CS 2124: DATA STRUCTURES
Spring 2024

4th Lecture (Part – I)

Topics: **Recursion**

Topics

- Assignment – 2 (Any Questions)
- Mid-Term Exam (Discussion)
- Recursion
  - Recursion (Properties)
  - Recursion (Types)
- Recursion vs Iteration
  - Example using Factorial
  - Example using Fibonacci Sequence
- Recursion (Memory)
- Recursion (Advantages and Disadvantages)
- Recursion (Real world examples)
- Binary Search (Using Recursion and Iteration)
- Towers of Hanoi

Midterm Exams

- Data Structure (Midterm Exam – In person) – Thursday, 29th Feb
  - Exam will be on Canvas
  - Attendance is compulsory
  - Location: NPB 1.226
  - Timing:

Section			**Time/ NPB 1.226**	**Students**
CS 2124 - 0C1	CS 2124-0CA	36734	10:00 – 10:30	30
	CS 2124-0CB	36736	10:40 – 11:10	29
CS 2124 - 0D4	CS 2124-0DA	36738	11:30 – 12:00	30
	CS 2124-0DB	36739	12:10 – 12:40	30
CS 2124 - 0E1	CS 2124-0EA	42879	01:30 – 02:00	30
	CS 2124-0EB	42880	02:10 – 02:40	30

Recursion

- **Recursion** in data structure is when a function calls itself indirectly or directly, and the function calling itself is known as a recursive function.
- It's generally used when the answer to a larger issue could be depicted in terms of smaller problems.
- A **function** is called ‘recursive’ if a statement within the body of a function calls the same function. Sometimes called ‘circular definition’, recursion is thus the process of defining something in terms of itself.
- Recursion algorithm are based on divide and conquer principle to conquer problem.

Recursion (Properties)

- A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have
  1. **Base criteria** − There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
  2. **Progressive approach** − The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

- **Examples:** We can use recursive functions for problems such as Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.
- Recursion uses more memory, because the recursive function adds to the stack with each recursive call, and keeps the values there until the call is finished.

Recursion

```c
#include <stdio.h>
void recursion()
{
    int i=0;
    while (i<=5)
    {
    printf("Function");
    recursion(); /* function calls itself */
    i--;
    }
}
int main() {
    printf("Main");
    recursion();
    printf("Back in Main");
}
```

- Base criteria ?
- Progressive approach ?

Recursion (Types)

1. **Direct Recursion**
   - Direct recursion in C occurs when a function calls itself directly from inside. Such functions are also called direct recursive functions.
2. **Indirect Recursion**
   - Indirect recursion in C occurs when a function calls another function and if this function calls the first function again. Such functions are also called indirect recursive functions.

- function_01()
- {
- //some code
- function_01();
- //some code
- }

- function_01()
- {
- //some code
- function_02();
- }

- function_02()
- {
- //some code
- function_01();
- }

Recursion

```c
#include <stdio.h>

void recurse ( int count )
{
    printf( "Count is: %d\n", count );
    if (count > 9)
    {
    return;
    }
    else
    recurse ( count + 1 );
}

int main()
{
  recurse ( 1 );
  return 0;
}
```

- Try to identify:
  1. Error in Program
  2. Infinite Loop
  3. Base criteria ?
  4. Progressive approach ?

Recursion (What do you think about these codes?)

A

```c
#include <stdio.h>
void count_to (int count)
{
    printf("In recursion: %d \n", count);
if (count < 2)
{
    printf("In if statement 1: %d \n", count);
    count_to(count+1);
    printf("In if statement 2: %d \n", count-1);
}
}
int main()
{
    count_to(0);
    printf("Main Function \n");
    return 0;
}
```

B

```c
#include <stdio.h>
void count_to (int count)
{
    printf("In recursion: %d \n", count);
if (count < 2)

    printf("In if statement 1: %d \n", count);
    count_to(count+1);
    printf("In if statement 2: %d \n", count-1);

}
int main()
{
    count_to(0);
    printf("Main Function \n");
    return 0;
}
```

Output A:

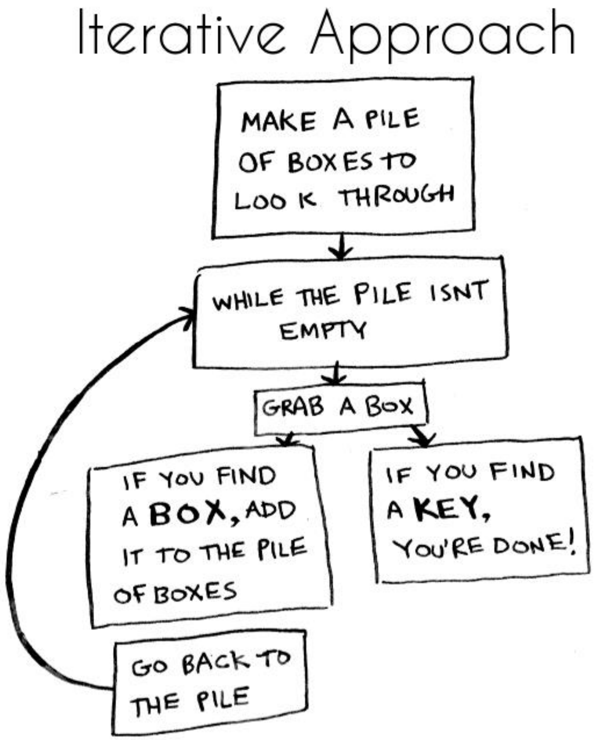
Output B:

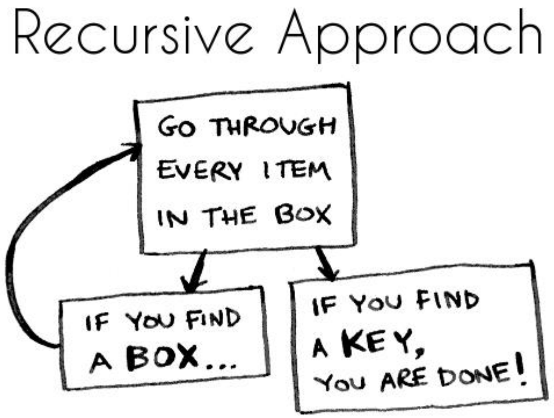
Options:

A. Both Program are same i.e. same output
B. Program A infinite loop
C. Program B infinite loop
D. Program A will run Program B will have a warning

Recursion vs Iteration

- **Iterative:** Use a explicit series of steps to solve a problem using loops and conditional statements or loops to repeat some part of the code.
- **Recursive:** Solve a problem by reducing it to a smaller version of itself. Eventually reach a base condition is reached and the recursion stops or calls itself again to repeat the code





Recursion vs Iteration

- **When to Use Recursion?**

- **When to Use Iteration?**

```python
a = 1
while a < 7 :
    if(a % 2 == 0):
        print(a, "is even")
    else:
        print(a, "is odd")
    a += 1
```

code

output

variables

Recursion vs Iteration

- For issues that can be broken down into several, smaller pieces, recursion is far superior to iteration. Using **recursion** in the divide and conquer method can minimize the size of your problem at each step and take less time than a naive iterative approach.
- **Iteration** can be used to repeatedly execute a set of statements without the overhead of function calls and without using stack memory. Iteration is faster and more efficient than recursion.

Recursion vs Iteration

	Recursion	Iteration
Time Complexity	Very high(generally exponential) time complexity.	Relatively lower time complexity(generally polynomial-logarithmic).
Usage (Time VS Complexity)	If time complexity is not an issue and shortness of code is, recursion would be the way to go.	If time complexity is the point of focus, and number of recursive calls would be large, it is better to use iteration
Over Head	Because of the overhead of maintaining the stack, the recursion process is usually slower than iteration and needs more memory	Iteration consumes less memory, but makes the code longer which is difficult to read and write
Infinite Repetition	Infinite recursive calls may lead to system CPU crash.	Infinite iteration may or may not lead to system errors, but will surely stop program execution as memory exhaust.

Recursion vs Iteration

(Example using factorial – Which of the following is fast)

```c
#include <stdio.h>
#include <time.h>
int factorial(unsigned int i)
{
    if (i<=1)
    {
        return 1;
    }
    return i*factorial (i-1);
}
int main()
{
    clock_t start_t, end_t;
    double total_t;
    int i=5;
    start_t = clock();
    printf("%d Factorial is: %d", i, factorial(i));
    end_t = clock();
    total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
printf("\nCPU Cycle/time (Recursive): %f\n", total_t);
    return 0;
}
```

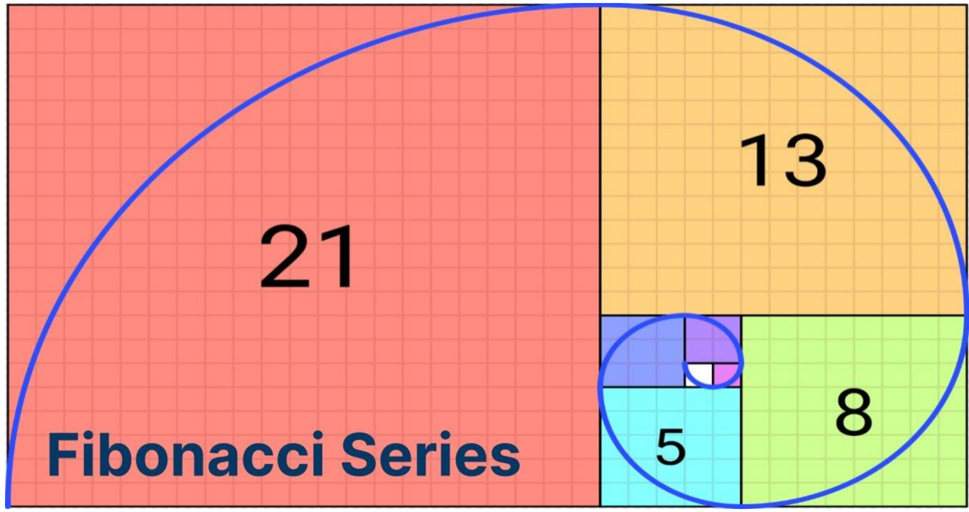
```c
#include <stdio.h>
#include <time.h>
int main()
{
    clock_t start_t, end_t;
    double total_t;
    int i, num=5, factorial=1;
    start_t = clock();
    for(i=1; i<=num; i++)
    {
        factorial=factorial*i;
    }
    printf("%d Factorial is: %d", num, factorial);
    end_t = clock();
    total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
   printf("\n CPU cycle/time (Iterative): %f\n", total_t);
    return 0;
}
```

Will there be difference in time or it will be the same (almost)?

Recursion vs Iteration
(Example using Fibonacci Sequence)

The Fibonacci Sequence

- $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$
- Each element = sum of two preceding Fibonacci elements
  - (Except for 0 and 1)
- For example, fib(6) = fib(4) + fib(5)



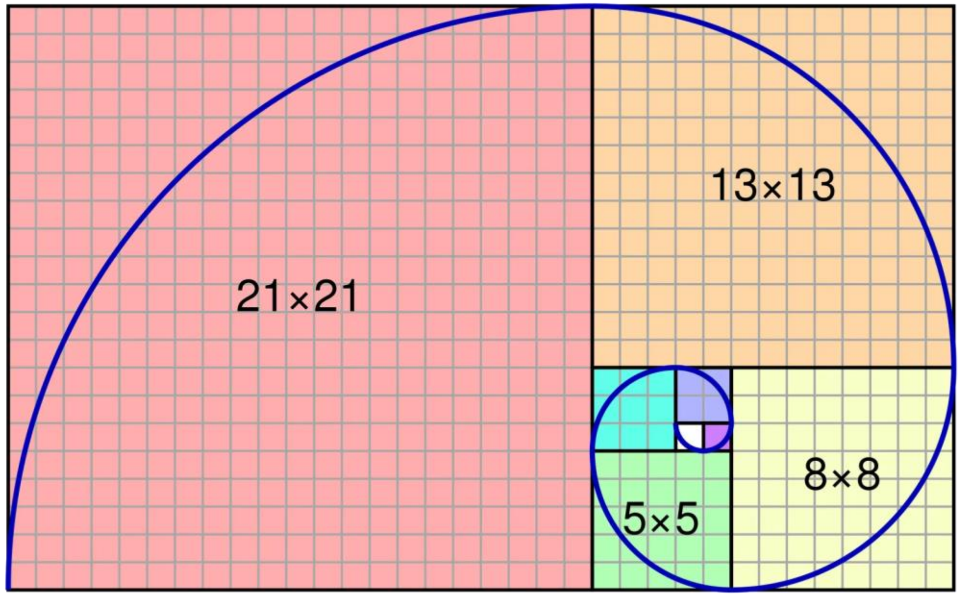


Image Sources: Link, Link

Recursion vs Iteration (Example using Fibonacci Sequence)

1. Add integer value to num variable.
2. Then initialize two variables n1 and n2 with values 0 and 1.
3. Check if the num value is equal to 1, then print n1 only.
4. Else
   I. Print the value of n1 and n2.
   II. Then run a for loop from range(i = 2; i < num; i++) and inside the for loop perform the following operations.
      i. Initialize n3 with value of n1 + n2.
      ii. Update n1 to n2.
      iii. Update n2 to n3.
      iv. At last print n3.

Recursion vs Iteration
(Example using Fibonacci Sequence)

```
#include<stdio.h>
#include <time.h>
int main(void) {
    int num = 10;
    int n1 = 0, n2 = 1, i, n3;
    clock_t start_t, end_t;
    double total_t;
    start_t = clock();
    if (num==1){
        printf("%d", n1);
    }
    else{
        printf("%d, %d, ", n1, n2);
        for(i = 2; i < num; i++){
            n3 = n1 + n2;
            n1 = n2;
            n2 = n3;
            printf("%d, ",n3);
        }
    }
    end_t = clock();
    total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
    printf("\nCPU Cycle/time (Iteration): %f\n", total_t);
}
```

**This is an online implementation so CPU cycles may also depend on other factures. i.e. internet, server load, location etc.*

Recursion vs Iteration
(Example using Fibonacci Sequence)

1. Add integer value to num.
2. Then run a for loop from (i = 0; i < num; i++).
3. In each iteration print and call the fibonacciSeries function with i as a parameter.
4. In the Recursive function fibonacciSeries,
5. Check if i <= 1, if it is True then return i
6. Else return fibonacci(i - 1) + fibonacci(i - 2).

Recursion vs Iteration
(Example using Fibonacci Sequence)

```c
#include <stdio.h>
#include <time.h>
int fibonacci(i)
{
    if (i <= 1)
        return i;
    else
        return (fibonacci(i - 1) + fibonacci(i - 2));
}
int main(void) {

    int num = 10, i;
    clock_t start_t, end_t;
    double total_t;
    start_t = clock();
    for(i = 0; i < num; i ++){
        printf("%d ", fibonacci(i));
    }
    end_t = clock();
    total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
    printf("\nCPU Cycle/time(Recursion): %f\n", total_t);
   }
```

Recursion vs Iteration
(Example using Fibonacci Sequence)

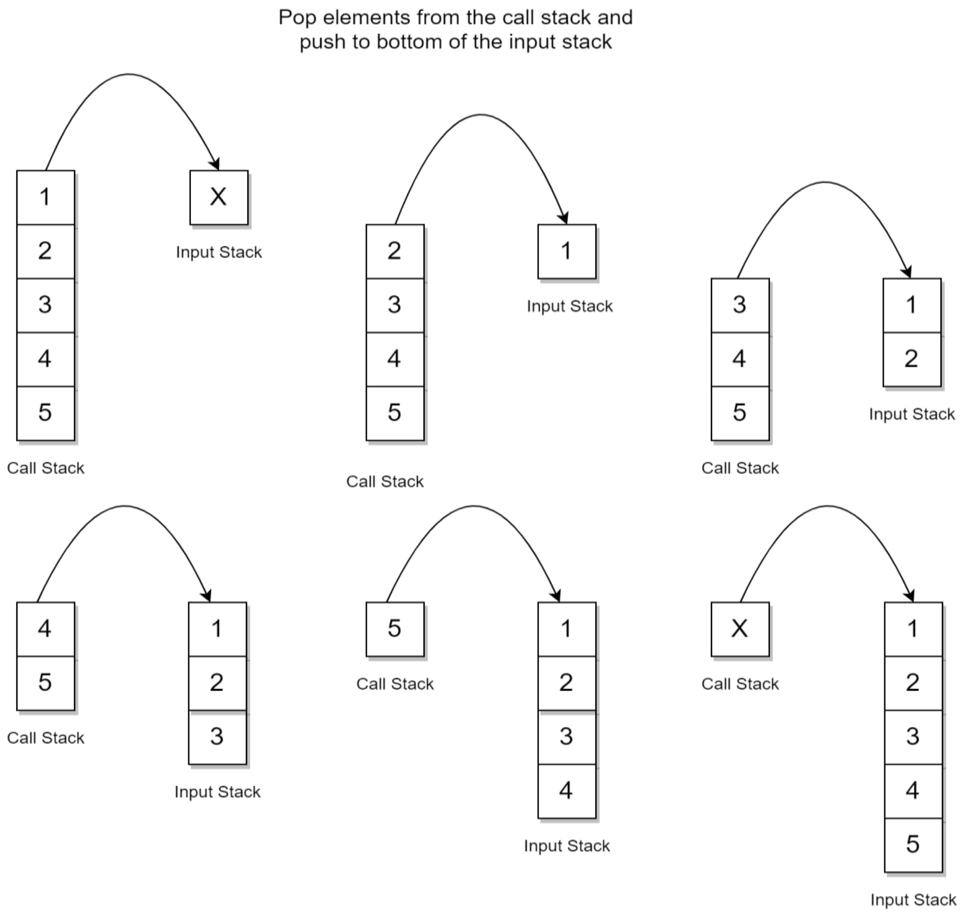
```c
#include <stdio.h>
#include <time.h>
int fibonacci(int i)
{
    if (i <= 1)
        return i;
    else
        return (fibonacci(i - 1) + fibonacci(i - 2));
}
int main(void) {

    int num = 10, i;
    clock_t start_t, end_t;
    double total_t;
    start_t = clock();
    for(i = 0; i < num; i ++){
        printf("%d ", fibonacci(i));
    }
    end_t = clock();
    total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
    printf("\nCPU Cycle/time(Recursion): %f\n", total_t);
   }
```

Source: [Link]()

**This is an online implementation so CPU cycles may also depend on other factures. i.e. internet, server load, location etc.*

Recursion (Memory)

- Since recursion is a repetition of a particular process and has so much complexity, the stack is maintained in memory to store the occurrence of each recursive call.
- Each recursive call creates an activation record(copy of that method) in the stack inside the memory when recursion occurs.
- Once something is returned or a base case is reached, that activation record is de-allocated from the stack, and that stack gets destroyed.
- Each recursive call whose copy is stored in a stack stored a different copy of local variables declared inside that recursive function.



Source: Link

Recursion (Memory)

```c
#include <stdio.h>
int rfunc (int a)  //2) recursive function
{
    if(a == 0)
        return 0;
    else
    {
     printf("Digit: %d, Address: %p \n",a, &a);
     //Print number and its address
     return rfunc(a-1); // 3) recursive call is made
    }
}
int main()
{
   rfunc(5); // 1) function call from main
   return 0;
}
```

- What will be the output ?
- Which element will be at the top and which element will be at the bottom of stack/output?

Recursion (Memory)

```c
#include <stdio.h>
int rfunc (int a)  //2) recursive function
{
    if(a == 0)
        return 0;
    else
    {
     printf("Digit: %d, Address: %p \n",a, &a);
     //Print number and its address
     return rfunc(a-1); // 3) recursive call is made
    }
}
int main()
{
   rfunc(5); // 1) function call from main
   return 0;
}
```

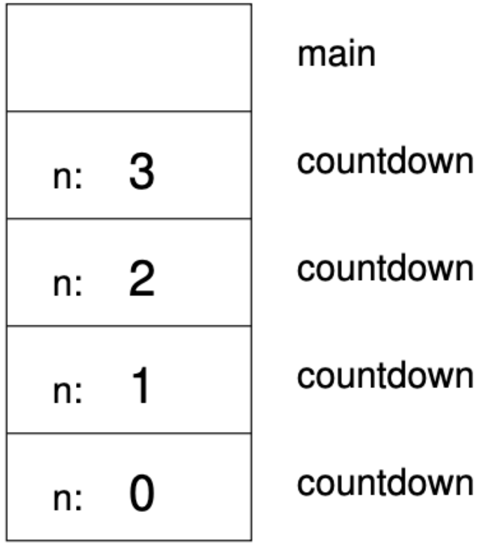


Image source : [link]()

1. The first call to the function rfunc() having value a=5 will be a copy on the bottom of the stack, and it is also the copy that will return at the end.
2. Meanwhile, the rfunc() will call another occurrence of the same function but with 1 subtracted, i.e., a=4.
3. Each time a new occurrence is called, it is stored at the top of the stack, which goes on until the condition is satisfied.
4. As the condition is unsatisfied, i.e., a=0, there will be no further calls, and each function copy stored in the stack will start to return its respected values, and the function will now terminate.

Recursion (Memory)

```c
#include <stdio.h>
int rfunc (int a)  //2) recursive function
{
    if(a == 0)
        return 0;
    else
    {
     printf("Digit: %d, Address: %p \n",a, &a);
     //Print number and its address
     return rfunc(a-1); // 3) recursive call is made
    }
}
int main()
{
   rfunc(5); // 1) function call from main
   return 0;
}
```

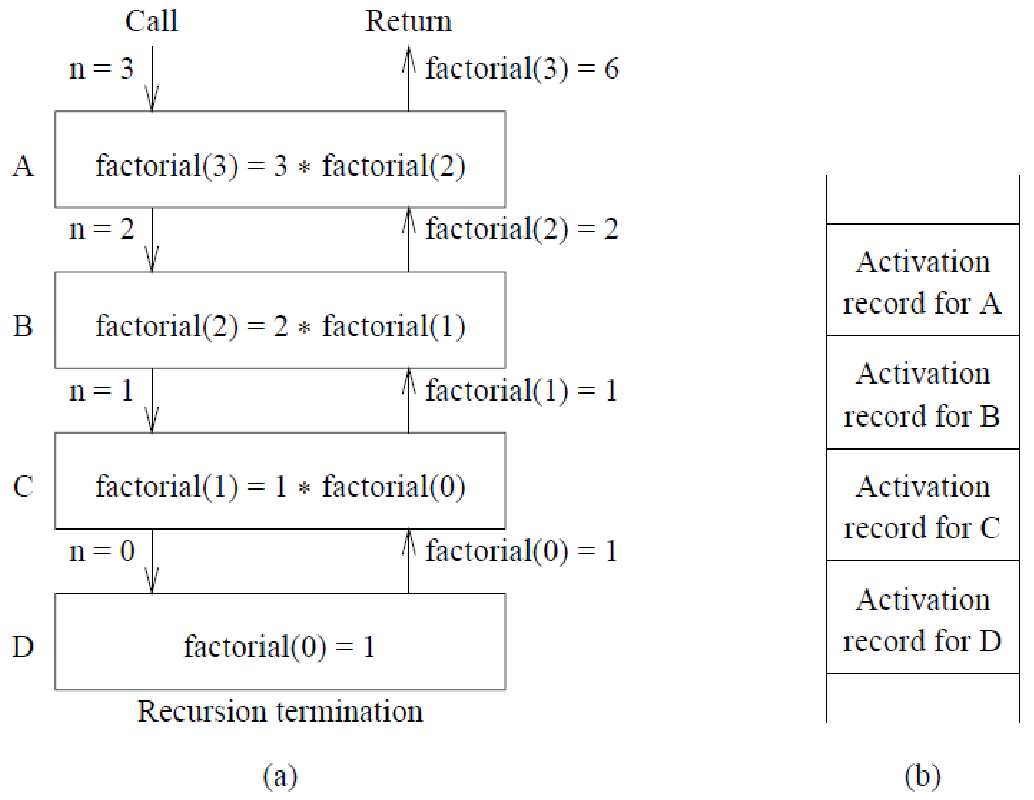


Image source : link

1. The first call to the function rfunc() having value a=5 will be a copy on the bottom of the stack, and it is also the copy that will return at the end.
2. Meanwhile, the rfunc() will call another occurrence of the same function but with 1 subtracted, i.e., a=4.
3. Each time a new occurrence is called, it is stored at the top of the stack, which goes on until the condition is satisfied.
4. As the condition is unsatisfied, i.e., a=0, there will be no further calls, and each function copy stored in the stack will start to return its respected values, and the function will now terminate.

Recursion

- **Advantages:**
  - The code becomes shorter and reduces the unnecessary calling to functions.
  - Useful for solving formula-based problems and complex algorithms.
  - Useful in Graph and Tree traversal as they are inherently recursive.
  - Recursion helps to divide the problem into sub-problems and then solve them, essentially divide and conquer.
- **Disadvantages:**
  - The code becomes hard to understand and analyze.
  - A lot of memory is used to hold the copies of recursive functions in the memory.
  - Time and Space complexity is increased.
  - Recursion is generally slower than iteration.

End of Lecture
Midterm Exams (Reminder)

- Data Structure (Midterm Exam – In person) – Thursday, 29th Feb
  - Exam will be on Canvas
  - Attendance is compulsory
  - Location: NPB 1.226
  - Timing:

Section			Time/ NPB 1.226	Students
CS 2124 - 0C1	CS 2124-0CA	36734	10:00 – 10:30	30
	CS 2124-0CB	36736	10:40 – 11:10	29
CS 2124 - 0D4	CS 2124-0DA	36738	11:30 – 12:00	30
	CS 2124-0DB	36739	12:10 – 12:40	30
CS 2124 - 0E1	CS 2124-0EA	42879	01:30 – 02:00	30
	CS 2124-0EB	42880	02:10 – 02:40	30