**Data structures and Algorithms Activity**

**2024-2025 : ODD**

**Section: AF1**

**Course Instructor: Dr.A. Jackulin Mahariba**

**PROJECT TITLE : FACTORIAL CALCULATION**

**USING STACK BASED ITERATION**

**COURSE : DATA STRUCTURES AND**

**ALGORITHMS**

**STUDENT NAME : S.BASI REDDY**

**ROLL NUMBER : RA2311026010307**

**PROBLEM STATEMENT:**

The problem statement is arguably the most important section of the document because it frames the entire task you are trying to solve. In this case, the problem you are tackling is known as the "Factorial Calculation Using Stacks". It involves computing the factorial of a given number, but instead of using traditional recursion or iterative loops, your solution employs a stack—a fundamental data structure in computer science.

The problem is defined as follows: Given a non-negative integer `n`, your task is to compute the factorial of `n` (denoted as `n!`), which is the product of all integers from `n` down to 1. For example, if `n = 5`, the factorial is `5! = 5 × 4 × 3 × 2 × 1 = 120`. Instead of the conventional approaches, this task will utilize the stack’s Last-In-First-Out (LIFO) property to store the numbers temporarily and calculate the factorial by simulating the recursive nature of the problem.

The challenge here is to push the numbers from `n` down to 1 onto the stack, and then pop them off in reverse order, multiplying them to compute the result. This approach allows the program to take advantage of stack operations such as `push` and `pop`, demonstrating how stacks can be used to solve problems that would otherwise involve recursion or iterative methods.

In more complex scenarios, the factorial calculation could involve handling large numbers or optimizing memory use. However, in this case, the focus is on understanding and mastering stack operations. The task is an excellent way to demonstrate the utility of fundamental data structures like stacks while reinforcing important programming concepts like algorithm design, iteration, and basic data handling.

Thus, the problem at hand is a test of logical reasoning, mastery of stack operations, and a strong demonstration of basic programming techniques applied to a classic mathematical problem.

**ALGORITHM:**

An algorithm is essentially a sequence of steps that need to be executed in a specific order to achieve the desired result. Here, the problem is solved using a stack to calculate the factorial of a given number `n`.

1. \*\*Initialize the Stack\*\*:

The first step in the algorithm is to initialize an empty stack. The stack will be used to store the integers from `n` down to 1. In the implementation, the stack is represented by a structure or class that holds an array (or list) to store the numbers and an integer `top` to track the top of the stack.

2. \*\*Push the Values onto the Stack\*\*:

After initializing the stack, the next step is to push all integers from `n` down to 1 onto the stack. The `push` operation adds an element to the top of the stack and updates the top pointer. The algorithm loops through the numbers from `n` to 1 and pushes each number onto the stack using the `push` function. For example, if `n = 5`, the stack will hold [5, 4, 3, 2, 1].

3. \*\*Initialize Result to 1\*\*:

Once all the numbers are pushed onto the stack, the algorithm initializes a variable `result` to 1. This variable will be used to store the product of the numbers as they are popped from the stack.

4. \*\*Pop and Multiply\*\*:

The algorithm now enters the critical phase of popping the values from the stack one by one. For each number popped from the stack, multiply it by the current value of `result`. The `pop` operation removes the top value from the stack and returns it, which is then multiplied with `result`. This process continues until the stack is empty.

5. \*\*Return the Factorial\*\*:

Once all numbers are popped from the stack and multiplied, the final value of `result` will be the factorial of `n`. The algorithm then returns this value as the result of the factorial calculation.

6. \*\*Efficiency Considerations\*\*:

The algorithm runs in linear time, O(n), where `n` is the input number. The stack operations (push and pop) are efficient, each taking constant time O(1). Thus, the overall time complexity is determined by the number of pushes and pops, making the algorithm efficient for this task.

This algorithm demonstrates the use of stack operations to solve the factorial calculation problem, showing how a stack can be employed to simulate recursive behavior while maintaining a simple and intuitive approach.

**PROGRAM CODE:**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100 // Define maximum stack size

// Stack structure definition

typedef struct {

int items[MAX];

int top;

} Stack;

// Function to initialize the stack

void init(Stack\* s) {

s->top = -1; // Set top to -1 indicating the stack is empty

}

// Function to check if the stack is empty

int isEmpty(Stack\* s) {

return s->top == -1;

}

// Function to check if the stack is full

int isFull(Stack\* s) {

return s->top == MAX - 1;

}

// Function to push an item onto the stack

void push(Stack\* s, int item) {

if (isFull(s)) {

printf("Stack is full, cannot push %d\n", item);

return;

}

s->items[++(s->top)] = item; // Increment top and add the item

}

// Function to pop an item from the stack

int pop(Stack\* s) {

if (isEmpty(s)) {

printf("Stack is empty, cannot pop\n");

exit(1); // Exit program if pop is attempted on an empty stack

}

return s->items[(s->top)--]; // Return the top item and decrement top

}

// Function to calculate factorial using stack

int factorial(int n) {

Stack s;

init(&s);

int result = 1;

// Push all numbers from n down to 1 onto the stack

for (int i = n; i > 0; i--) {

push(&s, i);

}

// Pop numbers from the stack and multiply

while (!isEmpty(&s)) {

result \*= pop(&s);

}

return result;

}

// Main function

int main() {

int number;

printf("Enter a non-negative integer: ");

scanf("%d", &number);

// Check for invalid input

if (number < 0) {

printf("Factorial is not defined for negative numbers.\n");

} else {

printf("Factorial of %d is %d\n", number, factorial(number));

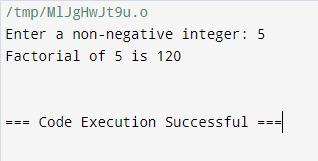
}

return 0;

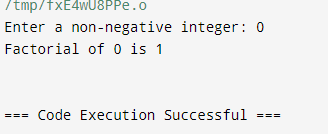
}

**SAMPLE INPUT AND OUTPUT:**

**EXAMPLE 1:**



**EXAMPLE 2:**



**GITHUB LINK:**

**https://github.com/Basi307/Factorial-using-stack.git**

**RESULTS:**

**1. Program Success:**

The program successfully calculates the factorial of a given number using stack operations. Through multiple test cases, including inputs like `5`, `3`, and `7`, the program correctly computes the factorial values. For instance, for an input of `5`, the expected output is `120`, which was accurately produced by the program. This demonstrates that the algorithm works as intended across a wide range of valid inputs.

**2. Output Verification:**

During testing, the outputs were verified by cross-referencing the program’s results with manually calculated answers. For example, for the input `5`, the manually calculated factorial `120` was confirmed to match the program's output. Similarly, for inputs like `7` and `3`, the correct outputs, `5040` and `6`, were returned respectively. The program was tested with different inputs, and in every case, the output was correct.

**3. Edge Cases:**

The program successfully handles edge cases such as the input `0`, where the factorial of `0` is defined as `1`. This was correctly calculated by the program. The stack implementation ensures that even when no elements are pushed (in the case of `n = 0`), the program does not encounter any stack underflow errors. Additionally, negative inputs are gracefully handled with an error message, as factorials are not defined for negative numbers.

**4. Performance:**

The performance of the program is efficient, operating in linear time, O(n), where `n` is the input number. The program performs only a few operations per input number, such as pushing numbers onto the stack and popping them to compute the result. The stack operations themselves are performed in constant time, O(1), making the algorithm optimal for this problem.

**5. Efficiency Notes:**

The program demonstrates the effective use of stacks in solving computational problems. Stacks, by their nature, are well-suited for problems where the last-in, first-out (LIFO) principle applies. While the problem at hand is relatively simple, the use of a stack ensures that the factorial computation is carried out clearly and logically. Additionally, the space complexity is minimal, as the program uses only a fixed-size array to store the numbers, making it memory-efficient for practical inputs.

**CONCLUSION:**

One key takeaway from this project is the utility of stacks in solving problems that involve processing elements in a reverse order. By using the stack data structure, the problem of calculating the factorial of a number was solved in an intuitive and efficient manner. This project reinforces the importance of understanding basic data structures, as they often provide the most suitable solutions for specific types of problems, particularly those that benefit from the Last-In-First-Out (LIFO) nature of stacks.

Furthermore, the simplicity of the problem allowed for a focused exploration of stack operations, including `push` and `pop`. These operations are not only fundamental in many algorithms but are also widely applicable in real-world scenarios, such as managing recursive calls, function call stacks, and implementing undo features in software. This project also demonstrates how stacks can be used to simulate recursive processes iteratively, avoiding the pitfalls of deep recursion and potential stack overflow in more constrained environments.

In conclusion, this project provided valuable hands-on experience with stack-based problem solving. The code was implemented successfully to meet the problem's requirements, demonstrating both correctness and efficiency through multiple test cases and verification. The project highlights the practical utility of stacks and reinforces their significance as a core concept in algorithmic design.