CS-2001
Data Structures
Fall 2023

Recursion

Rizwan Ul Haq

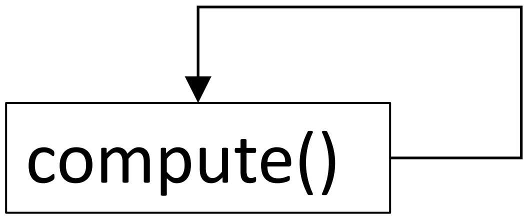
Assistant Professor

FAST-NU

rizwan.haq@nu.edu.pk

Introduction to Recursion

- Recursive method
  - A recursive method is one that calls itself



Introduction to Recursion

- Recursion can be used to manage repetition
- Recursion is a process in which a module achieves a repetition of algorithmic steps by calling itself
- Each recursive call is based on a different, generally simpler, instance

Introduction to Recursion

- Recursion is something of a divide and conquer, top-down approach to problem solving
- It divides the problem into pieces or selects out one key step, postponing the rest

Fundamental Rules

- **Base Case:** Always have at least one case that can be solved without recursion
- **Make Progress:** Any recursive call must progress towards a base case
- **Always Believe:** Always assume the recursive call works
- **Compound Interest Rule:** Never duplicate work by solving the same instance of a problem in separate recursive calls

Basic Form

```
recurse ()
{
        recurse (); //Function calls itself
}
int main ()
{
        recurse (); //Sets off the recursion
}
```

How does it work?

- The module calls itself
- New variables and parameters are allocated storage on the stack
- Function code is executed with the new variables from its beginning. It does not make a new copy of the function. Only the arguments and local variables are new
- As each call returns, old local variables and parameters are removed from the stack
- Then execution resumes at the point of the recursive call inside the function

Why use Recursive Methods?

- In computer science, some problems are more easily solved by using recursive functions
- **For example:**
  - Traversing through a directory or file system
  - Traversing through a tree of search results

Visualizing Recursion

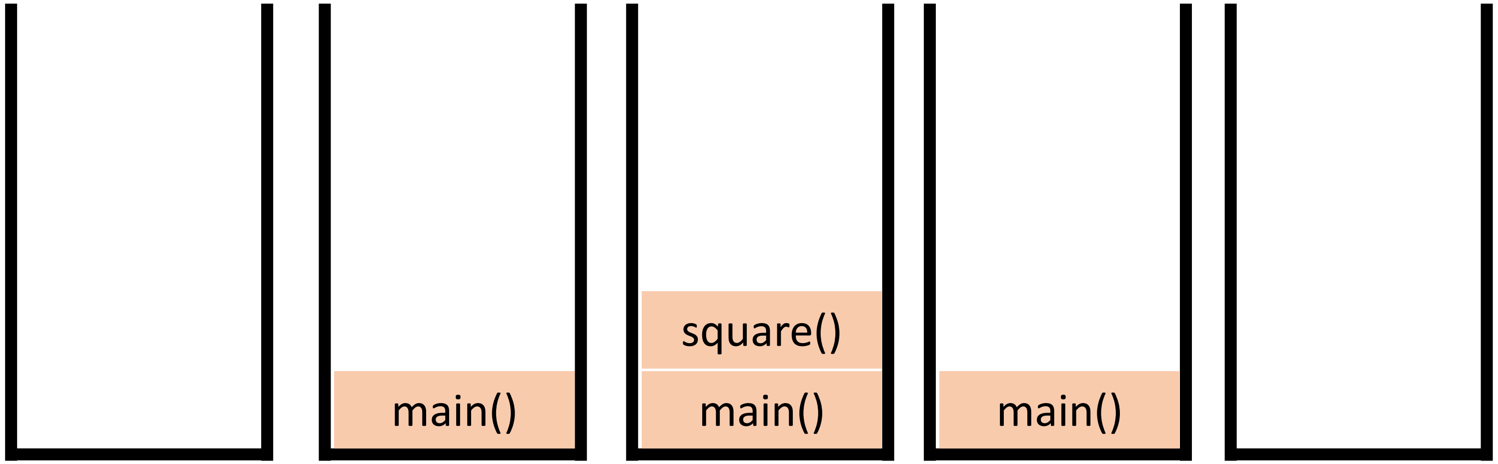
- To understand how recursion works, it helps to visualize what's going on
- To help visualize, we will use a common concept called the Stack
- A stack basically operates like a container of trays in a cafeteria. It has- only two operations:
  - **Push:** you can push something onto the stack
  - **Pop:** you can pop something off the top of the stack

Stacks and Methods

- When you run a program, the computer creates a stack for you
- Each time you invoke a method, the method is placed on top of the stack
- When the method returns or exits, the method is popped off the stack
- The diagram on the next page shows a sample stack for a simple C++ program

Stacks and Methods

```
void main()
{
        square()
}
```



Time = 0	Time = 1	Time = 2	Time = 3	Time = 4
Empty Stack	Push: main()	Push: square()	Pop: square() returns a value method exits	Pop: main() returns a value method exits

A simplest Recursion Program

```
int main ( )
{
     count(0);
}
void count (int index)
{
     cout << index;
     if (index < 2)
          count(index+1);
}
```

This simple program counts 0-2:
012

This is where the recursion occurs. You can see that the count() function calls itself.

What will be the output?

```
int main ( )
{
      count(0);
}
void count (int index)
{
      cout << index;
      if (index < 2)
            count(index+1);
      cout << index;
}
```

Stacks and Recursion

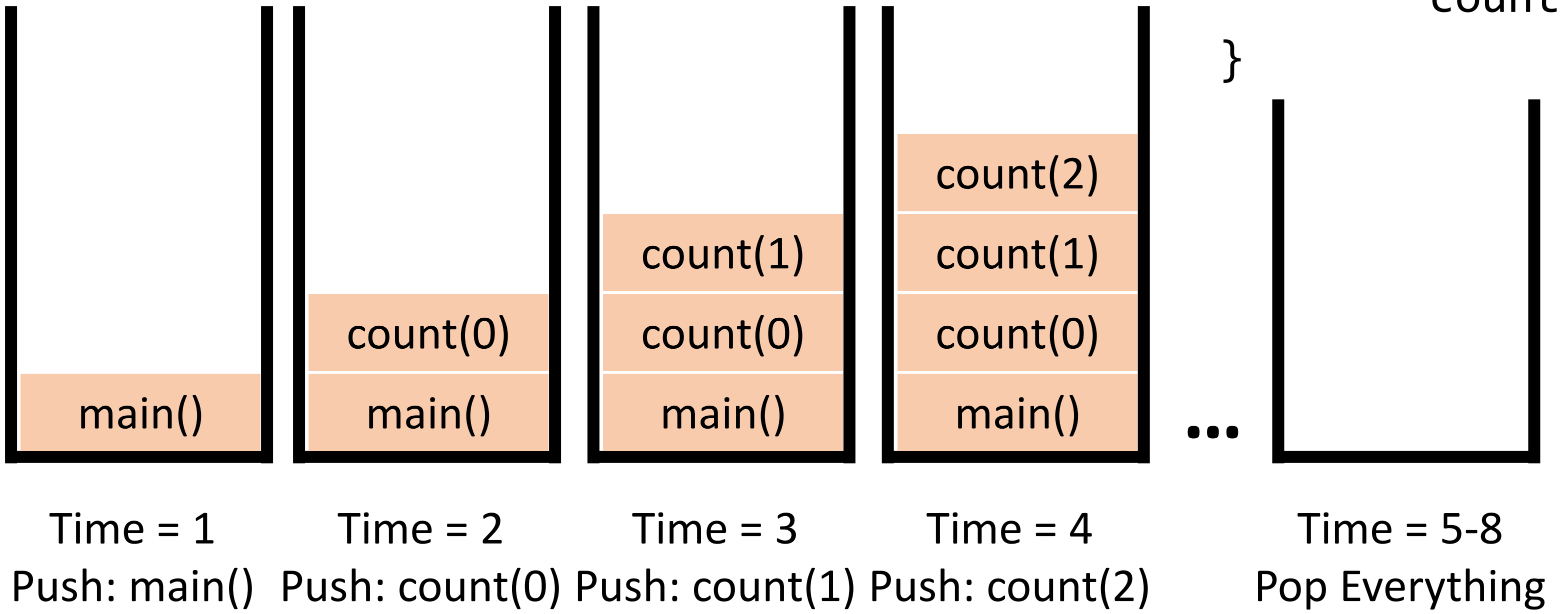
- Each time a method is called, you push the method on the stack
- Each time the method returns or exits, you pop the method off the stack
- If a method calls itself recursively, you just push another copy of the method onto the stack
- We therefore have a simple way to visualize how recursion really works

Back to the Simple Recursion Program

```
int main ( )
{
	count(0);
}
void count (int index)
{
	cout << index;
	if (index < 2)
		count(index+1);
}
```

Stacks and Recursion in Action

```
void count (int index)
{
     cout << index;
     if (index < 2)
          count(index+1);
}
```



Inside count(0):
print (index); → 0
if (index < 2)
count(index+1);

Inside count(1):
print (index); → 1
if (index < 2)
count(index+1);

Inside count(2):
print (index); → 2
if (index < 2)
count(index+1);
This condition now fails,
Hence recursion stops and
we pop up all the functions

Recursion Example #2

```
int main (void)
{
      upAndDown(1);
}
void upAndDown (int n)
{
      cout << "\nlevel:";
      cout << n ;
      if (n < 4)
            upAndDown (n+1);
      cout<<"\nLEVEL: " ;
      cout<< n;
}
```

Determining the Output

- Suppose you were given this problem and your task is to "determine the output."
- How do you figure out the output?
- **Answer:** Use Stacks to Help Visualize

Stack Short-Hand

- Rather than draw each stack like we did last time, you can try using a short-hand notation

Time **Stack**

- time 0: empty stack
- time 1: f(1)
- time 2: f(1), f(2)
- time 3: f(1), f(2), f(3)
- time 4: f(1), f(2), f(3), f(4)
- time 5: f(1), f(2), f(3)
- time 6: f(1), f(2)
- time 7: f(1)
- time 8: empty

Output
level: 1
level: 2
level: 3
level: 4
LEVEL: 4
LEVEL: 3
LEVEL: 2
LEVEL: 1

```
int main (void)
{
    upAndDown(1);
}
void upAndDown (int n)
{
    cout << "\nlevel:";
    cout << n ;
    if (n < 4)
        upAndDown (n+1);
    cout<<"\nLEVEL: " ;
    cout<< n;
}
```

Computing Factorial

- Computing factorials are a classic problem for examining recursion.
- A factorial is defined as follows:

  n! = n * (n-1) * (n-2) …. * 1;

- For example:

  1! = 1 (Base Case)

  2! = 2 * 1 = 2

  3! = 3 * 2 * 1 = 6

  4! = 4 * 3 * 2 * 1 = 24

  5! = 5 * 4 * 3 * 2 * 1 = 120

If you study this table closely, you will start to see a pattern.
The pattern is as follows:
You can compute the factorial of any number (n) by taking n and multiplying it by the factorial of (n-1).
For example:
5! = 5 * 4!
(which translates to 5! = 5 * 24 = 120)

Iterative Approach

```
int main ()
{
        int answer, n;
        cout << "Enter a number = " ;
        cin >> n ;
        answer = findFactorial (n);
}
int findFactorial (int n)
{
    int i, factorial = n;
    for (i = n - 1 ; i >= 1 ; i--)
        factorial = factorial * i;
    return factorial;
}
```

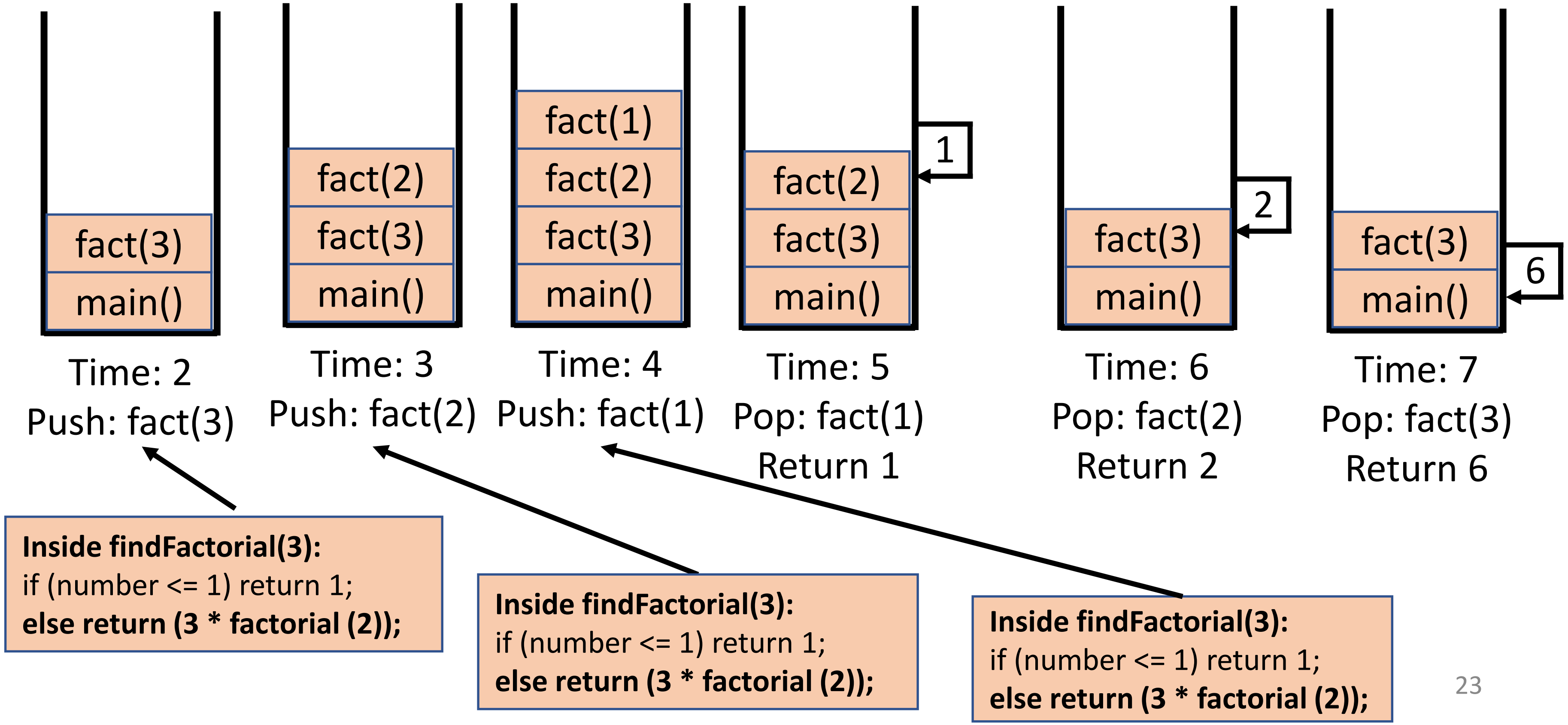
This is an iterative solution to finding a factorial. It's iterative because we have a simple for loop. Note that the for loop goes from n-1 to 1.

Recursive Solution

```
int findFactorial (int number)
{
    if (number == 0 || number==1)
        return 1;
    return number * findFactorial(number-1);
}
```

Finding the factorial of 3

```
int findFactorial (int number)
{
    if (number == 0 || number==1)
        return 1;
    return number*findFactorial(number-1);
}
```

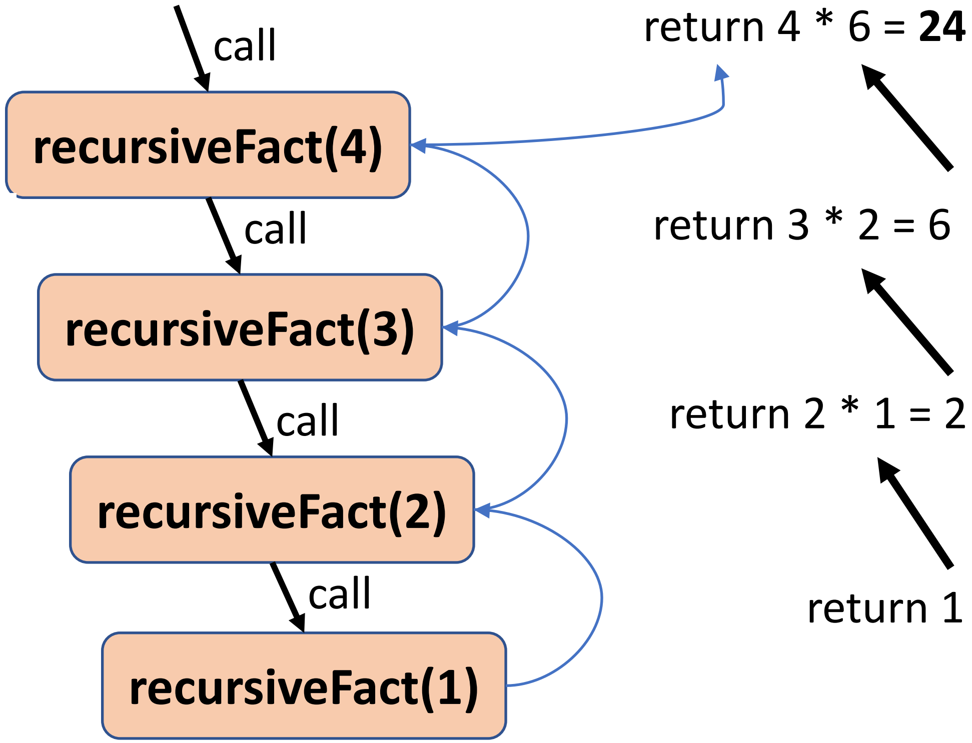


Visualizing Recursion

Recursion trace

- A box for each recursive call
- An arrow from each caller to callee
- An arrow from each callee to caller showing return value

```
int findFactorial (int number)
{
  if (number == 0 || number==1)
    return 1;
  return number*findFactorial(number-1);
}
```



Example: The Fibonacci Series

- Fibonacci series
  - Each number in the series is sum of two previous numbers
- e.g., 0, 1, 1, 2, 3, 5, 8, 13, 21…

fibonacci(0) = 0

fibonacci(1) = 1

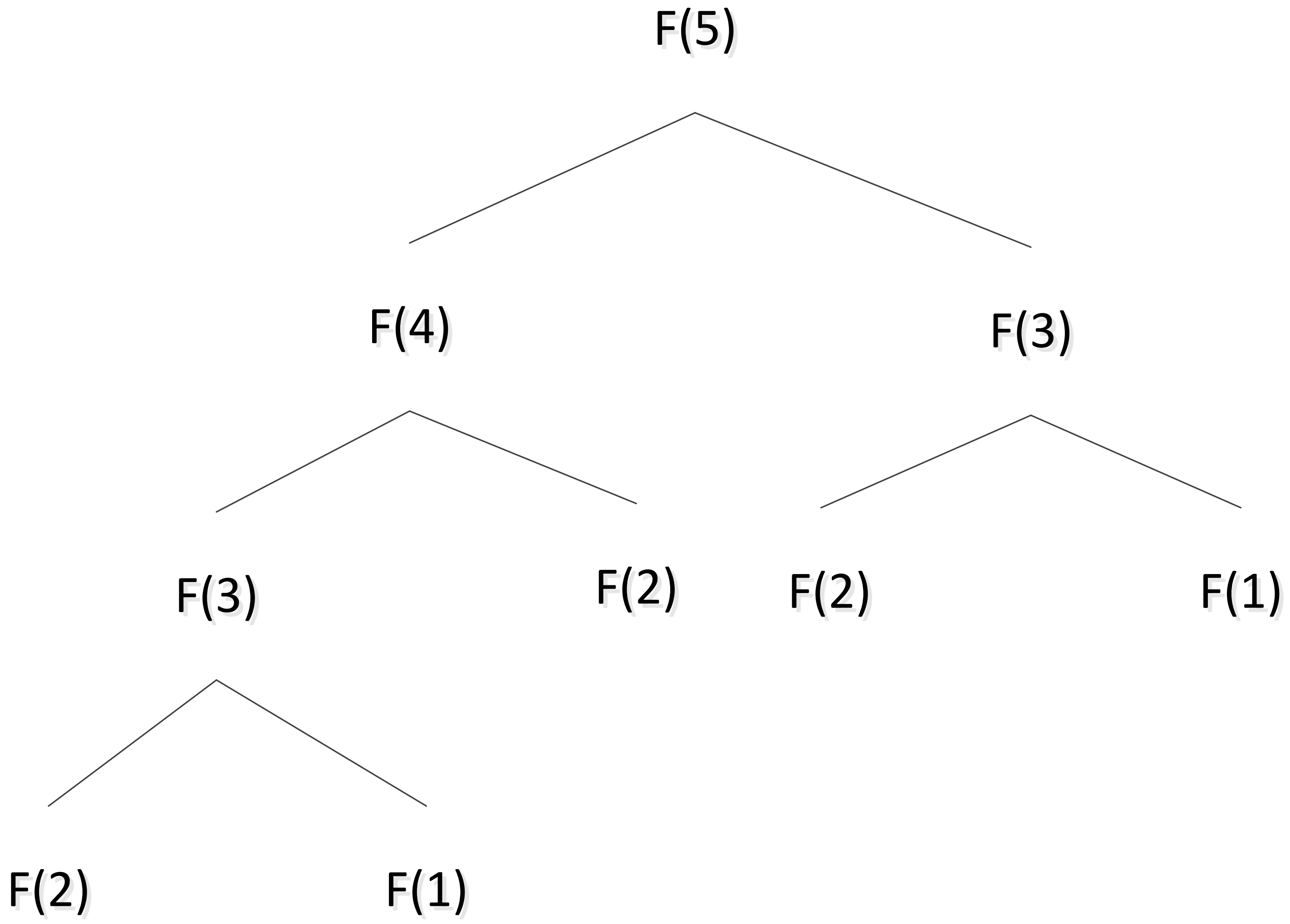
fibonacci(n) = fibonacci(n - 1) + fibonacci(n – 2)

fibonacci(0) and fibonacci(1) are base cases

Fibonacci series

```
// recursive declaration of method fibonacci
long fibonacci( long n )
{
   if ( n == 0 || n == 1 )
      return n;
   else
      return fibonacci( n - 1 ) + fibonacci( n - 2 );
} // end method fibonacci
```

Recursion Tree showing Fibonacci calls



Recursion vs. Iteration

Iteration

- Uses repetition structures (for, while or do…while)
- Repetition through explicit use of repetition structure
- Terminates when loop-continuation condition fails
- Controls repetition by using a counter

Recursion

- Uses selection structures (if, if…else or switch)
- Repetition through repeated method calls
- Terminates when base case is satisfied
- Controls repetition by dividing problem into simpler one
- Often can be implemented with only a few lines of code

Recursion vs. Iteration

Recursion

- More overhead than iteration
- More memory intensive than iteration
- Can also be solved iteratively

Why use it?

PROS

- Clearer logic
- Often more compact code
- Often easier to modify

CONS

- Overhead costs

Some Uses For Recursion

- Numerical analysis
- Graph theory
- Tree traversals
- Game playing
- Sorting
- List processing
- Symbolic manipulation

Practice Problem – 1

- Find the output of following function calls for given piece of code and determine what does the function do.

I. exercise (5);

II. exercise (10);

```
void exercise(int x)
{
  if (x > 0 && x < 10)
  {
    cout << x << " " ;
    exercise(x + 1);
  }
}
```

Practice Problem – 2

- Find the output of following function calls for given piece of code and determine what does the function do.

```
I.   cout << test (5, 10);
II.  cout << test (3, 9);
III.cout << test (4, 1);
```

```
int test(int x, int y)
{
   if(x==y)
         return x;
   else if (x > y)
         return (x + y);
   else
        return test(x + 1, y - 1);
}
```

Practice Problem – 3

Write recursive procedures for the following problems

1. Length of a linked list
2. Print linked list
3. Reverse print linked list
4. Largest element in an array
5. Tower of Hanoi

Tower of Henoi

```
void towers(int n, char from, char to, char aux)
{
    if(n==1)
    {
        cout << "Move disk " << n << " from " << from
                << " to " <<  to << endl;
        return;
    }
    else
    {
        towers(n-1, from, aux, to);
        cout << "Move disk " << n << " from "
                << from << " to " << to << endl;
        towers(n-1, aux, to, from);
    }
}
```

Home Work

1. Write a recursive Sequential search in an array (base and general case)
2. Write a recursive solution for the given sequence.

   $a_0 = 1, a_1 = 2$

   $a_n = 2 * a_{n-1} + 3 * a_{n-2} + 4$
3. Think carefully about base case, base case value and prototype of the function
4. Write a recursive solution to print a binary number from given decimal value