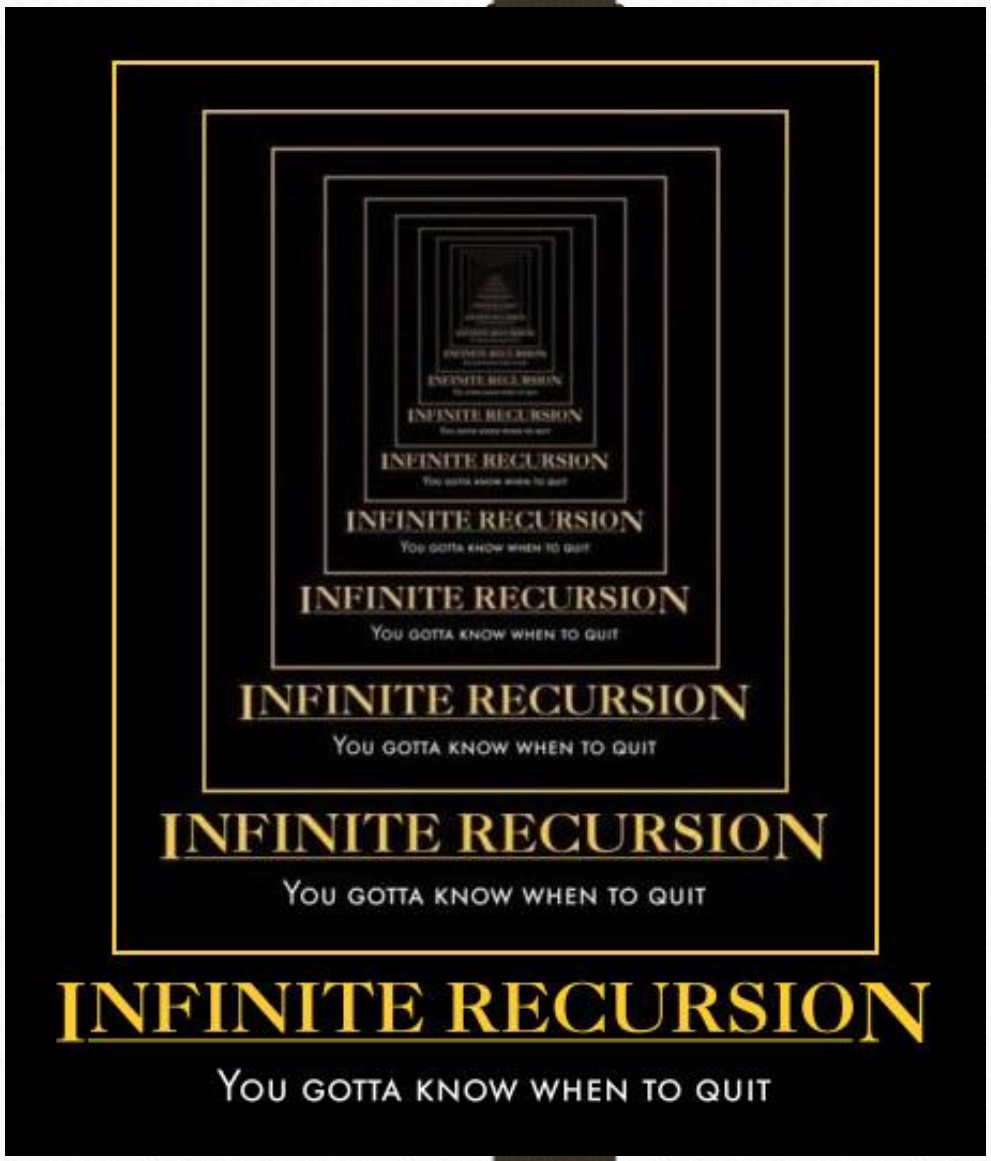
# CS330

The wonders about recursion


INFINITE RECURSION
You gotta know when to quit
INFINITE RECURSION
You gotta know when to quit
INFINITE RECURSION
You gotta know when to quit
INFINITE RECURSION
You gotta know when to quit
INFINITE RECURSION
You gotta know when to quit

# Recursive Algorithms

- **Definition:**

- **Recursion**
  - See "Recursion".

# What is a recursive Algorithm?

- Recursion breaks a problem into several smaller problems
- The smaller problems are of exactly the same type as the original problem
- How does it work?
  - Solve a small instance of the big problem
  - The small instance is solved by solving an even smaller instance
  - The new problem is so small, its solution is obvious or known.
  - This solution will lead to the solution of the original problem
- Are recursive solutions always better than iterative ones?
  - NO!

# How to find a recursive Solution?

1. What is the base case, and can it be solved?
2. What is the general case?
3. Does the recursive call make the problem smaller and does it approach the base case?

# How to find a recursive Solution?

- Base Case:
  - Is the problem we know the answer to, that can be solved without any more recursive calls
    - Stops the recursion
    - At least one per recursive function
- General Case:
  - Is what happens the most of the time, and is when the recursive call takes place

# analysis oF recursvie algorithms
# Order of function calling

```
void Recursive(int i)
{
 if(i < 5)
 {

        printf("I am %d\n", i);
        Recursive(i + 1);
 }
}
```

| | | | | |
|---|---|---|---|---|
| Recursive(0) | | | | |
| "I am 0" | | | | |
| | Recursive(1) | | | |
| | "I am 1" | | | |
| | | Recursive(2) | | |
| | | "I am 2" | | |
| | | | Recursive(3) | |
| | | | "I am 3" | |
| | | | | Recursive(4) |
| | | | | "I am 4" |

# analysis oF recursvie algorithms
# Order of function calling

```
void Recursive(int i)
{
 if(i < 5)
 {

        Recursive(i + 1);
        printf("I am %d\n", i);
 }
}
```

| | | | | |
|---|---|---|---|---|
| Recursive(0) | | | | |
| | Recursive(1) | | | |
| | | Recursive(2) | | |
| | | | Recursive(3) | |
| | | | | Recursive(4) |
| | | | | "I am 4" |
| | | | "I am 3" | |
| | | "I am 2" | | |
| | "I am 1" | | | |
| "I am 0" | | | | |

# analysis of recursvie algorithms

- Direct recursion
  - The function calls itself
- Indirect recursion
  - Function 1 calls Function 2
  - Function 2 calls Function 3
  - Function 3 calls Function 4
  - Function 4 calls Function 1

# Example Factorial of N

- Write down an iterative and a recursive algorithm to calculate N!
  - What is the base case?
  - What is the general case?
- What is BigO of your iterative algorithm?
- What is BigO of your recursive algorithm?

# Example Factorial of N

- What is the real running time ?
  - Let's make an experiment!

# Overhead of Recursion

- Keeping track of the information about all active functions can use many system resources (esp. if the recursion goes many levels deep)
- Recursion might not be the most efficient way to implement an algorithm. Each time a function is called, there is a certain amount of "overhead" that takes up memory and system resources.
- When a function is called from another function, all the information about the first function must be stored so that the computer can return to it after executing the new function.

# Head Recursion vs tail recursion

- When a function calls another function, it stores one or more variables on the stack
  - The stack is a memory area where temporary variables are stored
  - The stack works using the LIFO method (Last In First Out)
- The stored values (Pushed) are saved because they're going to be needed later

# Head Recursion vs tail recursion

```
unsigned long int Factorial_Recursive(int n)
{
        if (n == 0)          return 1;
        else
        return n * Factorial_Recursive(n - 1);
}
```

- First time this function is called: n = 2
- The function knows the value of n, but not the value of "Factorial_Recursive(n – 1)"
- Therefore it pushes n (whose value is 2) on the stack, and calls itself
- Now n is 1. Again, the function knows the value of n, but not the value of "Factorial_Recursive(n – 1)"
- It pushed n (whose value is 1) on the stack, and calls itself
- Now n is 0, and the base case is reached. The function returns 1
- This "1" is multiplied by the top of stack which is also "1", and pops it out of the stack
- The result is again multiplied by the new top of the stack, which is 2

# Head Recursion vs tail recursion

- The stack has a limited amount of memory
- A recursive function with a very high recursive depth might reach this limitation
  - Stack overflow
- Head recursion is defined when the function has instructions to do after calling itself
  - Middle and multi recursion are defined in the middle of the function
  - From the efficiency point of view, there are no differences between a head, a middle or a multi recursive function
- Tail recursion is defined when the recursive call is at the very end of the recursive function

# Head Recursion vs tail recursion

- Tail recursion factorial function
- Recursive call is the last statement in the function. Nothing is pushed onto the stack

```
int Factorial_Tail(int n)
{
        if(0 == n)      return 1;
        return Factorial_Prime(n, 1);
}

int  Factorial_Prime(int CurrentNumber, int Sum)
{
        if(1 == CurrentNumber)     return Sum;
        return Factorial_Prime(CurrentNumber – 1,
Sum*CurrentNumber);
}
```

# Analyze of recursive Algorithms

- Correctness
- Runningtime