Competitive Programming

Recursion

B.Bhuvaneswaran, AP (SG) / CSE

9791519152

bhuvaneswaran@rajalakshmi.edu.in

RAJALAKSHMI ENGINEERING COLLEGE

An AUTONOMOUS Institution

Affiliated to ANNA UNIVERSITY, Chennai

Introduction

- Recursion is an important concept in computer science.
- It is a foundation for many other algorithms and data structures.
- However, the concept of recursion can be tricky to grasp for many beginners.

Questions?

- What is recursion? How does it work?
- How to solve a problem recursively?
- How to analyze the time and space complexity of a recursive algorithm?
- How can we apply recursion in a better way?

Principle of Recursion

- Recursion is an approach to solving problems using a function that calls itself as a subroutine.
- The trick is that each time a recursive function calls itself, it reduces the given problem into subproblems.
- The recursion call continues until it reaches a point where the subproblem can be solved without further recursion.

Principle of Recursion

- A recursive function should have the following properties so that it does not result in an infinite loop:
  - A simple base case (or cases) — a terminating scenario that does not use recursion to produce an answer.
  - A set of rules, also known as recurrence relation that reduces all other cases towards the base case.
- Note that there could be multiple places where the function may call itself.

Types of Recursions

- Direct recursion
- Indirect recursion

Direct Recursion

- In the direct recursion, the same function calls itself from its own body.

Indirect Recursion

- In indirect recursion the function invokes some other function which again invokes the function which has invoked it.
- That is, it creates a cycle of function calls.

Example

- An useful example of recursion is the evaluation of factorials of a given number.
- The factorial of a number n is expressed as a series of repetitive multiplications as shown below:

  factorial of n = n(n–1)(n–2) . . . 1

- For example:

  factorial of 4 = 4 × 3 × 2 × 1 = 24

Program

```
int fact(int n)
{
        if(n ==0 || n == 1)
                return 1;
        else
                return(n * fact(n - 1));
}
```

Explanation

- Let us see how the recursion works. Assume n = 5. Since the value of n is not 1, the statement:

  fact = n * factorial(n-l);

- will be executed with n = 5. That is,

  fact = 5 * factorial(4);

- will be evaluated. The expression on the right-hand side includes a call to factorial with n = 4. This call will return the following value:

  4* factorial(3)

- Once again, factorial is called with n = 3. That is,

  3 * factorial(2);

- will be evaluated. The expression on the right-hand side includes a call to factorial with n = 2. This call will return the following value:

  2 * factorial(1);

- Once again, factorial is called with n = 1. This time, the function returns 1.

Explanation

- The sequence of operations can be summarized as follows:

fact = 5 * factorial(4)

= 5 * 4 * factorial(3)

= 5 * 4 * 3 * factorial(2)

= 5 * 4 * 3 * 2 * factorial(1)

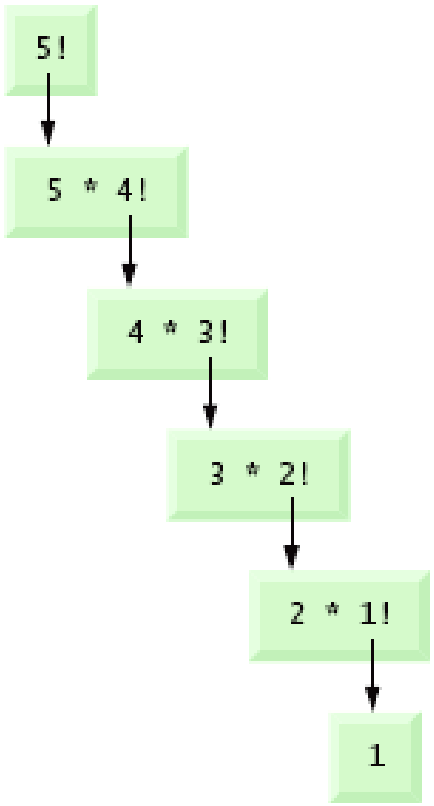
= 5 * 4 * 3 * 2 * 1

= 120

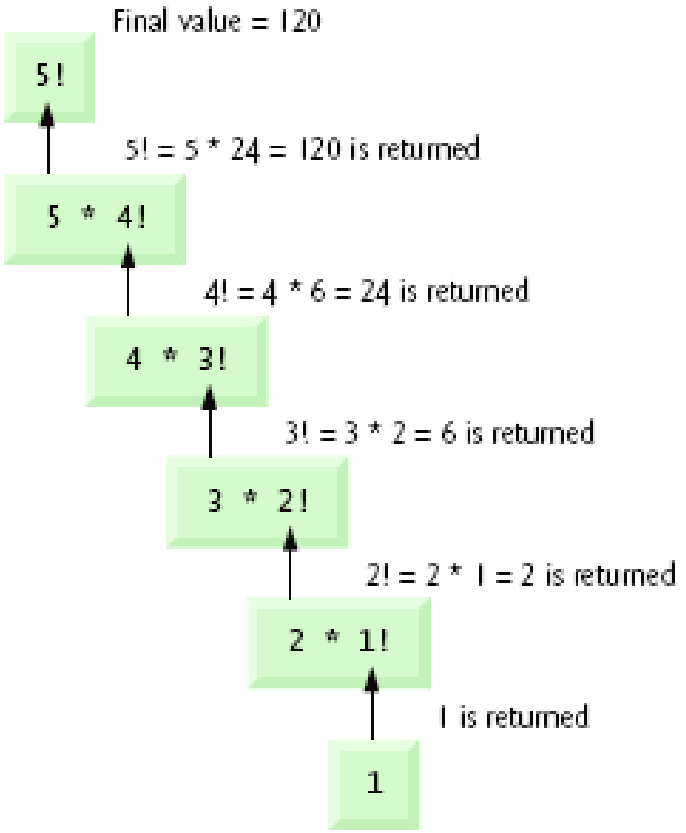
Explanation

- Recursive functions can be effectively used to solve problems where solution is expressed in terms of successively applying the same solution to subsets of the problem.
- When we write recursive functions, we must have an if statement somewhere to force the function to return without the recursive call being executed.
- Otherwise, the function will never return.

Recursive evaluation of 5!



(a) Sequence of recursive calls.



(b) Values returned from each recursive call.

Program

```
#include <stdio.h>
int fact(int);
int main()
{
        int n, f = 1;
        scanf("%d", &n);
        f = fact(n);
        printf("%d", f);
        return 0;
}
```

Program

```
int fact(int n)
{
        if(n ==0 || n == 1)
                return 1;
        else
                return(n * fact(n - 1));
}
```

Output

5

120

Sum of First n Natural No.s (Bottom up – Iteration)

```
#include <stdio.h>

int main()
{
        int n, i, sum = 0;
        scanf("%d", &n);
        for (i = 1; i <= n; i++)
                sum = sum + i;
        printf("%d", sum);
        return 0;
}
```

Sum of First n Natural Nos. (Top down – Recursion)

```
#include <stdio.h>

int sum(int n)
{
        if (n == 1)
                return 1;
        return sum(n - 1) + n;
}

int main()
{
        int n;
        scanf("%d", &n);
        printf("%d", sum(n));
        return 0;
}
```

Print n^{th} Fibonacci Term (Recursion)

```
#include <stdio.h>

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

int main()
{
        int n;
        scanf("%d", &n);
        printf("%d", fib(n));
        return 0;
}
```

Print a string in reverse order

- You can easily solve this problem iteratively, i.e. looping through the string starting from its last character. But how about solving it recursively?
- First, we can define the desired function as printReverse(str[0...n-1]), where str[0] represents the first character in the string. Then we can accomplish the given task in two steps:
  - printReverse(str[1...n-1]): print the substring str[1...n-1] in reverse order.
  - print(str[0]): print the first character in the string.
- Notice that we call the function itself in the first step, which by definition makes the function recursive.

Code Snippet

```
void printReverse(const char *str)
{
        if (!*str)
                return;
        printReverse(str + 1);
        putchar(*str);
}
```

Time Complexity - Recursion

- You can easily solve this problem iteratively, i.e. looping through the string starting from its last character. But how about solving it recursively?
- First, we can define the desired function as printReverse(str[0...n-1]), where str[0] represents the first character in the string. Then we can accomplish the given task in two steps:
  - printReverse(str[1...n-1]): print the substring str[1...n-1] in reverse order.
  - print(str[0]): print the first character in the string.
- Notice that we call the function itself in the first step, which by definition makes the function recursive.

Recursion to Iteration

- Sometimes it is desirable to implement the algorithm with iteration instead of recursion, due to the constraint of memory consumption or efficiency.

Unfold Recursion

- Recursion could be an elegant and intuitive solution, when applied properly.
- Nevertheless, sometimes, one might have to convert a recursive algorithm to iterative one for various reasons.

Risk of Stackoverflow

- The recursion often incurs additional memory consumption on the system stack, which is a limited resource for each program.
- If not used properly, the recursion algorithm could lead to stackoverflow.
- One might argue that a specific type of recursion called tail recursion could solve this problem.
- Unfortunately, not every recursion can be converted to tail recursion, and not every compiler supports the optimization of the tail recursion.

Efficiency

- Along with the additional memory consumption, the recursion could impose at least the additional cost of function calls, and in a worse case duplicate calculation, i.e. one of the caveats of recursion.

Complexity

- The nature of recursion is quite close to the mathematics, which is why the recursion appears to be more intuitive and comprehensive for many people.
- However, when we abuse the recursion, the recursive program could become more difficult to read and understand than the non-recursive one, e.g. nested recursion etc.

Recursion vs Iteration

Criteria	Recursion	Iteration
Definition	When a function calls itself directly or indirectly.	When some set of instructions are executed repeatedly.
Implementation	Implemented using function calls.	Implemented using loops.
Format	Base case and recursive relation are specified.	Includes initializing, control variable, termination condition, and update of the control variable.
Current State	Defined by the parameters stored in the stack.	Defined by the value of the control variable.
Progression	The function state approaches the base case.	The control variable approaches the termination value.

Recursion vs Iteration

Criteria	Recursion	Iteration
Memory Usage	Uses stack memory to store local variables and parameters.	Does not use memory except initializing control variables.
Infinite Repetition	It will cause stack overflow error and may crash the system if the base case is not defined or is never reached.	It will cause an infinite loop if the control variable does not reach the termination value.
Code Size	Recursive code is generally smaller and simpler.	Iterative code is generally bigger.
Overhead	Possesses overhead of repeated function call.	No overhead as there are no function calls.
Speed	Slower in execution.	Faster in execution

Queries?

Thank You…!