Recursion

Recursion is like a mirror that reflects itself. Imagine you're looking into two mirrors facing each other, and you see an infinite tunnel of reflections. In C++, recursion is a concept where a function calls itself to solve a problem, creating a similar infinite loop of function calls. Following are steps in which are followed while writing a recursive function.

How recursion works:

Following steps are usually followed to solve a problem using recursion

1. **Base Case**: Every recursive function needs a base case, like a stopping point. It is the simplest case(s) where the problem can be solved directly or a condition that tells the function when to stop calling itself. Without it, the function would keep calling itself endlessly, like the endless reflections in the mirrors.
2. **Breaking down the Problem**: In a recursive function, you break down a big problem into smaller, more manageable parts. These smaller parts should be similar to the original problem but simpler. It's like solving a big puzzle by solving smaller pieces of it.
3. **Calling Itself**: Here comes the magic! The function calls itself with those smaller parts of the problem. It's like solving each piece of the puzzle one by one.
4. **Getting Results**: As the function keeps calling itself with smaller problems, it eventually reaches the base case. When that happens, it starts collecting results from all the smaller problems it solved.
5. **Combining Results**: Now, the function combines all the results it gathered from solving smaller problems. It's like putting the puzzle pieces back together to solve the whole puzzle.
6. **Returning the Answer**: Finally, the function returns the answer, which is often the solution to the original big problem. This is like getting the final picture of the puzzle after putting all the pieces together.

Recursion is a powerful programming technique where a function calls itself to solve a problem by breaking it into smaller, more manageable sub-problems. Examples include calculating factorial, Fibonacci numbers, or traversing complex data structures like trees.

But be careful! If you forget the base case, your program will run forever (just like those endless reflections in the mirrors). So, when using recursion, make sure you have a clear stopping point, break down the problem into smaller parts, and let the recursion do its work!. And, keep in mind - Understanding recursion requires practice :).

Recursive Cases:

Every recursive function must have at least two cases: the recursive case and the base case. The base case is a small problem that we know how to solve and is the case that causes the recursion to end.

Base Case:

At this point, the problem is small enough that we can answer it without breaking it down into the smaller sub-problems. For example, you know what the answer of 1**!** (Factorial) is. The base case in recursion is like the stop sign for a recursive function. It tells the function when to stop calling itself and start giving answers.

Recursive Case:

The recursive case is where a function calls itself with a smaller or simpler version of the problem. It's the heart of recursion, breaking down complex tasks into smaller steps. This process continues until the base case is reached, at which point the recursion reverses and combines the results to solve the original problem. It's like solving a big puzzle by solving smaller pieces first.

Types of recursion:

Recursion in computer science can be categorized into several types based on how a function calls itself and how the recursive calls are structured. Here are some common types of recursion:

1. **Direct:** This is like a person talking to themselves. In programming, it means a function calls itself directly. Imagine you have a recipe, and one of the steps tells you to "repeat the whole recipe." That's direct recursion.
2. **Indirect:** Think of it as a conversation between two friends. They keep talking to each other in a circle. In programming, it's when two or more functions call each other in a circular manner. It's like Friend A says, "Hey, Friend B, what do you think?" And Friend B replies, "I'm not sure, let's ask Friend A." They keep asking each other.
3. **Linear:** This is like falling dominoes, one after the other. In programming, it means a function makes only one recursive call. It's a straightforward sequence, like opening a box, finding another box inside, and repeating until you find what you want.
4. **Tree:** Imagine a tree with branches splitting into more branches. In programming, a function makes multiple recursive calls. It's like saying, "If you have one box, open it. If you find more boxes, open each of those too."
5. **Tail:** Picture a list of instructions where the last step is to do the same thing again. In programming, a tail recursion is when the recursive call is the last operation in the function. It's like saying, "After you finish reading this book, read it again."
6. **Non-Tail:** This is when the recursive call isn't the last thing you do. In programming, there are some other operations after the recursive call. It's like saying, "Read a few pages of this book, then read a different book, and finally come back to this one."
7. **Nested:** Think of it as a Russian nesting doll. A doll inside a doll inside a doll, and so on. In programming, a function calls itself with an altered argument. It's like saying, "If you can solve a small problem, try solving a smaller one inside it, and keep going until you reach the tiniest problem you can solve."

These concepts help programmers use recursion effectively to solve different kinds of problems. Each type has its own unique way of working, like different tools in a toolbox.

Recursive Trace

Before we move towards some logical recursive functions, let's practice recursive trace a little bit with the following snippets. Get your pen and paper and try to figure out what would be the output of the following functions with the help of recursion tree. Don't do it on compiler at first but later for verification.

1. Assume the value of N to be 4.

```
3   void f1(int N)  // dual recursion
4   {
5       if (N <= 1)
6       {
7           return;
8       }
9       cout << N;
10      f1(N - 1);
11      f1(N - 1);
12  }
```

Figure 1 (Recursion Example 1)

Explanation:
In the given recursive function, each function invocation triggers two subsequent calls to itself within its body. In line 9, the value of 'N' passed as an argument when the function is called is printed. In line 10, the function recurs by invoking itself with an argument that is one less than its original parameter value. The same recursive process is followed in line 11.

2. Assume the value of N to be 4.

```
3   void f2(int N)  // dual recursion
4   {
5       if (N <= 1)
6       {
7           return;
8       }
9
10      f2(N - 1);
11      cout << N;
12      f2(N - 1);
13  }
```

Figure 2 (Recursion Example 2)

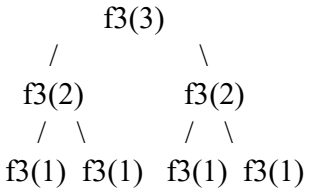
Explanation:
At line number 11 value of parameter of recursive function will be displayed while at line 10 and 12 function calls itself recursively for value of N less than its current parameter value.

3. Assume the value of N to be 3.

```
3   void f3(int N)  // dual recursion
4   {
5       if (N <= 1)
6       {
7           return;
8       }
9
10      f3(N - 1);
11      f3(N - 1);
12      cout << N;
13  }
```

Figure 3 (Recursion Example 3)

Recursive Tree of function f3()
Here's a recursive tree representation of the given function **f3(int n)**: for function call F3(3)



In this recursive tree:

- Each node represents a function call with the value of **n**.
- The left child of each node represents the first recursive call **f3(n - 1)**.
- The right child of each node represents the second recursive call **f3(n - 1)**.
- The leaf nodes (with **n <= 1**) are the base cases, and they don't make further recursive calls.

The function calls progress from the root node (f3(3)) down to the base cases (f3(1)), and then the values of **n** are printed.

Time and Space Complexity of function f3
Now, let's analyze the computational complexity of this corrected function. The function **f3** is a recursive function that calls itself twice with **n - 1** in each recursive call. It then prints the value of **n**. This function essentially creates a binary tree structure, where each node branches into two child nodes until **n** reaches a base case of **n <= 1**.

Time Complexity:

- In each recursive call, the function reduces the value of **n** by 1 (**n - 1**).
- The function continues making two recursive calls until **n** reaches the base case (**n <= 1**).
- The work done in each recursive call includes two more recursive calls and a print statement (**count << n**).

To analyze the time complexity, let's consider the number of calls made at each level of recursion:

- Level 0: 1 call (**f3(n)**)
- Level 1: 2 calls (**f3(n - 1)** and **f3(n - 1)**)
- Level 2: 4 calls (**f3(n - 2)** for each of the 2 calls at level 1)
- Level 3: 8 calls (**f3(n - 3)** for each of the 4 calls at level 2)
- ...

In general, at each level **i**, there are 2^i calls. The recursion continues until **n** reaches the base case. Therefore, the total number of calls made is 2^0 + 2^1 + 2^2 + ... + 2^(n-1), which is a geometric progression. The sum of a geometric progression is 2^n - 1, so the time complexity of this function is O(2^n), where 'n' is the input value.

Space Complexity:

- The space complexity is determined by the depth of the recursion stack because each recursive call adds a new frame to the call stack.
- In the worst case, the recursion depth is 'n' because the function makes 'n' recursive calls before reaching the base case.

The space complexity of the function is O(n) because the space used on the call stack is proportional to the value of 'n'. It's linear with respect to the input 'n'.

In summary, the time complexity of the **f3** function is O(2^n), and the space complexity is O(n), where 'n' is the input value. This function has an exponential time complexity due to the binary branching of recursive calls and a linear space complexity due to the recursion stack. It's highly inefficient for large values of 'n'.

Examples of Recursive Functions

Go through the following algorithms and you will get some idea of how to write the recursive function.

1. Print numbers up to n in ascending order - non-tail recursion

```
1   #include <iostream>
2   using namespace std;
3   void printNumbers(int n)
4   {
5       //Base case :Stop the recursion when n reaches 0
6       if (n == 0)
7           return;
8       // Recursive call to print numbers from 1 to n-1
9       printNumbers(n - 1);
10
11      // Print the current number
12
13      cout << n << " ";
14  }
15  int main()
16  {
17      printNumbers(10);
18  }
```

Figure 4 (Recursion Example 4)

Explanation:

In line 6 there is base case. When number reaches zero function should return to its calling statement and control will move to the next statement. Otherwise in line 9 we have called recursive function again for value of N-1 and so on for next recursive call. In line 13 value of n is also displayed that will be passed to that function at calling time as argument.

Output:

```
Select Microsoft Visual Studio Debug Console
1 2 3 4 5 6 7 8 9 10
```

Figure 5 (Output)

2. Print numbers in descending order – tail recursion

```cpp
#include <iostream>
using namespace std;
void printNumbersDescending(int n)
{
    //Base case :Stop the recursion when n reaches 0
    if (n == 0)
        return;

    // Print the current number

    cout << n << " ";

    // Recursive call to print numbers from n to 1
    printNumbersDescending(n - 1);
}
int main()
{
    printNumbersDescending(10);
}
```

Figure 6 (Descending Print)

Explanation:

In line 6 there is base case. When number reaches zero function should return to its calling statement and control will move to the next statement. In line 11 value of n is also displayed that will be passed to that function at calling time as argument. In line 14 we have called recursive function again for value of N-1 and so on for next recursive call.

Output

```
Select Microsoft Visual Studio Debug Console
10 9 8 7 6 5 4 3 2 1
```

Figure 7 (Output)

3. Factorial of a number - linear, non-tail recursion

```cpp
#include <iostream>
using namespace std;
int factorial(int n)
{
    if (n == 0 || n == 1)   // base case
        return 1;  // we know the answers for the base cases
    int sub_problem = factorial(n - 1); // recursive case
    int answer = n * sub_problem;  // combine results
    return answer;
}
```

Figure 8 (Factorial)

Explanation:

This is an example of linear, none-tail recursion. If we call this function for n = 4

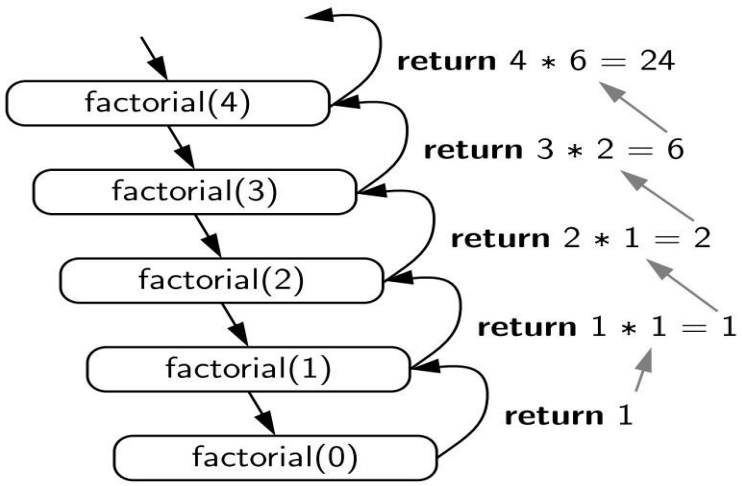


Figure 9 (Factorial Recursion Trace)

4. Nth Fibonacci number:

The Fibonacci sequence is a sequence of numbers in which each number is the sum of the two preceding ones, usually starting with 0 and 1. In mathematical terms, it can be defined by the recurrence relation:

$F(0) = 0$ $F(1) = 1$ $F(n) = F(n-1) + F(n-2)$ for $n > 1$

Here's the initial part of the Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... Fibonacci numbers have a wide range of applications in mathematics, science, art, and technology. They are a fundamental concept with both theoretical and practical significance.

```
1  #include <iostream>
2  using namespace std;
3  int fib(int n)
4  {
5      if (n <= 1)
6          return n;
7      return fib(n - 1) + fib(n - 2);
8  }
```

Figure 10 (Fibonacci Series)

Explanation:

The given function, **int Fibonacci(int n)**, is a recursive function that calculates the nth Fibonacci number. Here's how the function works:

1. It takes an integer **n** as input, which represents the position in the Fibonacci sequence for which you want to calculate the value.
2. It has three cases:
    - If **n** is less than or equal to 0, it returns 0. This is the base case for the Fibonacci sequence because the sequence starts with 0.
    - If **n** is equal to 1, it returns 1. This is also a base case because the second number in the Fibonacci sequence is 1.
    - For all other values of **n**, it uses recursion to calculate the nth Fibonacci number. It does this by making two recursive calls:

- **Fibonacci(n - 1)** calculates the (n-1)th Fibonacci number.
- **Fibonacci(n - 2)** calculates the (n-2)nd Fibonacci number.
- It then adds the results of these two recursive calls together to get the nth Fibonacci number.

3. The recursion continues until it reaches one of the base cases (n <= 0 or n == 1), at which point it starts returning values back up the recursive calls to calculate the final result.

Here's how the function works for a few examples:

- **Fibonacci(0)** returns 0 because it's the first number in the sequence.
- **Fibonacci(1)** returns 1 because it's the second number in the sequence.
- **Fibonacci(2)** returns **Fibonacci(1) + Fibonacci(0)**, which is 1 + 0 = 1.
- **Fibonacci(3)** returns **Fibonacci(2) + Fibonacci(1)**, which is 1 + 1 = 2.
- **Fibonacci(4)** returns **Fibonacci(3) + Fibonacci(2)**, which is 2 + 1 = 3.
- And so on...

While this recursive function accurately calculates Fibonacci numbers, it's not the most efficient way to do so for large values of **n** because it involves redundant calculations. Dynamic programming techniques or iterative approaches are more efficient for large Fibonacci numbers.

Recursive Tree of Fibonacci function:
Here is the visual explanation if we call Fibonacci function with argument n = 4.

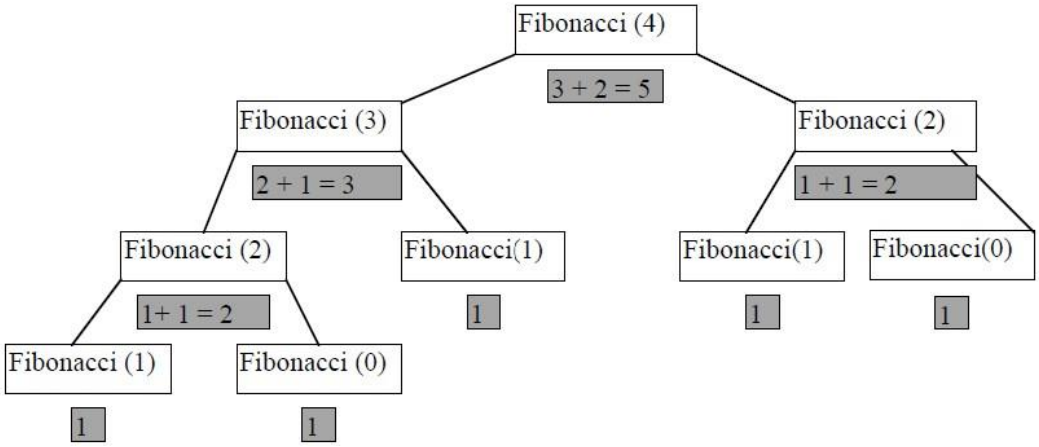


Figure 11 (Fibonacci Series Trace)

Time and Space Complexity of Fibonacci function:
Following is an estimation of the time and space complexity of the given **Fibonacci** function:

Time Complexity:

- The time complexity of this recursive function is exponential. Specifically, it's O(2^n), where 'n' is the input value representing the position in the Fibonacci sequence.

Explanation of Time Complexity:

- When you call **Fibonacci(n)** for some 'n', it makes two recursive calls: **Fibonacci(n - 1)** and **Fibonacci(n - 2)**.

- Each of those calls, in turn, makes two more recursive calls, and so on.
- This creates a binary tree-like structure where each level of the tree branches into two new levels.
- The depth of this recursive tree is 'n', and at each level, you have two subproblems. As a result, the total number of function calls made is approximately 2^n, which leads to an exponential time complexity.

Space Complexity:

- The space complexity of this function is determined by the maximum depth of the recursive call stack.
- In the worst case, the depth of the call stack is 'n' (when n is a positive integer). Therefore, the space complexity is O(n).

Explanation of Space Complexity:

- Each recursive call creates a new activation record (stack frame) on the call stack, which stores information about the function call.
- In this case, because we have 'n' levels of recursion, there will be 'n' activation records on the call stack simultaneously at the peak of the recursion.
- The space complexity is determined by the maximum number of activation records that need to be stored simultaneously, which is 'n' in this case.

Hence, the given Fibonacci function has exponential time complexity (O(2^n)) and linear space complexity (O(n)) due to the recursive nature of the algorithm and the need to store activation records on the call stack. This makes it inefficient for large values of 'n', and alternative methods like dynamic programming should be used for more efficient Fibonacci number calculations.

5. Function to find the minimum element in an array:

```
3   int findMinRec(int* A, int size)
4   {
5       // Base case: if there is one element in, it`s the minimum
6       if (size == 1)
7           return A[0];
8       int this_element = A[size - 1];
9       int sub_problem = findMinRec(A, size - 1); //Recursive case
10      int answer = min(this_element, sub_problem); // combine result;
11      return answer;
12      
13  }
```

Figure 12 (Find Min Number)

Explanation:
The function **findMinRec** is a recursive function that finds the minimum element in an array of integers **A** of size **size**. It does so by breaking down the problem into smaller subproblems and finding the minimum element among those subproblems. Here's how the function works:

1. It takes two parameters: an array of integers **A** and the size of the array **size**.

2. It checks if the size of the array is equal to 1 (**size == 1**). If this condition is met, it means that the array has only one element, so it returns that element as the minimum. This serves as the base case for the recursion.
3. If the size of the array is greater than 1, it proceeds to find the minimum element in the following way:
    - It selects the last element of the array (**this_element = A[size - 1]**).
    - It then recursively calls itself (**findMinRec**) with the array **A** and a reduced size (**size - 1**). This recursive call aims to find the minimum element in the subarray formed by removing the last element.
    - Finally, it compares the **this_element** with the result of the recursive subproblem (**sub_problem**) using the **min** function, and the smaller of the two is returned as the minimum.
4. The recursion continues to break down the problem into smaller subproblems, each time removing one element from the end of the array until it reaches the base case where the size is 1.
5. At each level of recursion, it compares the current element with the minimum obtained from the **sub_problem** and returns the minimum of the two.
6. As the recursion unwinds (from the base case towards the original call), it eventually returns the minimum element of the entire array.

The function uses a divide-and-conquer approach to find the minimum element in the array. It divides the problem into smaller subproblems and combines the results to find the overall minimum.

Time and Space Complexity of findMinRec:

The given function **findMinRec** recursively finds the minimum element in an array of integers **A** of size **size**. Let's estimate its time and space complexity:

Time Complexity:

- In each recursive call, the function reduces the size of the problem by 1 (**size - 1**) until it reaches the base case (**size == 1**).
- Therefore, the function makes 'size' recursive calls.
- In each recursive call, it performs constant-time operations (e.g., comparisons and assignments).

The time complexity of the function is O(n), where 'n' is the size of the input array. It makes 'n' recursive calls, each of which takes constant time.

Space Complexity:

- The space complexity is determined by the depth of the recursion stack because each recursive call adds a new frame to the call stack.
- In the worst case, the recursion depth is 'size' because the function makes 'size' recursive calls before reaching the base case.
- Each stack frame stores local variables such as **this_element**, **sub_problem**, and **answer**, as well as the return address.

The space complexity of the function is O(size), which is equivalent to O(n) in terms of the size of the input array. This means that the space used on the call stack is proportional to the size of the input array.

Hence, the time complexity of the **findMinRec** function is O(n), and the space complexity is O(n), where 'n' is the size of the input array. The function has a linear time complexity because it makes a linear number of recursive calls and a linear space complexity due to the recursion stack.

Task 01 Power Find:

Write the recursive function to calculate the power of a number. Calculate its time and space complexity.

int pow (int number, int power);

Task 02 Num Digits:

Write the recursive function to count the number of digits in a number. Calculate its time and space complexity.

int number_of_digits(int number);

Task 03 String Reverse:

Given a string s, the function should return the reverse of the string. Calculate its time and space complexity.

string reverse (string s, int len);

Task 04: Binary Search:

Given a sorted array of size **n** and target element. Perform binary search using recursion and return the index of element. Calculate its time and space complexity.

int binary_search (int* arr , int left , int right , int target);

Task 05 Dry run:

Dry run the following recursive function for binomial (7, 5) and find the output. Also, show the recursive trace.

```
int binomial(int n, int m)
{
        if (n == m || m == 0)
                return 1;
        else
                return binomial(n - 1, m) + binomial(n - 1, m - 1);
}
```

Calculate time and space complexity of the function int binomial (int n, int m).

Task 05 Sum Squares:

Write a recursive function to calculate the sum of squares of natural numbers up to a given number n. Calculate its time and space complexity.

int squares_sum(int n);

Example:
Input: 3
Output: 14
Explanation: 1^2 + 2^2 + 3^2 = 14

Task 06 Find GCD:

Write a recursive function to find the GCD of two numbers. Calculate its time and space complexity. **Hint:** The greatest common divisor (GCD) of two or more numbers is the greatest common factor number that divides them, exactly. It is also called the highest common factor (HCF). For example, the greatest common factor of 15 and 10 is 5, since both the numbers can be divided by 5. 15/5 = 3. 10/5 = 2.

Task 07 Print Bitsrings:

Write a function to print all the possible bitstrings of length n. Function call will be made from the main function in the following figure at line 10. Also, calculate time and space complexity of the function.

```
1   #include <iostream>
2   using namespace std;
3   void func(string s, int len)
4   {
5
6   }
7   int main()
8   {
9       func("", 3);
10      return 0;
11  }
```

Figure 13 (Print Bit string)

Hint: A bitstring is a binary sequence of bits, where each bit can have one of two values: 0 or 1. Bitstrings are commonly used in computer science and digital electronics to represent information, data, or instructions in a binary format.

Output:

```
Select Microsoft Visual Studio Debug Console
000
001
010
011
100
101
110
111
```

Figure 14 (Output)

Task 08 Rectangle Print:

Write a recursive function that should print pattern shown in the following figure. This function should take number of rows as argument and print the pattern. Also, calculate time and space complexity of the recursive function.

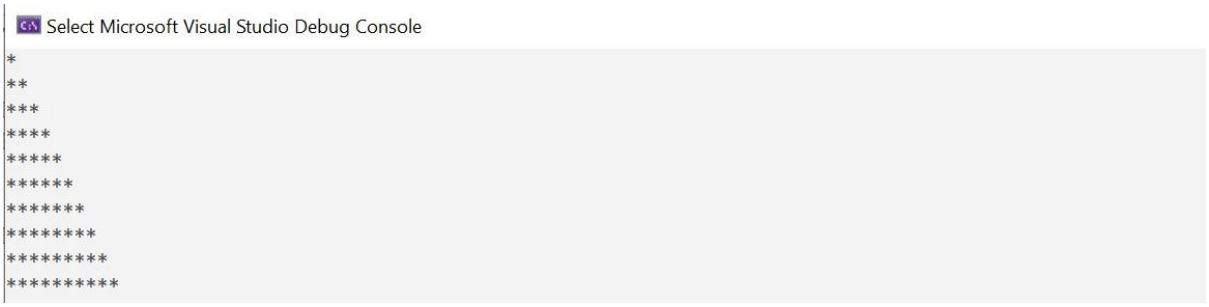


Figure 15 (Pattern Print)

Task 09 Tower of Hanoi:

Tower of Hanoi is a classic mathematical puzzle or problem that has intrigued mathematicians and computer scientists for centuries. It's often used as a pedagogical tool to teach recursion and algorithmic thinking. The problem is defined as follows:

Problem Statement: You are given three rods and a number of disks of different sizes, which can be slid onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, with the smallest disk at the top (i.e., the largest disk is at the bottom). The goal of the puzzle is to move the entire stack to another rod, subject to the following rules:

1. Only one disk can be moved at a time.
2. A disk can only be placed on top of a larger disk or an empty rod.

The puzzle starts with the initial configuration on one rod and ends with the final configuration on another rod. The objective is to move all the disks from the source rod to the destination rod while adhering to the rules, using the remaining rod as an auxiliary.

Your task is to write an algorithm such that for given number of plates N, it should print the **moves** to be taken for shifting all the plates from rod A to C. For example, for input N = 3, algorithm should print this:

- Move disc 1 from tower A to tower B.
- Move disc 2 from tower A to tower C. □ Move disc 1 from tower B to tower C.
- Move disc 3 from tower A to tower B.
- Move disc 1 from tower C to tower A. □ Move disc 2 from tower C to tower B.
- Move disc 1 from tower A to tower B.

Hint: Consider the very basic case where you have only two plates. What you'll do is:

- Lift the upper plate and shift it to the rod B.
- Lift the leftover plate on rod A and shift it to the rod C.
- Lift the small plate from the rod B and place it on the rod C.

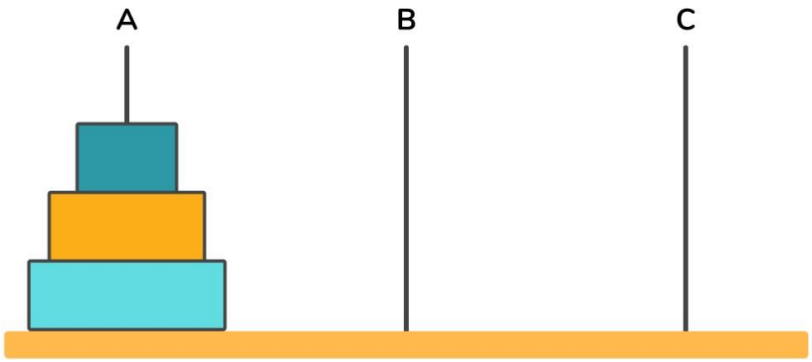


Figure 16 (Tower of Hanoi)

Task 10 Rat in a Maze Problem:

Rat in a Maze Problem is a classic algorithmic problem often used to teach recursion and backtracking. In this problem, you are given a maze with obstacles, and you need to find a path for a rat to navigate from the starting position to the destination. The rat can only move in four directions: up, down, left, and right, and it cannot cross obstacles or move outside the boundaries of the maze.

Here's how you can define the Rat in a Maze Problem as a recursive problem for a programming assignment in C++:

Problem Statement:
You are given a 2D maze represented as a grid, where each cell can be one of the following:

- '0': An open cell representing a valid path.
- '1': A blocked cell representing an obstacle.
- 'S': The starting cell.
- 'D': The destination cell.

Your task is to write a C++ program that finds a path for the rat to move from the starting cell ('S') to the destination cell ('D') if a valid path exists. The rat can move up, down, left, and right, but not diagonally, and it cannot pass through obstacles ('1').

Recursive Approach:
To solve the Rat in a Maze Problem recursively, you can use backtracking. Here's a high-level description of the recursive approach:

1. If the current cell is outside the maze boundaries or is an obstacle ('1'), return false, indicating that this path is invalid.
2. If the current cell is the destination ('D'), return true, indicating that a valid path has been found.
3. Mark the current cell as part of the path.
4. Recursively explore all four possible directions (up, down, left, and right) from the current cell, using the following logic:
    - Try to move in each direction.
    - If a valid path is found in any direction, return true.
    - If no valid path is found in any direction, backtrack by unmarking the current cell and return false.

5. If none of the directions lead to a valid path, return false to indicate that there is no path from the current cell to the destination.

Your C++ program should implement this recursive algorithm to find a path from 'S' to 'D' in the given maze. If a valid path exists, the program should print the path; otherwise, it should indicate that no path is found.

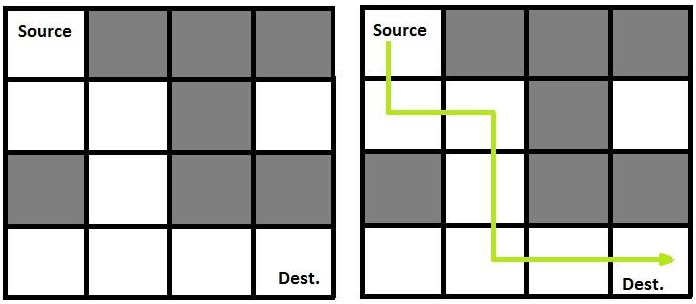


Figure 17 (Rat in a Maze)