Complexity Analysis & Recursion

Agenda

1. Complexity Analysis
    a. Introduction
    b. How to measure the efficiency of a program?
    c. Time Complexity
    d. Space Complexity
    e. Examples of Complexity analysis
    f. Time/Memory Limit Exceed
2. Recursion
    a. Introduction to Recursion
    b. How Recursion Works
    c. Recursion Flow
    d. Recursive Algorithms
    e. Direct recursion vs indirect recursion
    f. Advantages and disadvantages
    g. Recursive vs. iterative approaches

1. Complexity Analysis

a. Introduction

When we try to solve a problem or develop an application, it is common to have multiple solutions, each of which is correct and produces the desired output. However, implementing any of these solutions requires careful consideration, as each one comes with its own cost. The cost of a solution is determined by how much it consumes from the machine's resources, and the more it consumes, the worse it becomes. This makes sense because using more resources can make the solution work slower and less effectively.

b. How to measure the efficiency of a program?

When we want to measure how fast a program works, we can use a timer to see how long it takes to complete a task. However, there are factors that can affect the time, such as the computer's hardware.

For instance, a program that is not well-written might appear to work fast on a computer with a high-performance processor, but it might perform poorly on a slower computer. We also need to take into account other aspects such as the size of the task **(input size)** and the way the program was designed **(implementation)**.

To get a better understanding of how efficient a program is, we can use complexity analysis. **Complexity analysis** helps us analyze how much time and space a program needs to complete a task, based on the input size and how the program is designed.

By using complexity analysis, we can compare different programs more fairly and see how well they perform for different input sizes. This helps us create more efficient solutions that can handle larger tasks. So, complexity analysis is a valuable tool for evaluating the efficiency of programs and improving their performance.

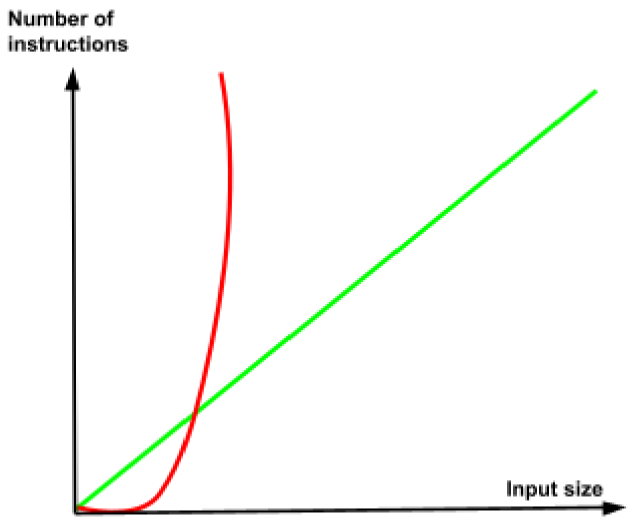
c. Time Complexity

- **What is Time Complexity?**

  When we solve problems with computer programs, we want to know how much time it takes for our program to complete its task. However, just counting the number of instructions in our program is not enough to measure how efficient it is. We need to take into account the size of the input and any differences in how the program is implemented. Time complexity is a way of measuring how the runtime of a program changes as the size of the input changes. It helps us understand the long-term behavior of our program, especially when the input gets very large.

- **How do we measure Time Complexity?**

  We measure the time complexity of a program by counting the number of basic operations it performs. The basic operations are usually comparisons, assignments, and arithmetic operations. To measure time complexity, we use a notation called Big O notation.



- **Understanding Big O Notation**

  Big O notation is a function that represents how much time our program takes to complete a task, based on the size of the input. The input size is the input to the function, and the output is the expected number of instructions that will be executed.

  For example, let’s say we have two programs that solve the same problem One program **the red solution** seems to be faster for small inputs, but another program **the green solution** is faster for larger inputs.

- **How can we compare two solutions?**

  Big O notation can help us here. We can use it to describe how the expected runtime of each program changes as the size of the input grows. The program with a lower Big O notation is generally more efficient, because it takes less time to complete its task as the input gets larger. By describing the long-term behavior of our program, we can compare different solutions and choose the one that will be the most efficient in the long run.

- **How to calculate Big O Notation?**

  Big O notation is not interested in calculating the exact number of instructions, as it is almost impossible due to many factors. Instead, Big O notation is interested in calculating the effect of the growth of the input on the time consumed or the number of instructions. To calculate the Big O notation for any solution, the first thing we need to do is to calculate the expected number of instructions we need to execute depending on the input size. For example, if we have solved a problem and it seems that the expected number of instructions is $\frac{n^2}{2} + 30n$, the first step is to get rid of constants $n^2 + n$, Then, we find which of the terms grows faster than the others, and this term will represent our solution so the complexity is $O(n^2)$.

- **Conclusion**

  The implementation differences are no longer considered, as we treat $O(n)$, $O(2n)$, and $O(3n)$ as all being $O(n)$. Our focus is on how the function's complexity grows, rather than the exact number of instructions it executes. The hardware is no longer considered a problem, and the issue of input size has been resolved. We now have a function that can handle any value of n and provide the expected number of instructions, so input size is no longer a concern.

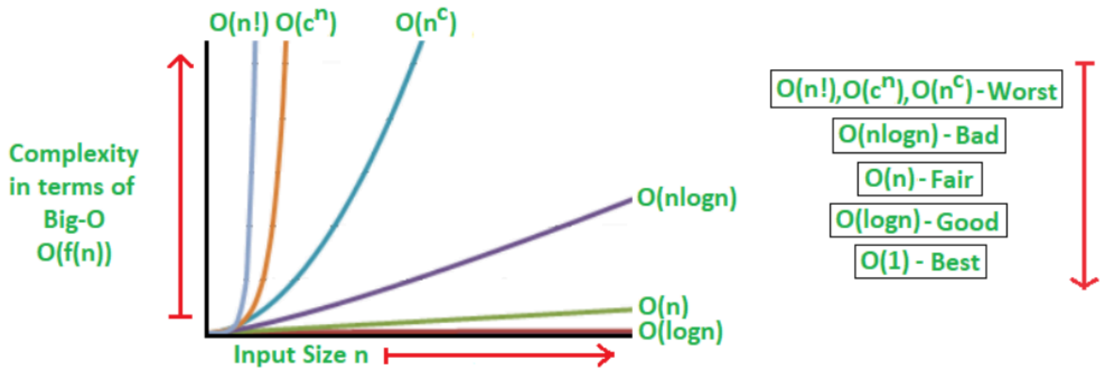
d. Space Complexity

- **What is Space Complexity?**

  Space Complexity is almost the same as Time Complexity but instead, it measures how much memory a program takes.

- **How do we measure Space Complexity?**

  We measure the space complexity of a program by counting the amount of memory it uses to complete its task. We consider the memory used by the program itself (i.e., the code) as well as the memory used by any container **(array)** and variables created by the program. We can use Big O notation to describe the space complexity of a program, just as we do for time complexity, which is getting rid of constants and finding the terms that grow faster than the others.



e. Examples of Complexity analysis

- **Code Examples**

Example: Given two numbers a and b, check if a is divisible by b.

Input:	Output:
25 5	Yes
18 8	No

Solution:

```
#include <iostream>
using namespace std;

int main() {
   int a, b;
   cin >> a >> b;
   cout << ((a % b == 0) ? "Yes" : "No") << endl;
}
```

- The Time Complexity of this solution is $O(1)$. This is because, regardless of the input size or the values of a and b, the algorithm executes a single operation. The growth of the input size does not affect the number of operations performed, resulting in a constant time complexity of $O(1)$.
- The Space Complexity of this solution is $O(1)$. This is because there are only two variables of type int which are allocated in the memory for the problem requirements so it's also a constant memory.

Example: Given a positive integer n, find the lowest power of 2 that is greater than n.

Input:	**Output:**
14	16
20	32

Solution:

```
#include <iostream>
using namespace std;

int main() {
   int n;
   cin >> n;
   int power = 1;
   while (power <= n)
       power *= 2;
   cout << power << endl;
}
```

- The Time Complexity of this solution is $O(log\ n)$. This is because, in the worst case scenario where n is a power of 2, it will take log n iterations to go through the loop until the highest power of 2 that is less than or equal to n is found.
- The Space Complexity of this solution is $O(1)$. This is because there are only two variables of type int which are allocated in the memory for the problem requirements so it's also a constant memory.

Example: Given n elements, find the maximum element among them.

Input:	**Output:**
5 10 20 13 50 7	50

Solution:

```cpp
#include <iostream>
using namespace std;

int main() {
   int n;
   cin >> n;
   int mx = INT_MIN;
   for (int i = 0; i < n; i++) {
       int cur;
       cin >> cur;
       mx = max(mx, cur);
   }
   cout << mx << endl;
}
```

- The Time Complexity of this solution is $O(n)$ because the loop iterates n times, making it run in linear time.
- The Space Complexity of this solution is $O(1)$. This is because there are four variables of type int which are allocated in the memory for the problem requirements so it's also a constant memory.

Example: Given an array of n elements and an integer k, find if there are two elements with sum divisible by k.

Input:	**Output:**
5 10 2 6 12 8 4	“YES”

Solution:

```cpp
#include <iostream>
using namespace std;

const int N = 1005;
int arr[N];

int main() {
   int n, k;
   cin >> n >> k;
   for (int i = 0; i < n; i++)
       cin >> arr[i];
   bool flag = false;
   for (int i = 0; i < n; i++)
       for (int j = 0; j < n; j++)
           if ((arr[i] + arr[j]) % k == 0)
               flag = true;
   cout << ((flag) ? "YES" : "NO") << endl;
}
```

- The Time Complexity of this solution is $O(n^2)$. This is because the outer loop iterates n times, and the inner loop iterates n times on each iteration the outer loop iterates.
- The Space Complexity of this solution is $O(n)$. This is because there is an array of size N which needs $N \times 4$ Bytes and it is the greatest term (int is 4 bytes).

- **Built-In Functions Complexity**

Function	Complexity
`min(a,b)`	$O(1)$
`max(a,b)`	$O(1)$
`swap(a,b)`	$O(1)$
`binary_search(arr, arr + n, x)`	$O(log(n))$
`lower_bound(arr, arr + n, x)`	$O(log(n))$
`upper_bound(arr, arr + n, x)`	$O(log(n))$
`pow(a,n)`	$O(n)$
`reverse(arr, arr + n)`	$O(n)$
`min_element(arr, arr + n)`	$O(n)$
`max_element(arr, arr + n)`	$O(n)$
`fill(arr, arr + n, x)`	$O(n)$
`count(arr, arr + n, x)`	$O(n)$
`find(arr, arr + n, x)`	$O(n)$
`sort(arr, arr + n)`	$O(n \cdot log(n))$
`next_permutation(arr, arr + n)`	$O(n!)$

f. Time/Memory Limit Exceed

There are two types of verdicts that may appear to most of us while solving a problem, namely "Time Limit Exceeded" (TLE) and "Memory Limit Exceeded" (MLE). These errors mean that our solution may be 100% correct, but it needs some optimization to be accepted. In such situations, we have to think of a better way to solve the problem or optimize the time or memory needed to solve it.

Theoretically, a common computer can perform 10^9 operations in one second. However, practically, our code should not take more than 10^8 operations due to several factors, such as ignored factors in Big O notation and online judge limitations.

For example, if we are given n $(1 \leq n \leq 10^5)$ and our complexity is $O(nlog(n))$, which is about 10^6, it's fine. But if our complexity is $O(n^2)$, which is about 10^{10}, we will get a Time Limit Exceeded error.

2. Recursion

a. Introduction to Recursion

As a programming project grows in size, it can become challenging to manage all the code in one place, especially if it's all written inside the **main()** function. To address this issue, developers often use functions to break the code into smaller, more manageable pieces. In this approach, the main function serves as a driver function that calls other functions to execute the program's logic.

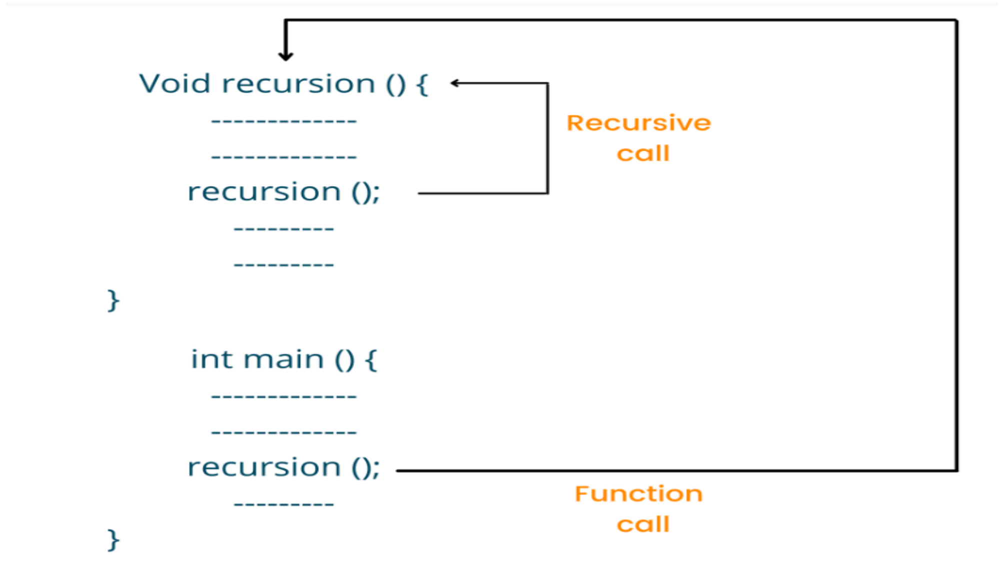
- **What is Recursion?**

  Recursion is a programming technique that involves breaking a problem down into smaller sub-tasks and repeatedly calling the function to solve them. It provides an alternative to the iterative method, which can be more time-consuming and require more effort.

  It is a useful technique that makes code shorter and easier to read and write.

```
void recursion() {
    recursion();
}

int main() {
    recursion();
}
```



```
Void recursion () {
    -------------
    -------------
    recursion ();
    ---------
    ---------
}
    int main () {
    -------------
    -------------
    recursion ();
    ---------
}
```

- **Importance of recursion in computer science**

  Recursion is an important concept in computer science because it is widely used in algorithms, data structures, and programming languages. It is particularly useful for dividing a problem into smaller subproblems and solving them independently. Recursion helps analyze the efficiency of algorithms and understand their complexity.

- **Basic example of recursive processes**

  Let’s discuss **Factorial** from the recursion point of view. Factorial is a mathematical operation that calculates the product of all integers from **1** to a given number **N**.

  How can Factorial(**N**) be represented by a smaller subproblem?

$$Factorial(N) = N * Factorial(N - 1)$$

  Factorial(N - 1) itself can be represented by a smaller subproblem as follows:

$$Factorial(N - 1) = (N - 1) * Factorial(N - 2)$$

And so on.

So, computing Factorial(N) recursively involves repeatedly multiplying N by Factorial($N - 1$) until 1 is reached.

For example:

- 5! = 5 * 4!
    - 4! = 4 * 3!
        - and so on.

This is how recursion breaks down the problem into smaller subproblems and combines the results.

b. How Recursion Works

➢ Base Case

- The base case in recursion is a fundamental condition that serves as the termination point for the recursive algorithm.
- It defines the scenario in which the recursive function stops calling itself and returns a specific result directly, without further recursion.
- Base cases are essential in recursion to prevent infinite recursion and to provide a final answer or outcome for the problem being solved.

➢ Recursive Case

- The recursive case in recursion refers to a specific condition within a recursive function or algorithm where the function calls itself with a **modified input**, typically moving **closer** to a base case, in order to break down a complex problem into simpler, more manageable subproblems.
- This process continues until the base case is reached, allowing the algorithm to solve the problem by aggregating the results from all the recursive calls.

➢ Stack memory and function calls in recursion

- In recursion, function calls create stack frames on the call stack. Each frame represents an active function call and contains relevant information. As the recursion progresses, stack frames are added and removed from the stack in a Last-In-First-Out (LIFO) order. Recursive calls continue until a base case is reached, and then the results propagate back through the stack.
- It is important to consider stack memory usage to avoid stack overflow errors. Recursion should be designed with proper termination conditions and mindful memory management.

➢ **Stack Overflow**

- A stack overflow is a software error that occurs when a program attempts to use more memory than is available on the stack, causing the program to crash.
- If a recursive function doesn't reach or define a base case, the recursive calls will never stop, which will cause a stack overflow problem.
- To avoid infinite recursion, Ensure that your recursive function has well-defined base cases. These are the conditions under which the recursion terminates, preventing further recursive calls. Make sure your base cases are reachable and correctly designed to cover all scenarios.

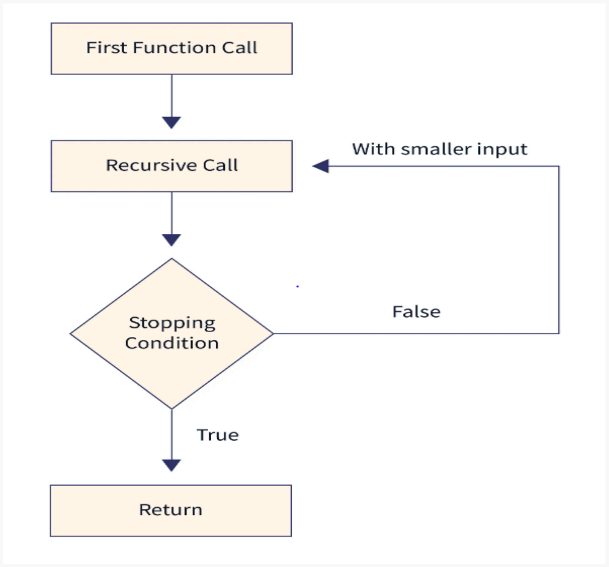
c. Recursion Flow

The recursion process begins with the first function call and continues until a base case is reached, after which the return statement is executed to terminate the recursion.

```cpp
#include <iostream>
using namespace std;

void printNumbers(int n) {
    if (n < 1) {
        return;
    }
    printNumbers(n - 1);
    cout << n << " ";
}

int main() {
    printNumbers(6);
    return 0;
}
```



```
Output:

1 2 3 4 5 6
```

Explanation:

- The main function calls the printNumbers function with an argument of 6, passing 6 as the value of the n input parameter.
- The printNumbers function is called with an input value of 6 for its n parameter.
- Since n is not less than 1, the function calls itself with an argument of n-1, which is 5.
- The new call to printNumbers is made with an input value of 5 for its n parameter.
- Since n is not less than 1, the function calls itself again with an argument of n-1, which is 4.
- The new call to printNumbers is made with an input value of 4 for its n parameter.
- This process continues until the function call is made with an input value of 1 for its n parameter.
- Since n is now less than 1, the function immediately returns to the previous call.
- The cout statement is executed, printing the value of n (which is 1) followed by a space.
- The function returns to the previous call, which also executes the cout statement and prints the value of n (which is 2) followed by a space.
- This process continues until the function returns to the original call made by the main function.
- The printNumbers function has printed the numbers 6, 5, 4, 3, 2, and 1 in reverse order.
- The main function returns 0, indicating that the program executed successfully.

To print numbers in reverse order using recursion, you can modify the printNumbers function from the previous example as follows:

```
void printNumbers(int n) {
    if (n < 1) {
        return;
    }
    cout << n << " ";
    printNumbers(n - 1);
}
```

The only change is the order of the cout statement and the recursive call. In this modified version of the function, the cout statement is executed before the recursive call, resulting in the numbers being printed in reverse order.

d. Recursive Algorithms

- **Examples**

  ➢ **Print the nth Fibonacci number:**

```
#include<iostream>
using namespace std;

int fibonacciRecursive(int n) {
    if (n <= 1)
        return n;
    return fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2);
}

int main () {
    int n = 10;
    cout << fibonacciRecursive(n) << endl;
    return 0;
}
```

➢ **Power Function: x^n:**

```cpp
#include<iostream>
using namespace std;

int recursivePow(int x, int n) {
    //Base Condition
    if(n == 0) return 1;

    //Edge case
    if(x == 0) return 0;

    //Recursive call
    return x * recursivePow(x, n - 1);
}

int main () {
    int x, n;
    cin >> x >> n;
    cout << recursivePow(x,n);
    return 0;
}
```

- **Time complexity in recursive algorithms**

  ➢ The time complexity of a recursive algorithm is the number of times the function calls itself, multiplied by the time it takes to execute each recursive call.

  ➢ The time complexity of a recursive algorithm can be analyzed using a recursion tree. A recursion tree is a tree diagram that shows all the recursive calls made by the function. The depth of a node in the tree represents the number of times the function has been called at that point.

For example, let's analyze the time complexity of finding the n-th Fibonacci number using recursion:

1. The maximum depth in the recursion tree is n recursive calls.
2. In each recursive call, the function makes two new recursive calls, which means that the total number of calls is multiplied by 2.
3. So, the total number of times the function calls itself to find the n-th Fibonacci number is 2^n.

That’s why, the time complexity of finding the n-th Fibonacci number using recursion is $O(2^n)$.

e. Direct recursion vs Indirect Recursion

➢ Direct recursion:

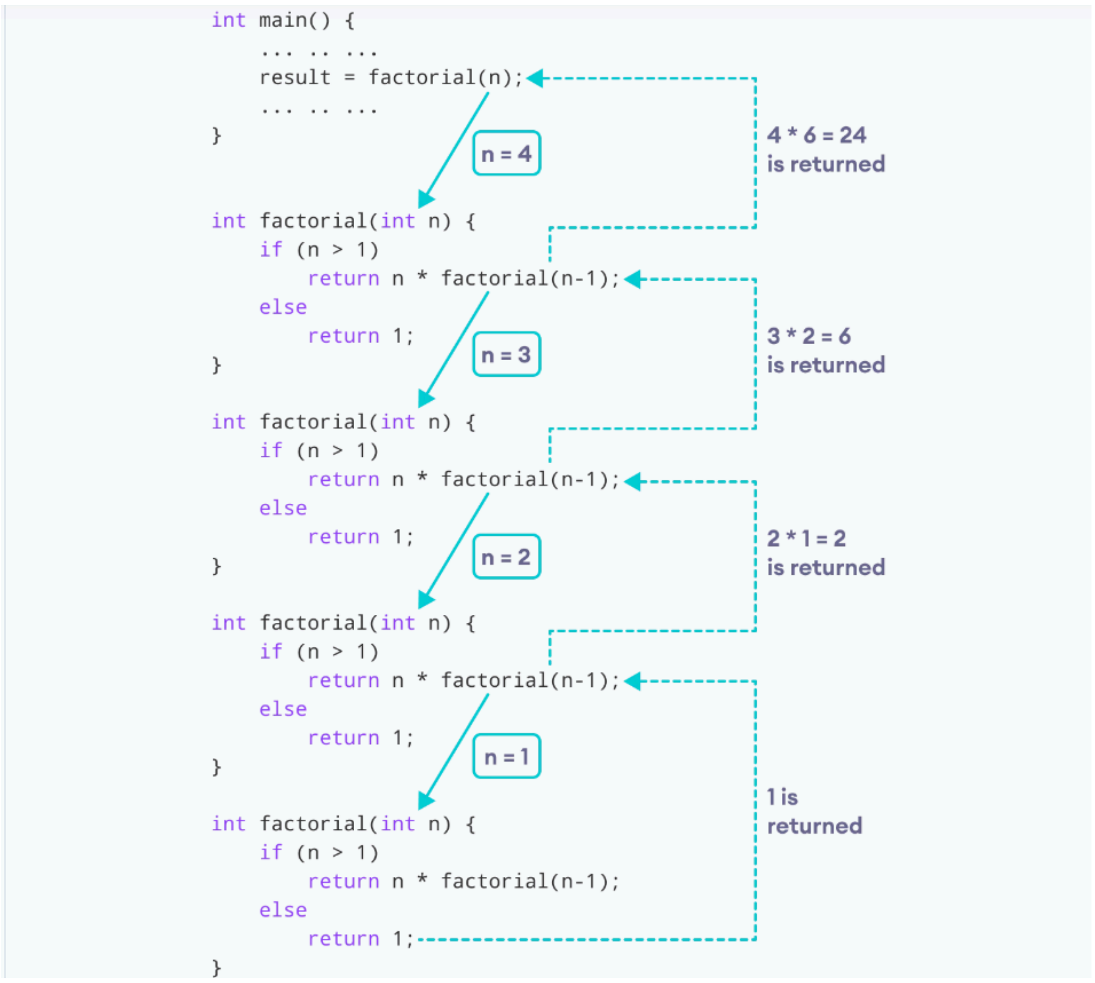
When a function calls itself, this is referred to as direct recursion. The example we saw earlier is an example of direct recursion.

```cpp
#include <iostream>
using namespace std;

int factorial(int n) {
    if(n > 1)
        return n * factorial(n - 1);
    else
        return 1;
}

int main() {
    int num;
    cin >> num;
    cout << factorial(num) << endl;
    return 0;
}
```

Working of factorial program:



```
int main() {
    ... .. ...
    result = factorial(n);
    ... .. ...
}

int factorial(int n) {
    if (n > 1)
        return n * factorial(n-1);
    else
        return 1;
}

int factorial(int n) {
    if (n > 1)
        return n * factorial(n-1);
    else
        return 1;
}

int factorial(int n) {
    if (n > 1)
        return n * factorial(n-1);
    else
        return 1;
}

int factorial(int n) {
    if (n > 1)
        return n * factorial(n-1);
    else
        return 1;
}
```

```
return 5 * factorial(4) = 120
  └── return 4 * factorial(3) = 24
        └── return 3 * factorial(2) = 6
              └── return 2 * factorial(1) = 2
                    └── return 1 * factorial(0) = 1
```

➢ **Indirect recursion:**

Indirect recursion happens when one function calls another function, and that second function later calls the first function again. A classic example of indirect recursion is when function A calls function B, which in turn calls function A.

```cpp
#include <iostream>
using namespace std;

void indirectRecursion2(int);

void indirectRecursion1(int n = 1) {
    if(n > 9)
        return;
    cout << n <<" ";
    indirectRecursion2(n + 1);
}

void indirectRecursion2(int n = 1) {
    if(n > 9)
        return;
    cout << n << " ";
    indirectRecursion1(n+1);
}

int main() {
    int start = 1;
    indirectRecursion1(start);
    return 0;
}
```

f. Advantages and Disadvantages of Recursion

- **Advantages of Recursion:**

  Recursion can offer several benefits, such as making code more readable and concise, increasing modularity and flexibility, reducing code duplication, and improving the time complexity for certain problems.

- **Disadvantages of Recursion:**

  Disadvantages of recursion include the potential for stack overflow errors if the recursion depth becomes too large, high memory consumption due to the creation of new sets of variables and data for each function call, reduced performance due to function call overhead and stack manipulation, difficulty in debugging recursive functions, and increased code complexity.

g. Recursion VS Iteration

Recursion	Iteration
Terminates when the base case becomes true.	Terminates when the condition becomes false.
Used with functions.	Used with loops.
Every recursive call needs extra space in the stack memory.	Every iteration does not require any extra space.
Smaller code size.	Larger code size.

h. To solve:

- Recursion: Fibonacci Numbers
- Recursive Digit Sum
- 10344 - 23 out of 5