Erick Gonzalez Parada #178145

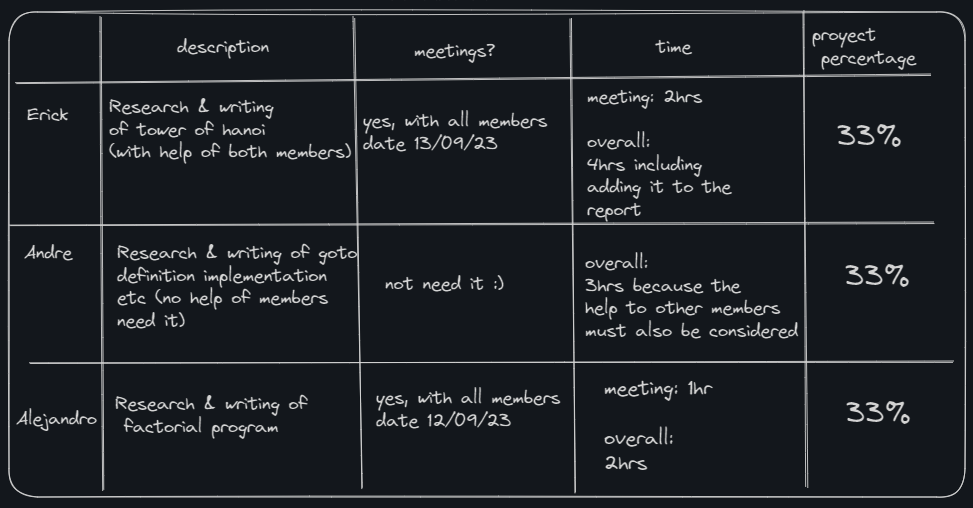
Andre Francois Duhamel Gutiérrez #177315

Alejandro González Díaz #178645

Activity #4 Tower of Hanoi

Link to appendix of teamwork: <https://drive.google.com/file/d/1dxa9-YJJBNCZ17uu5_R6gwndGzfWvVZg/view?usp=sharing>

(Download this above to have a better view)



The activity consists of the following 3 bullet points:

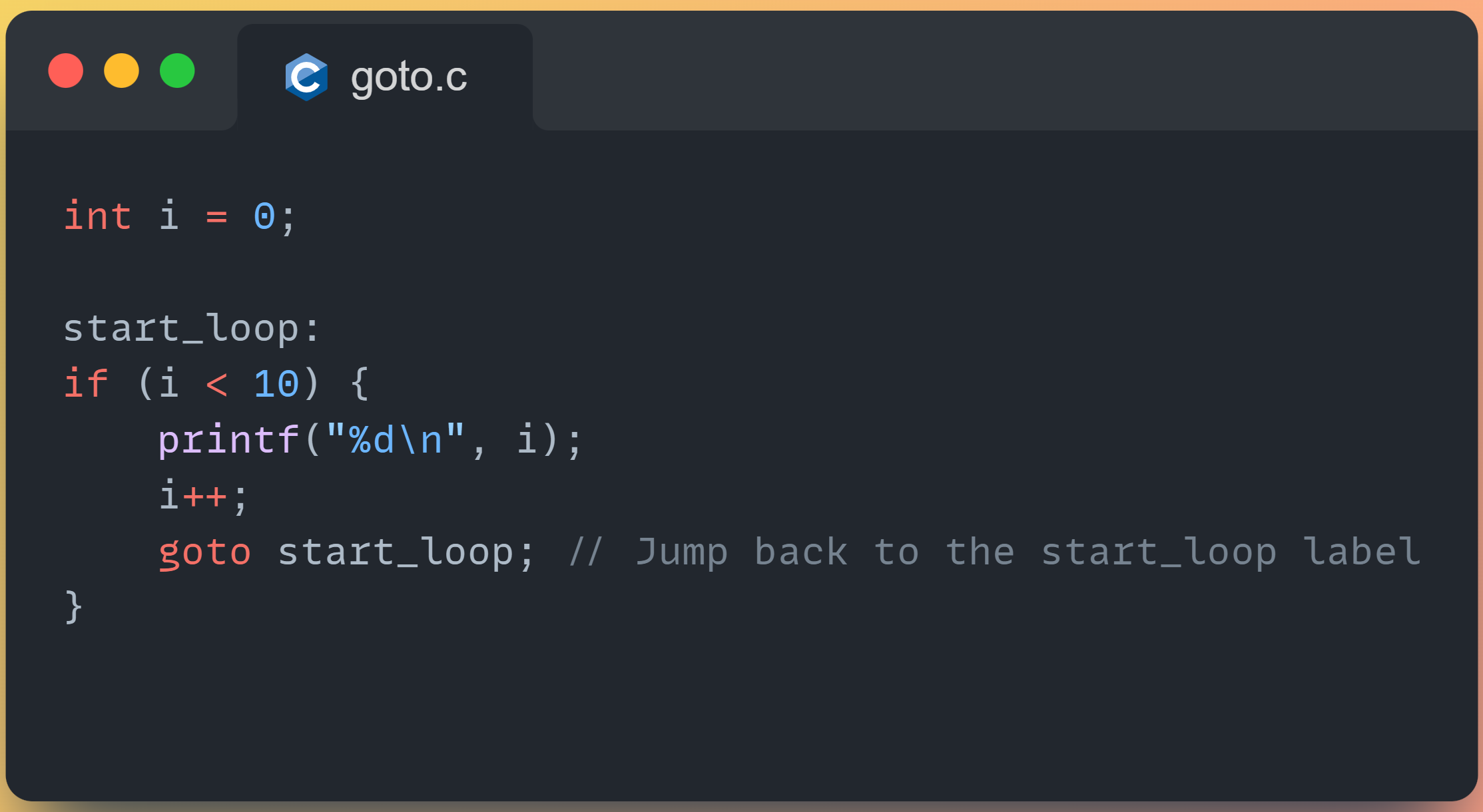
* GOTO statement. (1)
* Factorial program by managing recursion. (2)
* Implement “Towers of Hanoi” by managing recursion. (3)

# Problem 1 (goto)

## GOTO statement definition

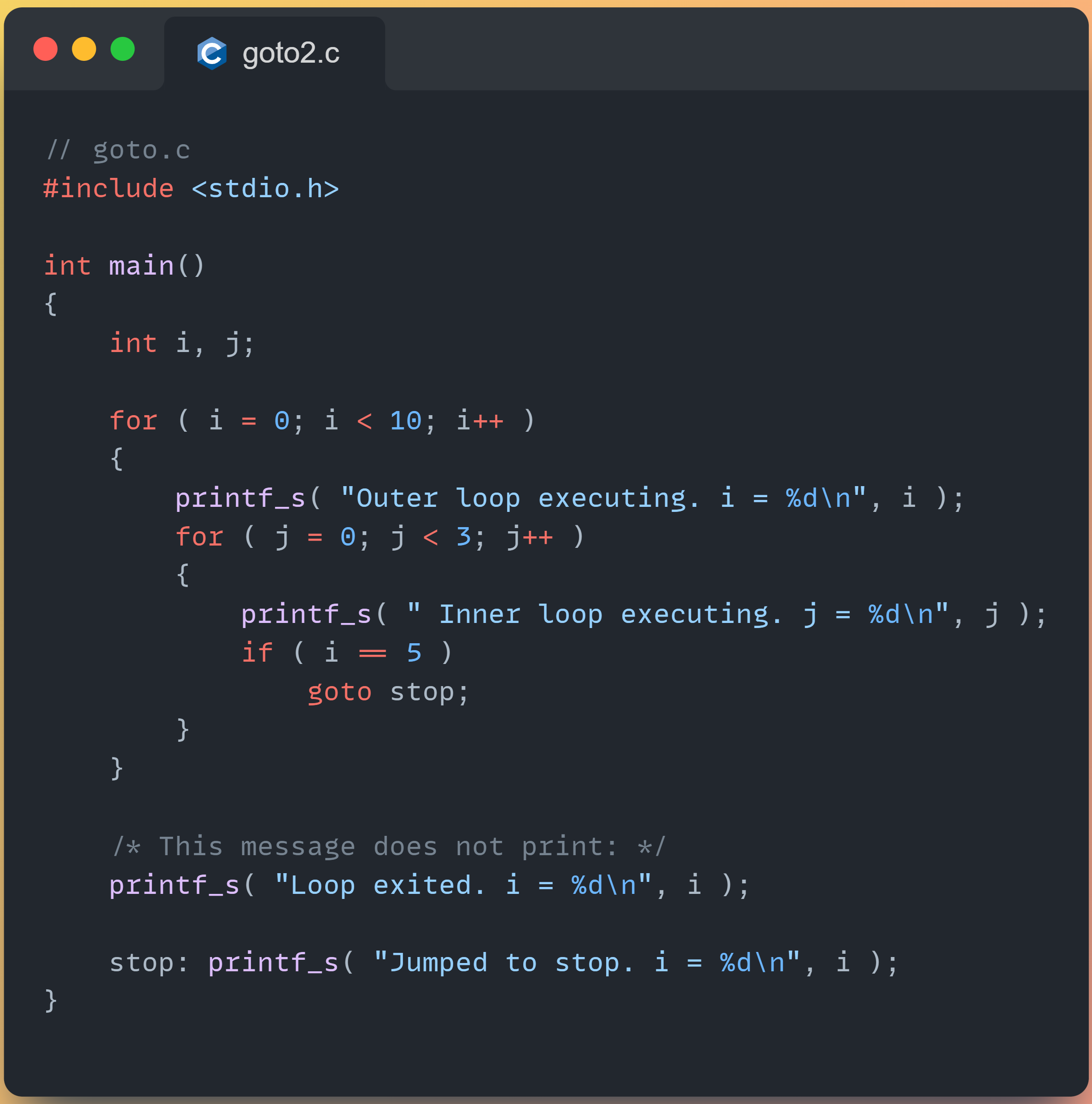
[[1]](#_References_&_Bibliography) The “goto” command is a controversial command in the C language. This keyword access is given by the header <stdio.h>, this allows the compiler to jump to a certain label. This means that we can get out of a double nested, to do something else.

## Sample 1



In this example, the program will repeatedly print numbers from 0 to 9 and then jump back to the start loop label until *i* becomes 10.

## Sample 2



In this example the goto jumps to the label stop when i = 5.

## Downsides

[[1]](#_References_&_Bibliography) Why It is not Recommended: The use of goto is discouraged for several reasons:

Readability: Code that relies heavily on goto statements can become hard to read and understand, especially in larger programs. It can lead to spaghetti code, where the flow of execution is difficult to follow.

Debugging: Debugging code with goto statements can be challenging because it makes it non-linear and can lead to unexpected behavior. Identifying the source of bugs can be more difficult.

Maintainability: Code with goto statements is often harder to maintain and modify. Making changes to the code can introduce subtle bugs due to the non-linear flow of execution.

Structured Programming: The use of goto can violate the principles of structured programming, which emphasizes clear, structured control flow through the use of loops and conditionals.

Alternative Control Structures: In modern C programming, there are better alternatives to achieve the same goals as goto. Loops, conditionals, and functions can provide structured and readable code that is easier to maintain.

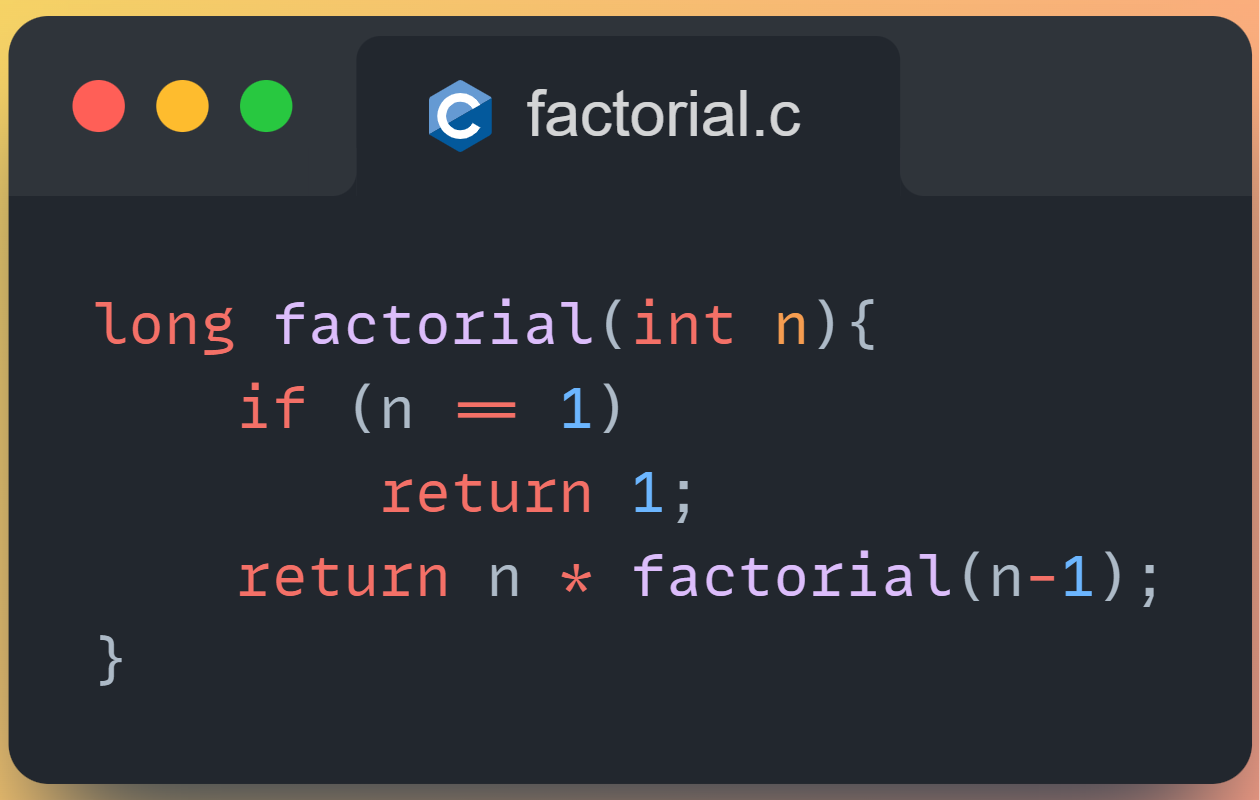
In most cases, it's recommended to avoid using goto and instead use more structured control flow constructs to make your code more readable and maintainable.

# Problem 2 (Factorial without recursion)

## Problem features

In most programming languages we have the elegant solution of having the algorithm that solves factorial by using recursion.

Code sample:



This previous sample uses recursion, this means that the function calls itself and we end up having tree like process in order to get to the solution, for now the time complexity of this is O(n) because we need to make n recursive calls.

## Proposed solution

We ended up using what is called as iterative solution for this program:

## Code

#include <stdio.h>

//Register to represent the stack frame

typedef struct {

int param;

int return\_address;

} StackFrame;

//must be fixed size we dont want to

//be dealing with malloc today

#define MAX\_STACK\_SIZE 100

StackFrame stack[MAX\_STACK\_SIZE];

int top = -1;

//stack push funct

void push(StackFrame frame) {

if (top >= MAX\_STACK\_SIZE - 1) {

printf("Stack overflow\n");

return;

}

stack[++top] = frame;

}

//pop funct

StackFrame pop() {

if (top < 0) {

printf("Stack underflow\n");

StackFrame error = {-1, -1};

return error;

}

return stack[top--];

}

//factorial function

int factorial(int N) {

//variables to store the result & temp frame

int FACT;

StackFrame TEMP\_REG;

//label for returning

int RETURN = 4;

//push initial frame

label1:

TEMP\_REG.param = N;

TEMP\_REG.return\_address = RETURN;

push(TEMP\_REG);

//this is the base case N = 0

label2:

if (N == 0) {

TEMP\_REG = pop();

FACT = 1;

RETURN = TEMP\_REG.return\_address;

goto label4;

//recursive case -> decrement N

} else {

N = N - 1;

RETURN = 3;

// another recursive call factorial

goto label1;

}

//process return frames

label3:

TEMP\_REG = pop();

N = TEMP\_REG.param;

// very important line ! (another recursive call)

FACT = N \* FACT;

RETURN = TEMP\_REG.return\_address;

label4:

if (RETURN == 3) goto label3;

return FACT;

}

//our main function where the user can calculate the factorial

//of the desire number

int main() {

int num;

printf("Enter a positive integer: ");

scanf("%d", &num);

printf("Factorial of %d = %d", num, factorial(num));

return 0;

}

## Code explanation

What is going on? First of all let’s say the user wants the factorial of 5, the algorithm has a “step by step” approach, first it sets up the stack and pushes this number 5 to the stack with the return address of 3 (this is because of goto) and checks if this number is 0, if this is not the case we subtract 1 to 5, and now we have 4 and we do the same until we reach this condition. After we finish then we start popping them off the stack and the popped elements get multiplied one by one.

## Sample case factorial 6 output



## Sample case factorial 7 output



## Why is recursion bad?

[[2]](#_References_&_Bibliography) recursion uses more memory, incurs greater call overhead and is sometimes harder to analyze for correctness compared to iterative solutions for the same problems. Other methods may help better with memory than using recursive nevertheless using a recursive method will be always be the most readable and elegant solution to most algorithms used in the real world.

# Problem 3 (tower of Hanoi (managing recursion))

## Approach

We as a team decided to focus our approach in this 3 points:

* Efficiency: to always remember that iteration is mostly more efficient than recursive approach.
* Newbie thought style: for most beginners (not us), it is easier to understand an iterative way rather than a recursive way.
* Clarity: The iterative approach tends to be more straightforward and easier to follow, making it more accessible and easier for a wider range of programmers.

## Code

NOTE: the first line with the code “#define \_CRT\_SECURE\_NO\_WARNINGS” helps to the compiler of the Visual Studio Professional (The purple visual studio) to avoid the warnings concerning security of strings with functions like scanf, for every other compiler, this line is not need it.

#define \_CRT\_SECURE\_NO\_WARNINGS

// C Program for Iterative Tower of Hanoi

// Erick, Alejandro & Andre

#include <stdio.h>

#include <math.h>

#include <stdlib.h>

#include <limits.h>

// A structure to represent a stack

struct Stack

{

unsigned capacity;

int top;

int\* array;

};

// function to create a stack of given capacity.

struct Stack\* createStack(unsigned capacity)

{

struct Stack\* stack =

(struct Stack\*)malloc(sizeof(struct Stack));

stack->capacity = capacity;

stack->top = -1;

stack->array =

(int\*)malloc(stack->capacity \* sizeof(int));

return stack;

}

// Stack is full when top is equal to the last index

int isFull(struct Stack\* stack)

{

return (stack->top == stack->capacity - 1);

}

// Stack is empty when top is equal to -1

int isEmpty(struct Stack\* stack)

{

return (stack->top == -1);

}

// Function to add an item to stack. It increases

// top by 1

void push(struct Stack\* stack, int item)

{

if (isFull(stack))

return;

stack->array[++stack->top] = item;

}

// Function to remove an item from stack. It

// decreases top by 1

int pop(struct Stack\* stack)

{

if (isEmpty(stack))

return INT\_MIN;

return stack->array[stack->top--];

}

//Function to show the movement of disks

void moveDisk(char fromPeg, char toPeg, int disk)

{

printf("Move the disk %d from \'%c\' to \'%c\'\n",

disk, fromPeg, toPeg);

}

// Function to implement legal movement between

// two poles

void moveDisksBetweenTwoPoles(struct Stack\* src,

struct Stack\* dest, char s, char d)

{

int pole1TopDisk = pop(src);

int pole2TopDisk = pop(dest);

// When pole 1 is empty

if (pole1TopDisk == INT\_MIN)

{

push(src, pole2TopDisk);

moveDisk(d, s, pole2TopDisk);

}

// When pole2 pole is empty

else if (pole2TopDisk == INT\_MIN)

{

push(dest, pole1TopDisk);

moveDisk(s, d, pole1TopDisk);

}

// When top disk of pole1 > top disk of pole2

else if (pole1TopDisk > pole2TopDisk)

{

push(src, pole1TopDisk);

push(src, pole2TopDisk);

moveDisk(d, s, pole2TopDisk);

}

// When top disk of pole1 < top disk of pole2

else

{

push(dest, pole2TopDisk);

push(dest, pole1TopDisk);

moveDisk(s, d, pole1TopDisk);

}

}

//Function to implement TOH puzzle

void tohIterative(int num\_of\_disks, struct Stack

\* src, struct Stack\* aux,

struct Stack\* dest)

{

int i, total\_num\_of\_moves;

char s = 'S', d = 'D', a = 'A';

//If number of disks is even, then interchange

//destination pole and auxiliary pole

if (num\_of\_disks % 2 == 0)

{

char temp = d;

d = a;

a = temp;

}

total\_num\_of\_moves = pow(2, num\_of\_disks) - 1;

//Larger disks will be pushed first

for (i = num\_of\_disks; i >= 1; i--)

push(src, i);

for (i = 1; i <= total\_num\_of\_moves; i++)

{

if (i % 3 == 1)

moveDisksBetweenTwoPoles(src, dest, s, d);

else if (i % 3 == 2)

moveDisksBetweenTwoPoles(src, aux, s, a);

else if (i % 3 == 0)

moveDisksBetweenTwoPoles(aux, dest, a, d);

}

}

int main()

{

// Input: number of disks

unsigned int num\_of\_disks = 0;

printf("number of disks:");

scanf("%u", &num\_of\_disks);

printf("pegs names: 's' for source 'a' for aux 'd' for destination\n");

struct Stack\* src, \* dest, \* aux;

// Create three stacks of size 'num\_of\_disks'

// to hold the disks

src = createStack(num\_of\_disks);

aux = createStack(num\_of\_disks);

dest = createStack(num\_of\_disks);

tohIterative(num\_of\_disks, src, aux, dest);

return 0;

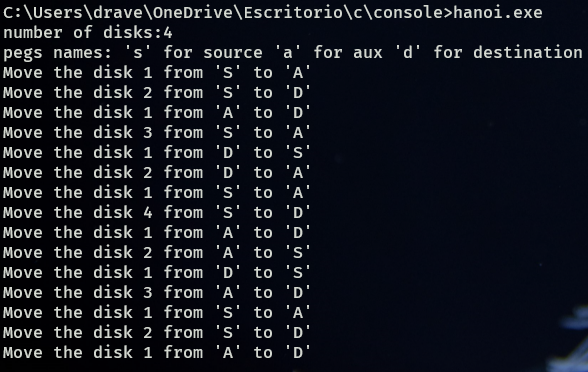
}

## Code explanation

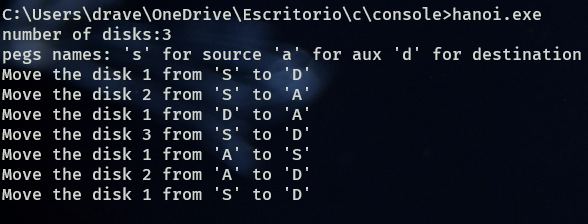
So, we have the three sticks Source, Auxiliar & Destination, we keep track of all the discs with a stack, the moveDisk function is to display the movement of the disks from one peg to another, the moveDisksBetweenTwoPoles implements the logic for moving a disk between two pegs, where the correct conditions are checked in order to take a decision, the tohIterative function solves the Tower of Hanoi problem iteratively. It takes the number of disks and the three pegs (source, auxiliary, and destination) as input and calculates the total number of moves required. It then iteratively calculates and executes the moves based on the total number of moves and the current step in the process.

IMPORTANT: limits.h header is used in the pop function due to the “INT\_MIN” constant that indicates a state of error when popping from an empty stack, and math.h is used for the pow (power of a number) function in line 120 of the code.

## Sample output (1)



## Sample output (2)



We can prove that both sample outputs are correct.

# Conclusion

Managing recursion is not a bad thing, nevertheless if we don’t require speed and we don’t need to worry to much about the memory we can always rely on recursive approaches, that makes more elegant and powerful solutions across many algorithms used to sort, select, or connect data. It is important to mention that *goto* is a very bad approach, not only we as a team agree that it make the code harder to understand but also is not efficient.

# References & Bibliography

[[1]](#_GOTO_statement_definition) TylerMSFT. (2023, 25 enero). GOTO and Labeled Statements (C). Microsoft Learn. https://learn.microsoft.com/en-us/cpp/c-language/goto-and-labeled-statements-c?view=msvc-170&viewFallbackFrom=vs-2019

[[2]](#_Why_is_recursion) Charles. (2017, 24 February). Recursion -The good, the bad, and the not so ugly. Coding Ninjas Blog. https://www.codingninjas.com/blog/2017/02/24/recursion-the-good-the-bad-and-the-not-so-ugly/#:~:text=A%20lot%20of%20programmers%20avoid,efficient%20than%20its%20iterative%20counterparts.&text=space%20is%20carved%20out%20on%20the%20stack%20for%20function%20arguments%20and%20variables.