# TRIBHUVAN UNIVERSITY

Paschimanchal Campus

Lamachaur 16, Pokhara

## Department of Computer Engineering

**Lab Report On:**

**Lab No: 2**

**Title:**

**Submitted By**

**Name:** Anish Karki

**Roll No:** PAS080BCT007

**Submitted To:** HPB Sir

**Date of submission: 19/06/2025**

# PRACTICAL-1
# TOWER OF HANOI

## THEORY

Solving the Tower of Hanoi in C++ involves recursively moving disks between rods while following strict rules. In this program, we handle the movement of $n$ disks from a source rod to a destination rod using an auxiliary rod. Each move is determined based on the number of disks and the recursive steps involved.

## Tower of Hanoi Steps

- Select the number of disks to move.
- Move `n-1` disks from the source rod to the auxiliary rod.
- Move the largest (nth) disk to the destination rod.
- Move the `n-1` disks from the auxiliary rod to the destination rod.

## Boundary Conditions:

Moving 1 disk is a direct operation (base case).
Moving 0 disks requires no action (termination condition).
Must ensure larger disks are **never** placed on smaller ones.

## Time Complexity:

Move 1 disk: O(1)
Move $n$ disks: $O(2^n - 1)$

Since the number of operations doubles with each added disk, solving Tower of Hanoi becomes exponentially slower as n increases. Efficient recursion and understanding base cases help avoid logical errors.

# ALGORITHM

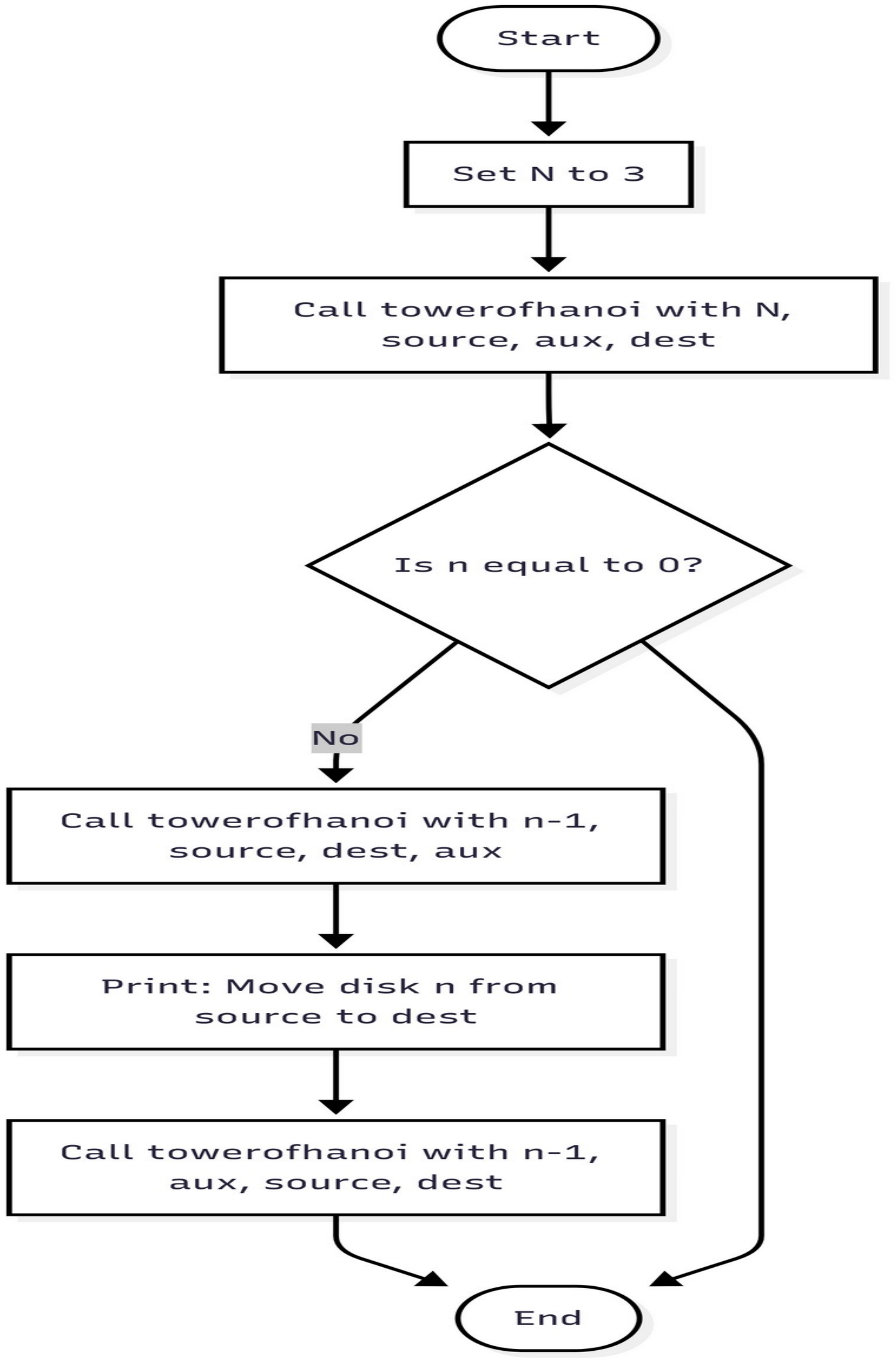
1) Start
2) Initialize Number of Disks
    - Set N = 3.

3) Call Recursive Function
    - Call towerofhanoi(N, 'A', 'B', 'C') to move N disks from rod 'A' to rod 'C' using rod 'B' as auxiliary.

4) towerofhanoi(n, from_rod, aux_rod, to_rod):
    - If n == 0
        - Return
    - Else
        - Recursively call towerofhanoi(n-1, from_rod, to_rod, aux_rod)
          (Move n-1 disks from from_rod to aux_rod using to_rod as auxiliary).
        - Print: "Move disk n from rod from_rod to rod to_rod".
        - Recursively call towerofhanoi(n-1, aux_rod, from_rod, to_rod)
          (Move n-1 disks from aux_rod to to_rod using from_rod as auxiliary).

5) End

# FLOWCHART



Start
Set N to 3
Call towerofhanoi with N, source, aux, dest
Is n equal to 0?
No
Call towerofhanoi with n-1, source, dest, aux
Print: Move disk n from source to dest
Call towerofhanoi with n-1, aux, source, dest
End

## CODE:

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

void towerofhanoi(int n, char from_rod, char aux_rod, char to_rod)
{
    if (n == 0)
        return;

    towerofhanoi(n - 1, from_rod, to_rod, aux_rod);

     cout << "Move disk " << n << " from rod " << from_rod << " to rod " << to_rod <<
endl;

     towerofhanoi(n - 1, aux_rod, from_rod, to_rod);
}

int main()
{
    int N = 3;
    towerofhanoi(N, 'A', 'B', 'C');

    return 0;
}
```

# PRACTICAL-2
# FIBONACCI SERIES

## THEORY

Generating the Fibonacci series in C++ involves calculating a sequence where each number is the sum of the two preceding ones. In this program, we handle the generation of n terms in the sequence, starting from the base values 0 and 1. Each new term is computed and added to the series iteratively or recursively.

### Fibonacci Series Steps

Select the number of terms to generate.
Start with the first two terms: 0 and 1.
For each subsequent term, add the previous two terms.
Continue this process until the required number of terms is reached.

### Boundary Conditions:

For 0 terms: No output.
For 1 term: Output only 0.
For 2 terms: Output 0 and 1 directly.
Be careful of invalid or negative term counts.

### Time Complexity:

Recursive approach: **O($2^n$)** (due to overlapping subproblems)

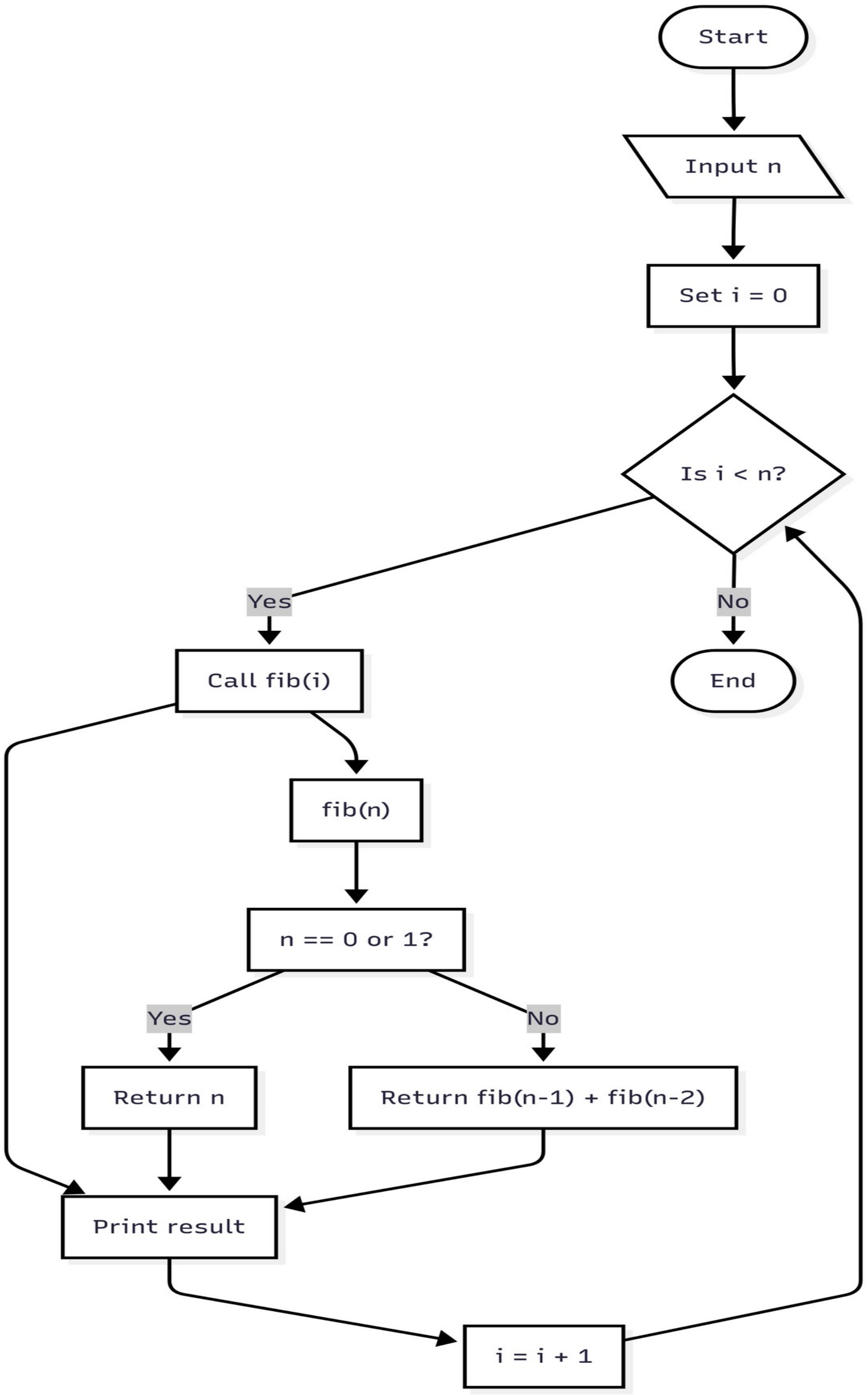
Using memoization or dynamic programming can reduce the recursive time complexity to **O(n)** by storing previously computed terms, avoiding redundant calculations.

# ALGORITHM

Steps:

```
        1. Start

2. Input:
   Prompt the user to enter the number of terms n.

3. Read the value of n.

4. For each integer i from 0 to n-1, do:
   a. Call the function fib(i) to compute the ith Fibonacci number.
   b. Print the result.

5. Function fib(x)
   a. If x == 0 or x == 1, return x.
   b. Else, return fib(x-1) + fib(x-2).

6. End
```

# FLOWCHART


Start
Input n
Set i = 0
Is i < n?
Yes
No
End
Call fib(i)
fib(n)
n == 0 or 1?
Yes
No
Return n
Return fib(n-1) + fib(n-2)
Print result
i = i + 1

## CODE:

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


int fib(int n)
{
    if (n == 0 || n == 1)
    {
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}

int main()
{
    int n;

    cout << "Enter the number of terms: ";

    cin >> n;

    for (int i = 0; i < n; i++)
    {
        cout << fib(i) << " ";
    }

    cout << endl;

    return 0;
}
```

# PRACTICAL-3
# PUSH POP OPERATION

## THEORY

Performing **push and pop operations** on a stack in C++ involves managing elements in a **Last In First Out (LIFO)** manner. In this program, we handle inserting (push) and removing (pop) elements using an array with a fixed size. The `top` index tracks the current top of the stack.

## Push Pop Operation Steps

**Push**: Check if space exists; if yes, insert at `top + 1`.
**Pop**: Check if stack is not empty; if yes, remove and return the top element.
Update `top` after every push and pop to reflect the current state of the stack.

## Boundary Conditions:

- Push when full causes stack overflow.
- Pop when empty causes stack underflow.
- Always update `top` correctly to avoid invalid memory access.

## Time Complexity:

Push: **O(1)**
Pop: **O(1)**

Since both operations deal only with the top of the stack, they are highly efficient. However, a fixed-size array limits capacity. Using dynamic structures like linked lists can allow flexible resizing.

# ALGORITHM

Steps:

1. Start

2. Define Stack Class:
    - Initialize top = -1.
    - Set capacity to the given size.
    - Allocate dynamic array arr of size capacity.

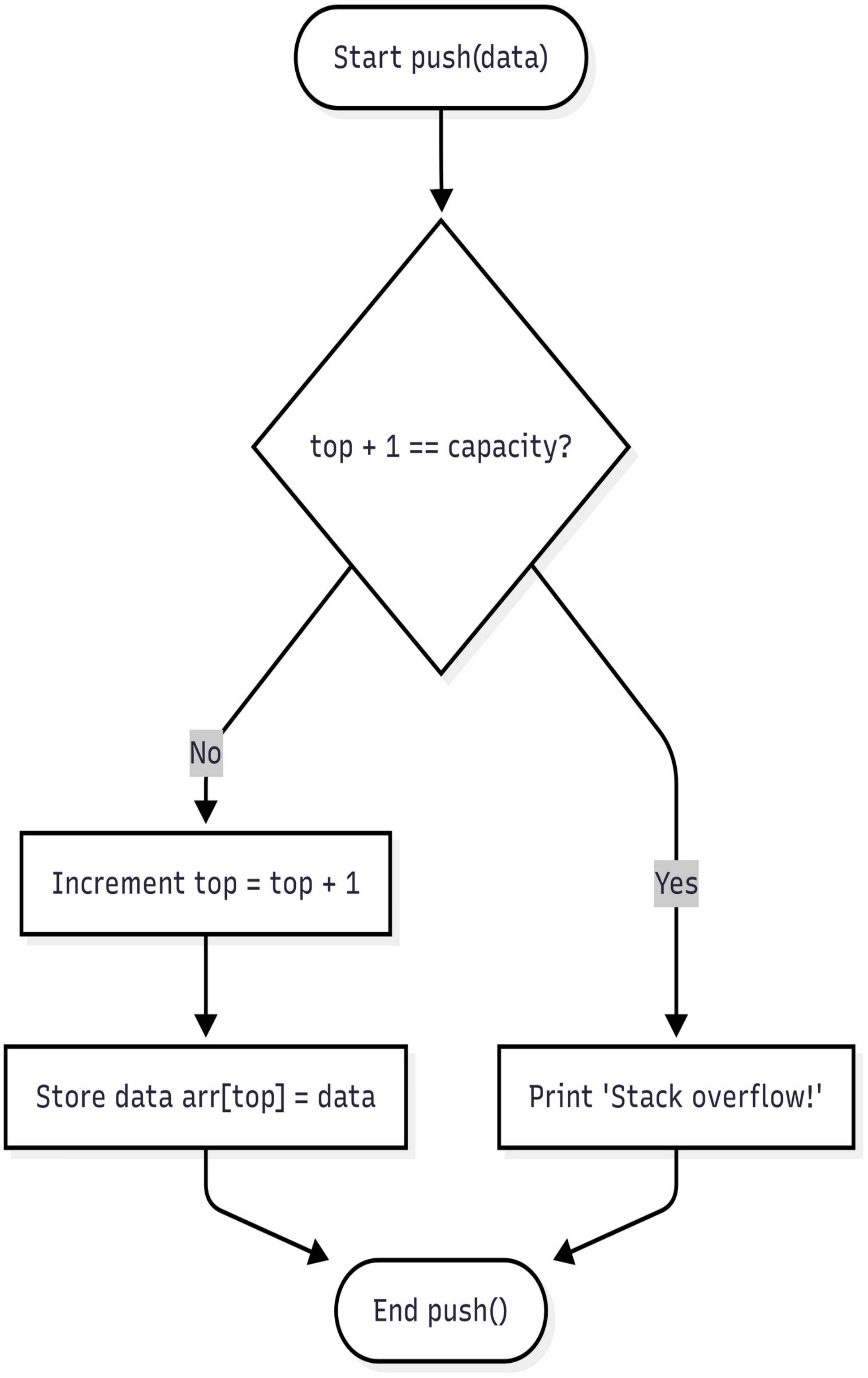
3. push(data):
    - If top + 1 == capacity, print "Stack overflow!" and return.
    - Else, increment top and set arr[top] = data.

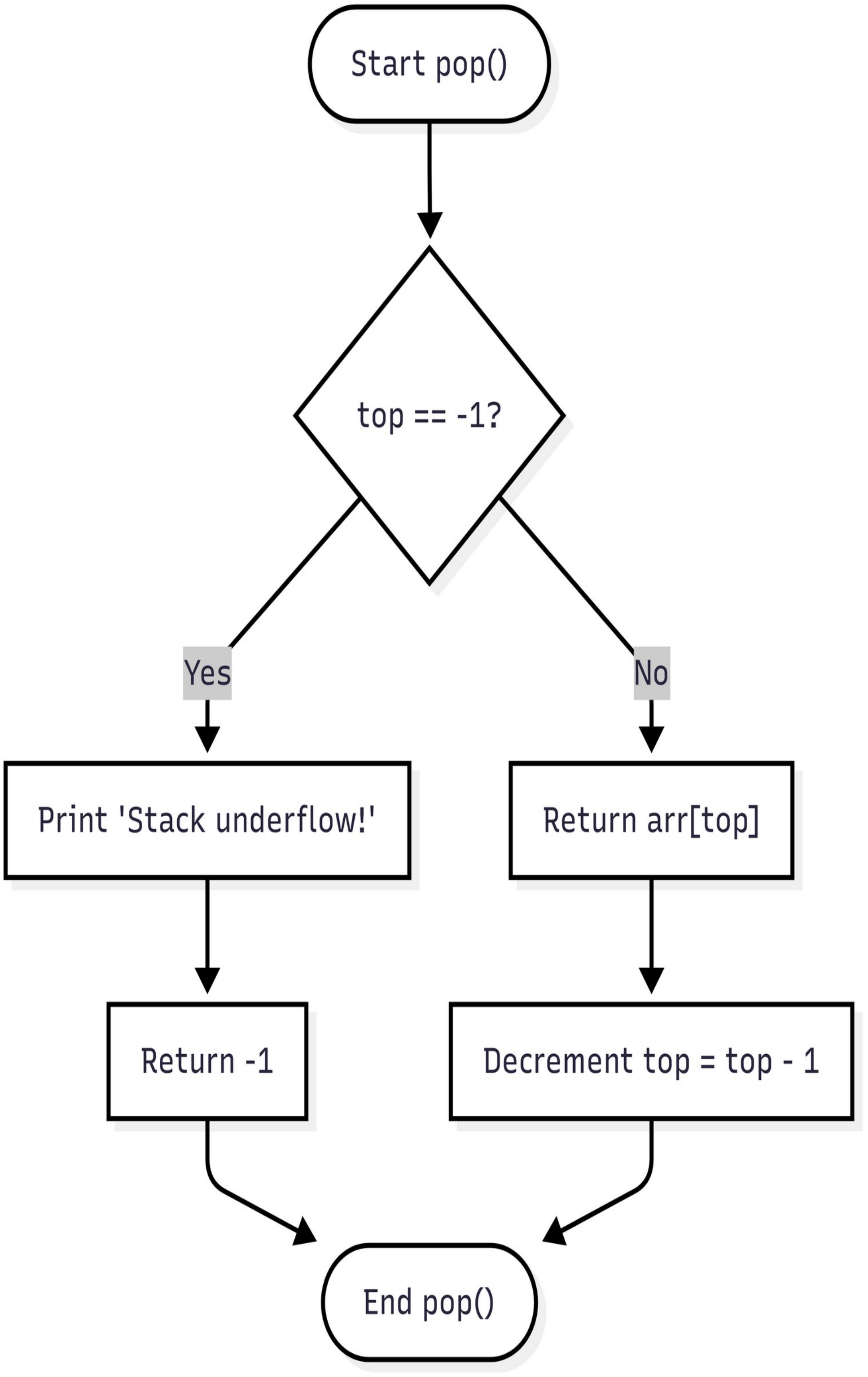
4. pop():
    - If top == -1, print "Stack underflow!" and return -1.
    - Else, return arr[top] and decrement top.

5. Main Function:
    - Create a stack s of size 5.
    - Push 10 onto the stack.
    - Push 20 onto the stack.
    - Pop and print the top element.
    - Pop and print the top element.
    - Pop and print the top element.

6. End

## FLOWCHART


Start push(data)
top + 1 == capacity?
No
Yes
Increment top = top + 1
Store data arr[top] = data
Print 'Stack overflow!'
End push()


Start pop()
top == -1?
Yes
No
Print 'Stack underflow!'
Return arr[top]
Return -1
Decrement top = top - 1
End pop()

## CODE:

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

class Stack {
private:
  int top;
  int *arr;
  int capacity;

public:
  Stack(int size) {
    top = -1;
    capacity = size;
    arr = new int[size];
  }

  void push(int data) {
    if (top + 1 == capacity) {
      cout << "Stack overflow!" << endl;
      return;}
    arr[++top] = data; }

 int pop() {
    if (top == -1) {
      cout << "Stack underflow!" << endl;
      return -1;  }
    return arr[top--];}
};

int main() {
  Stack s(5);
  s.push(10);
  s.push(20);
  cout << "Popped: " << s.pop() << endl;
  cout << "Popped: " << s.pop() << endl;
  cout << "Popped: " << s.pop() << endl; // underflow test
  return 0;
}
```

## CONCLUSION

In this lab, we implemented Fibonacci series using dynamic programming, Tower of Hanoi using recursion, and basic stack operations using arrays. These programs helped us understand core concepts like recursion, iteration, and stack-based memory management, along with handling edge cases such as overflow and underflow.