rtp

week 4_1:

recursion

rajin teaches programming

INTRODUCTION TO RECURSION:

Recursion is a **powerful programming technique** where a **function** calls **itself** to solve a problem. It involves **breaking down** a problem into **smaller subproblems**, solving **each subproblem recursively**, and **combining** the results to obtain the **final** solution. This technique can **simplify** the code for **problems** that have a **natural** recursive structure, such as **mathematical computations**, **data structure manipulations**, and **algorithmic** processes.

– IMPORTANCE:

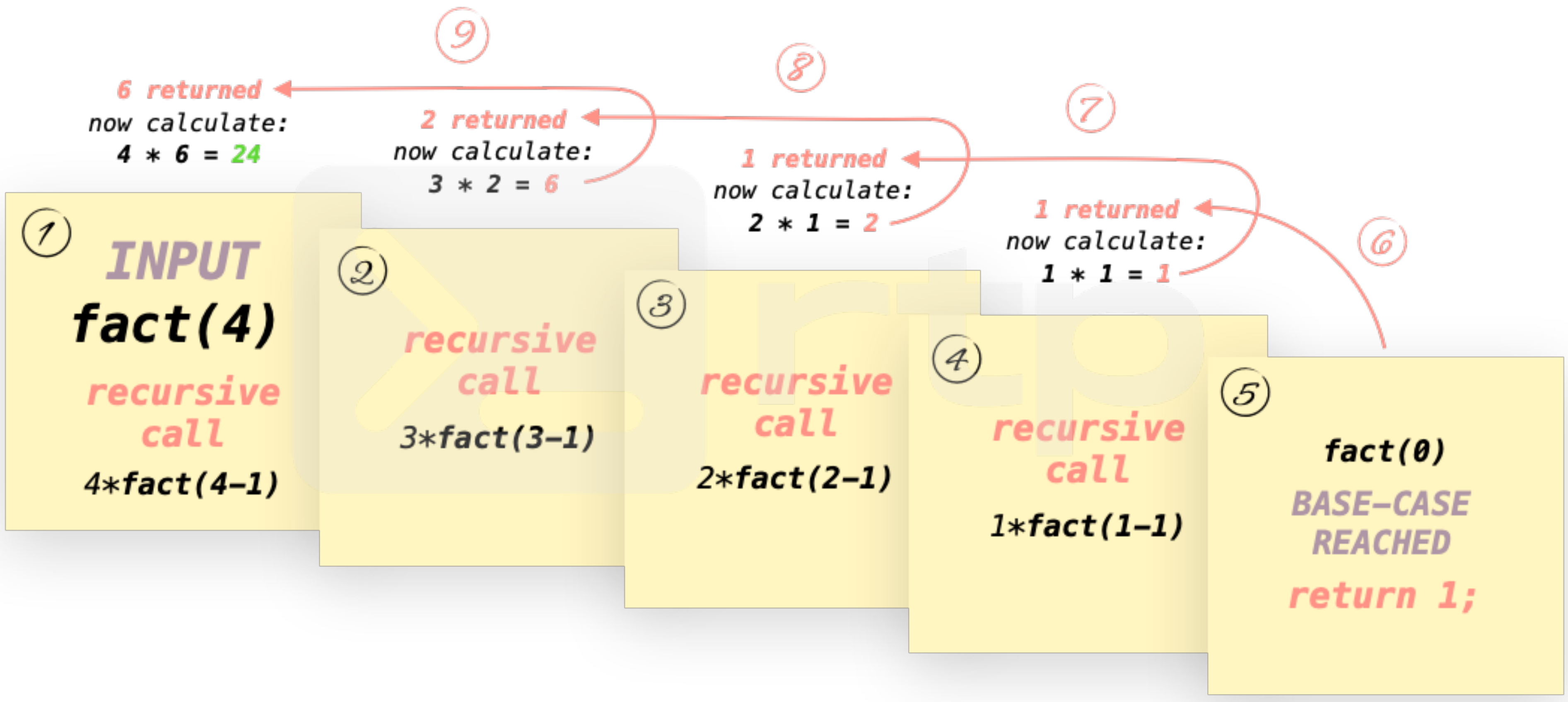
Recursion is **crucial** in C programming for several reasons. First, it provides a **clean** and **simple** way to write code for **complex** problems. Furthermore, many algorithms are **naturally recursive**, making recursion an **intuitive** approach. Finally, diving into a very core concept of programming, being **Divide and Conquer**, problems are **broken down** into **smaller**, more **manageable sub-problems.**

RECURSION USAGE:

Recursion occurs when a function **calls itself** to solve **smaller instances** of the **same** problem. **Each recursive call** should bring the problem **closer** to a **base case**, which is the **condition** that **stops** the **recursion.**

BUT HOW DOES IT WORK?

*Let's try finding the **factorial of 4** as an example.*



- STEPS:

We must first **define** the **Base Case,** which is the **condition** under which the **recursion stops.**

Then we must define the **Recursive Case,** which is the **part** of the **function** that **calls itself** with a **smaller** or **simpler input.**

RECURSION SYNTAX

```
return_type function_name(parameters) {

    if (base_case_condition) {

        // Base case: stop recursion
        return base_case_value;
    }
    else {

        // Recursive case: call function again
        return recursive_case;
    }
}
```

Developer - fact.c

```c
#include<stdio.h>

int fact(int n) {

    if (n==0) {

        return 1; //base case
    }
    else {

        return n * fact(n-1); //recursive case
    }
}

int main() {

    int num;

    printf("Enter a number to find its factorial: ");
    scanf("%d", &num);

    printf("%d! = %d\n", num, fact(num)); //function being called

    return 0;
}
```

example of the recursive factorial function

Developer - fibrecur.c

```c
#include <stdio.h>

int fibonacci(int n) {

    if (n <= 1) { // Base cases

        return n;

    }
    else {

        return fibonacci(n - 1) + fibonacci(n - 2); // Recursive case
    }
}

int main() {

    for (int i=0; i<5; i++) {

        printf("%d\t", fibonacci(i));
    }

    return 0;
}
```

example of a recursive fibonacci function

Developer - sumarrecur.c

```c
#include <stdio.h>

int sum(int arr[], int n) {

    if (n <= 0) { // Base case

        return 0;
    }
    else {

        return arr[n - 1] + sum(arr, n - 1); // Recursive case
    }
}

int main() {

    printf("Enter the size of your array: ");

    int size;
    scanf("%d", &size);

    int array[size];

    for (int i=0; i<size; i++) {

        printf("Enter the number at index[%d]", i);
        scanf("%d", &array[i]);
    }

    printf("Sum of array elements is %d\n", sum(array, size));

    return 0;
}
```

example of a recursive function to find sum of elements in an array

MEMORY USAGE FOR RECURSION:

It is important to realize that **every time** a **recursive call** is made, a **space** is **allocated** in the computer **memory** where the process is **stored**, waiting for an answer. Then, the **next** recursive call is made, “**pushing**” another “**frame**” on to the **stack.** These **frames** are **pushed** onto the **stack** as long as **recursive calls** are made, sort of like how you would “**stack**” plates **on top** of each other. Finally, when the **base case** is reached, an **answer** is found, and the **stack frame** is “**popped**” (**removed** from the **memory**). Then, using this answer, the **previous frame** is **popped** from the **stack**, and so on, until the answer value reaches the very **first frame** that was **pushed** onto the **stack**, and we receive our **answer.** A **visualization** in class will be provided by me.

RECURSION PITFALLS:

Keeping the previous concept in mind, there may be times when our code has an **error** and the **base case** is **never reached.** This can lead to **infinite recursion.** What this means in terms of **memory**, is that a **frame** keeps being **pushed** on to the **stack** one by one, **until** there is **no space left** in the **stack.** This **excessive stack usage** is often referred to as a **stack overflow** (yes, this is where the name of the website comes from). Maybe now the logo for the **stack overflow** website makes more sense.

stack overflow’s logo

CONCLUSION:

Recursion can be **less efficient** than **iterative** solutions due to **function call overhead** and **increased memory usage.** However, the **convenience** and **simplicity** it offers allows programmers to solve **complex** problems with **ease.** Understanding recursion is **essential** for writing efficient and **concise** code. Practice implementing **recursive** solutions to **various** problems to **master** this concept of thinking of **larger** problems as **smaller sub-problems.**

SOME FAQs:

What is the difference between recursion and iteration?
Recursion uses function calls to repeat code, whereas iteration uses loops. Recursion can be more elegant for certain problems, but iteration is generally more efficient in terms of memory and performance.

How do I know if a problem can be solved with recursion?
Problems that can be broken down into smaller, similar sub-problems are good candidates for recursion. Examples include factorials, Fibonacci numbers, and tree traversals.

Can all recursive functions be converted to iterative ones?
Yes, any recursive function can be converted to an iterative one, although the iterative version may be more complex and less intuitive.

How can I avoid stack overflow with deep recursion?
To avoid stack overflow, ensure that the recursion depth is limited or use an iterative solution. Additionally, some problems can be solved using tail recursion, which is more memory efficient.

next class 4_2:

recursion problems

rajin teaches programming

reach out for classes at: rajin.khan2001@gmail.com