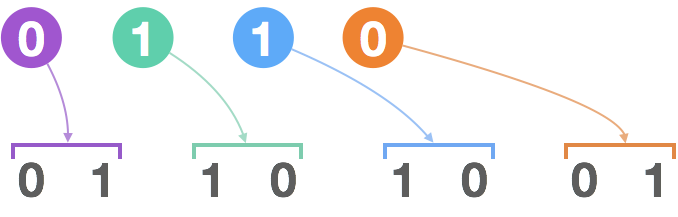
**Complexity Analysis**

* Time Complexity: O(2^N). We parse rows with lengths 2^0 + 2^1 + … + 2^{N-1}.
* Space Complexity: O(2^N), the length of the lastrow.

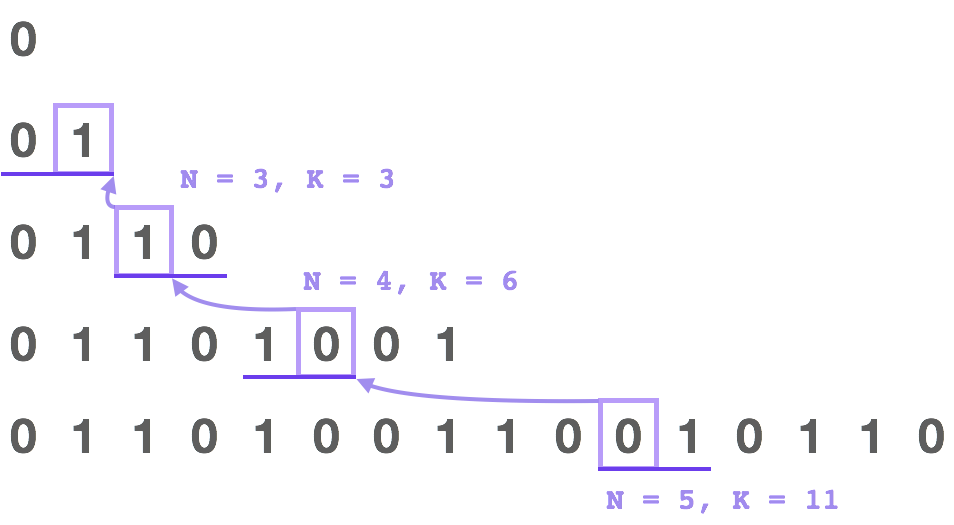
#### **Approach 2: Recursion (Parent Variant)**

**Intuition and Algorithm**

Since each row is made only using information from the previous row, let's try to write the answer in terms of bits from the previous row.



In particular, if we write say "0110" which generates "01101001", then the first "0" generates the first "01" in the next row; the next digit "1" generates the next "10", the next "1" generates the next "10", and the last "0" generates the last "01".



In general, the Kth digit's parent is going to be (K+1) / 2. If the parent is 0, then the digit will be the same as 1 - (K%2). If the parent is 1, the digit will be the opposite, ie. K%2.

|  |
| --- |
| class Solution {  public int kthGrammar(int N, int K) {  if (N == 1) return 0;  return (~K & 1) ^ kthGrammar(N-1, (K+1)/2);  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*N*). It takes *N*−1 steps to find the answer.
* Space Complexity: *O*(1).

#### **Approach 3: Recursion (Flip Variant)**

**Intuition and Algorithm**

As in Approach #2, we could try to write the bit in terms of it's previous bit.

When writing a few rows of the sequence, we notice a pattern: the second half is always the first half "flipped": namely, that '0' becomes '1' and '1' becomes '0'.

We can prove this assertion by induction. The key idea is if a string Xgenerates Y, then a flipped string X' generates Y'.

This leads to the following algorithm idea: if K is in the second half, then we could put K -= (1 << N-2) so that it is in the first half, and flip the final answer.

|  |
| --- |
| class Solution {  public int kthGrammar(int N, int K) {  if (N == 1) return 0;  if (K <= 1 << N-2)  return kthGrammar(N-1, K);  return kthGrammar(N-1, K - (1 << N-2)) ^ 1;  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*N*). It takes *N*−1 steps to find the answer.
* Space Complexity: *O*(1).

#### **Approach 4: Binary Count**

**Intuition and Algorithm**

As in Approach #3, the second half of every row is the first half flipped.

When the indexes K are written in binary (now indexing from zero), indexes of the second half of a row are ones with the first bit set to 1.

This means when applying the algorithm in Approach #3 virtually, the number of times we will flip the final answer is just the number of 1s in the binary representation of K-1.

|  |
| --- |
| class Solution {  public int kthGrammar(int N, int K) {  return Integer.bitCount(K - 1) % 2;  }  } |

**Complexity Analysis**

* Time Complexity: *O*(log*N*), the number of binary bits in N. If log*N* is taken to be bounded, this can be considered to be *O*(1).
* Space Complexity: *O*(1). (In Python, bin(X) creates a string of length *O*(log*X*), which could be avoided.)

**Unique Binary Search Trees II**

Given an integer n, return all the structurally unique ***BST'***s (binary search trees), which has exactly n nodes of unique values from 1 to n. Return the answer in **any order**.

**Example 1:**

A picture containing clipart, clock

Description automatically generated

**Input:** n = 3

**Output:** [[1,null,2,null,3],[1,null,3,2],[2,1,3],[3,1,null,null,2],[3,2,null,1]]

**Example 2:**

**Input:** n = 1

**Output:** [[1]]

**Constraints:**

* 1 <= n <= 8

## Solution

**Tree definition**

First of all, here is the definition of the TreeNode which we would use.

|  |
| --- |
| // Definition for a binary tree node.  public class TreeNode {  int val;  TreeNode left;  TreeNode right;  TreeNode(int x) {  val = x;  }  } |

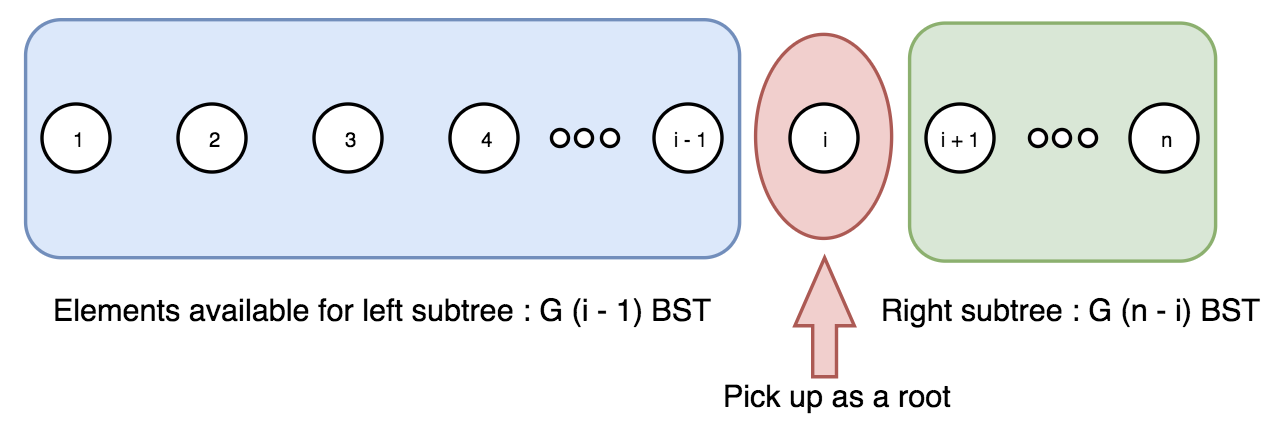
#### **Approach 1: Recursion**

First of all let's count how many trees do we have to construct. As you could check in [this article](https://leetcode.com/articles/unique-binary-search-trees/), the number of possible BST is actually a [Catalan number](https://en.wikipedia.org/wiki/Catalan_number).

Let's follow the logic from the above article, this time not to count but to actually construct the trees.

**Algorithm**

Let's pick up number i out of the sequence 1 ..n and use it as the root of the current tree. Then there are i - 1 elements available for the construction of the left subtree and n - i elements available for the right subtree. As we [already discussed](https://leetcode.com/articles/unique-binary-search-trees/) that results in G(i - 1) different left subtrees and G(n - i) different right subtrees, where G is a Catalan number.



Now let's repeat the step above for the sequence 1 ... i - 1 to construct all left subtrees, and then for the sequence i + 1 ... n to construct all right subtrees.

This way we have a root i and two lists for the possible left and right subtrees. The final step is to loop over both lists to link left and right subtrees to the root.

|  |
| --- |
| class Solution {  public LinkedList<TreeNode> generate\_trees(int start, int end) {  LinkedList<TreeNode> all\_trees = new LinkedList<TreeNode>();  if (start > end) {  all\_trees.add(null);  return all\_trees;  }  // pick up a root  for (int i = start; i <= end; i++) {  // all possible left subtrees if i is choosen to be a root  LinkedList<TreeNode> left\_trees = generate\_trees(start, i - 1);  // all possible right subtrees if i is choosen to be a root  LinkedList<TreeNode> right\_trees = generate\_trees(i + 1, end);  // connect left and right trees to the root i  for (TreeNode l : left\_trees) {  for (TreeNode r : right\_trees) {  TreeNode current\_tree = new TreeNode(i);  current\_tree.left = l;  current\_tree.right = r;  all\_trees.add(current\_tree);  }  }  }  return all\_trees;  }  public List<TreeNode> generateTrees(int n) {  if (n == 0) {  return new LinkedList<TreeNode>();  }  return generate\_trees(1, n);  }  } |

Text

Description automatically generated