Concepts of Algorithms
CS-211
Chapter 5
Recursion

Recursion

Recursion is an important concept in computer science. Many algorithms can be best described in terms of recursion. A recursive function / procedure containing a Call statement to itself. To make a function recursive one must consider the following properties:

- There must be certain (using arguments), called base criteria, for which the procedure / function does not call itself.
  - 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
- Each time the procedure / function does call itself, control must be closer to the base criteria.

Examples (Recursion without Criteria)

The following program demonstrates function call to itself. The following function does not follows the first property i.e. (recursion must has some criteria). Endless execution ????

```
#include <iostream.h>
void disp( ); // prototyping

    int main( )
    {
        cout<<"\n Hello";
        disp( );    /* A function,  call */
        return 0;
    }
    void disp( )
      {  cout<<"Hello \t";
        disp( ); /* A function call to itself without any criteria */
      }
```

Examples (Recursion, going far and far from Base Criteria) after every iteration

The second property of recursion i.e. (after each cycle/iteration, control must reach closer to the base criteria) and it is ignored here, so the following program is logically invalid.

```
#include <iostream.h>
    void numbers(int );  // function prototyping
    int main( )
    {     int n=10;
         numbers(n);
         return 0;
    }
    void numbers(int n )
    {     cout<<"Hello\t";
          if(n >= 1 ) numbers(n+1); // this needs explanation
       // after each iteration, control is going far and far from base criteria
    }
```

Recursive Functions

- The easiest examples of recursion to understand are functions in which the recursion is clear from the definition. As an example, consider the factorial function, which can be defined in either of the following ways:

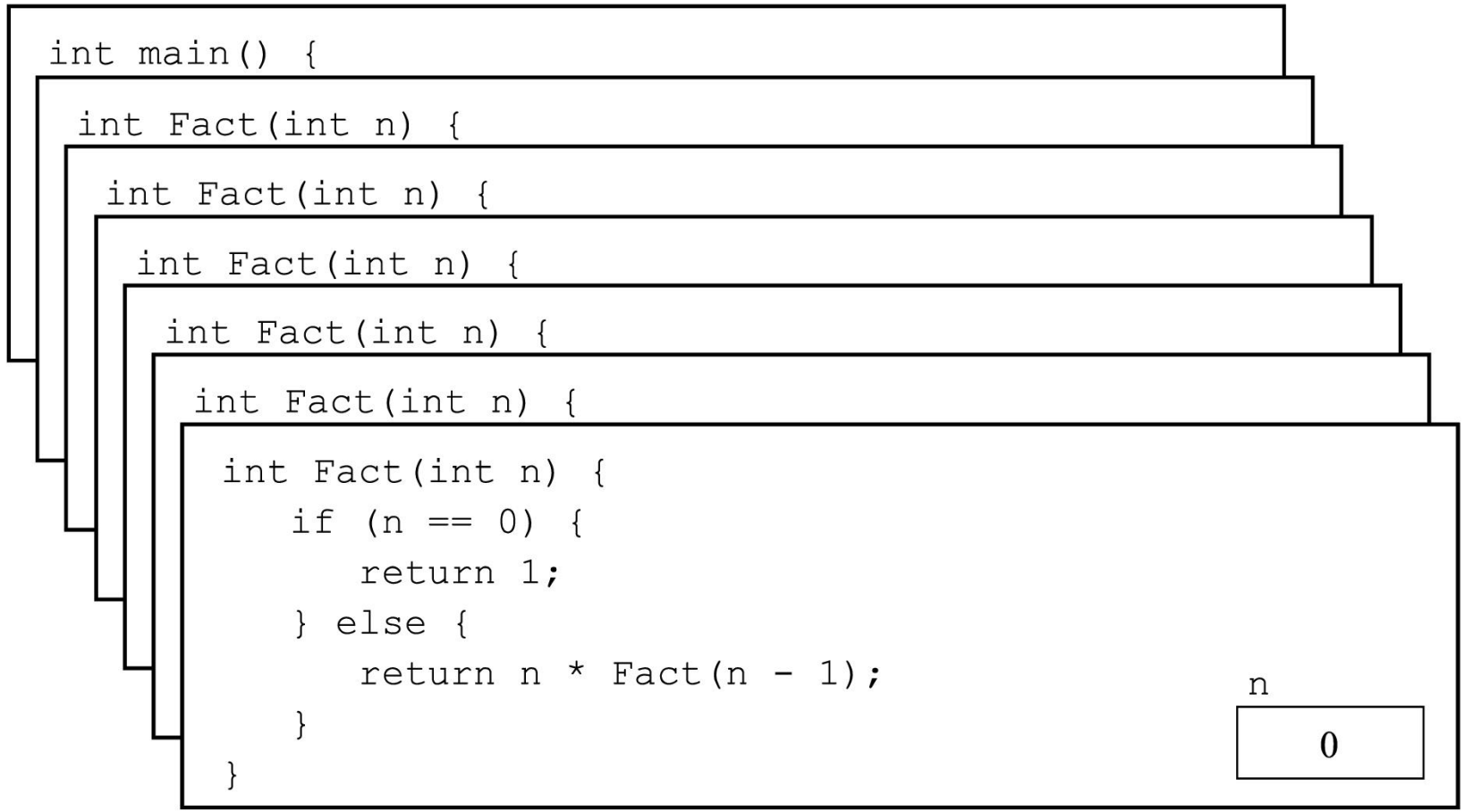
$$n! = n \times (n - 1) \times (n - 2) \times \cdots \times 3 \times 2 \times 1$$

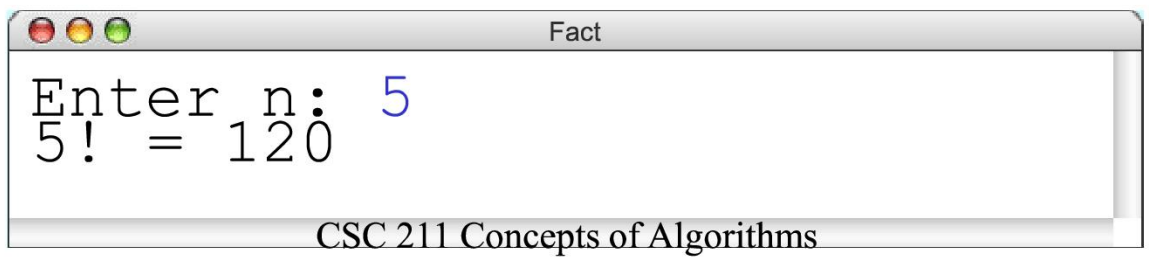
$$n! = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ n \times (n - 1)! & \text{otherwise} \end{cases}$$

- The second definition leads directly to the following code, which is shown in simulated execution on the next slide:

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

Simulating the `Fact` Function



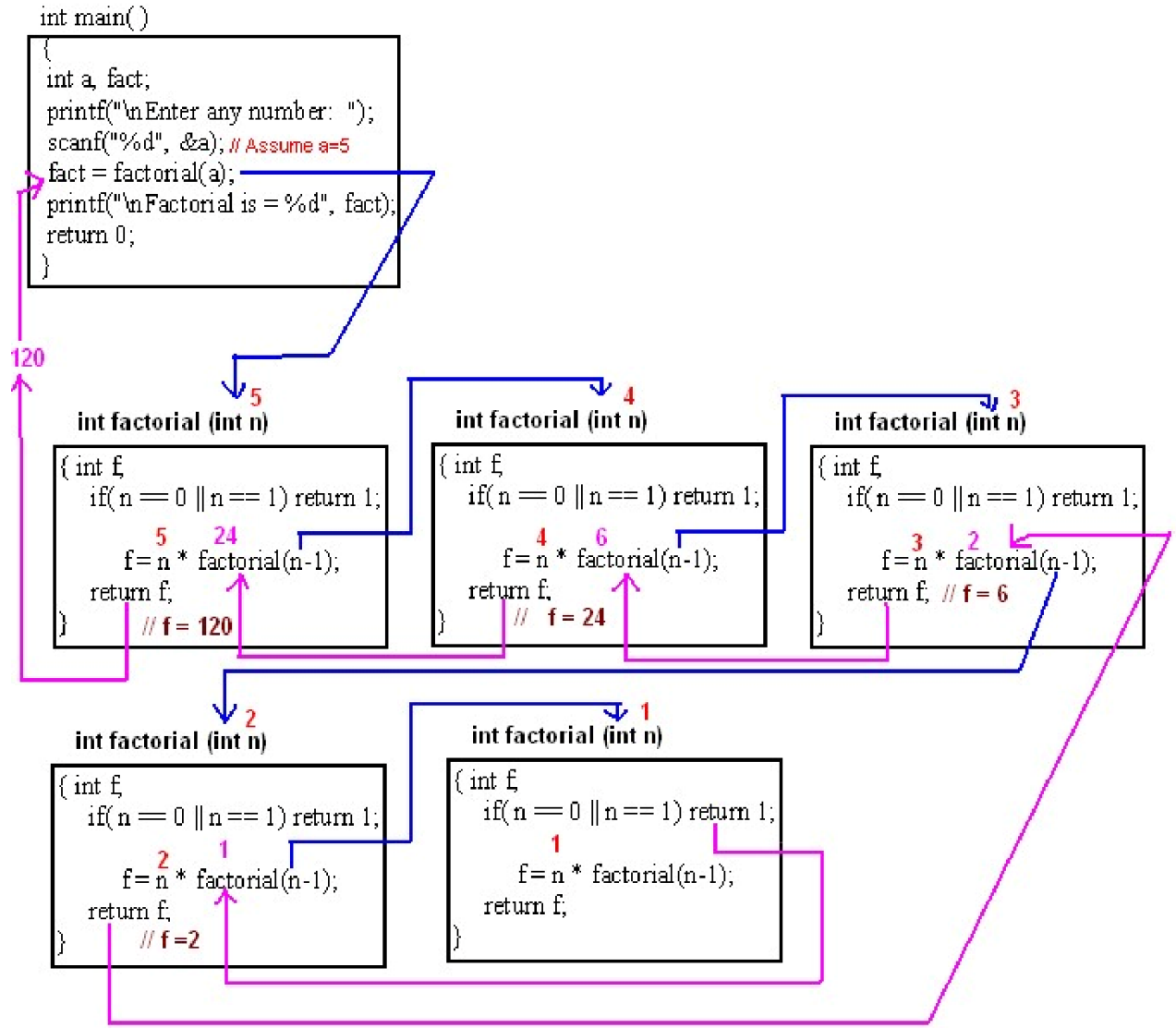


skip simulation

Code in C++ to find Factorial using Recursion

```
#include <iostream.h>
    int factorial(int);  // function prototyping
    int main( )
    { int n, fact;
      cout<<"\nEnter any number:  ";
       cin>>n;
     fact = factorial(n);// function call
     cout<<"\nFactorial is = "<< fact<<endl;
     return 0;
    }
    int   factorial(int  n)
    {
        if( n == 0 || n == 1) return 1;   else
            return  n * factorial(n-1);
     }
```

Dry Run of the code



The Recursive Paradigm

- Most recursive methods you encounter in an introductory course have bodies that fit the following general pattern:

```
if (test for a Base Criteria true) {
    Compute and return the simple solution without using recursion.
} else {
    Divide the problem into one or more subprobrlems that have the same form.
    Solve each of the problems by calling this method recursively.
    Return the solution from the results of the various subproblems.
}
```

- *Finding a recursive solution is mostly a matter of figuring out how to break it down so that it fits the paradigm. When you do so, you must do two things:*

1. *Identify simple **Base Criteria** that can be solved without recursion.*

2. *Find a recursive decomposition that breaks each instance of the problem into simpler subproblems of the same type, which you can then solve by applying the method recursively.*

Exercise: A Recursive GCD Function

One of the oldest known algorithms that is worthy of the title is Euclid's algorithm for computing the greatest common divisor (GCD) of two integers, *x* and *y*. Euclid's algorithm is usually implemented iteratively using code that looks like this:

```
int GCD(int x, int y) {
   int r = x % y;
   while (r != 0) {
      x = y;
      y = r;
      r = x % y;
   }
   return y;
}
```

Rewrite this method so that it uses recursion instead of iteration, taking advantage of Euclid's insight that the greatest common divisor of *x* and *y* is also the greatest common divisor of *y* and the remainder of *x* divided by *y*.

Exercise: A Recursive GCD Function

```
Int GCD(x, y)
{
        if y = 0 then
                return x
        else
                return GCD(y, x%y)
        endif
}
```

```
Int GCD(x, y)
{
        if x%y = 0 then
                return y
        else
                return GCD(y, x%y)
        endif
}
```

Exercise: A Recursive GCD Function

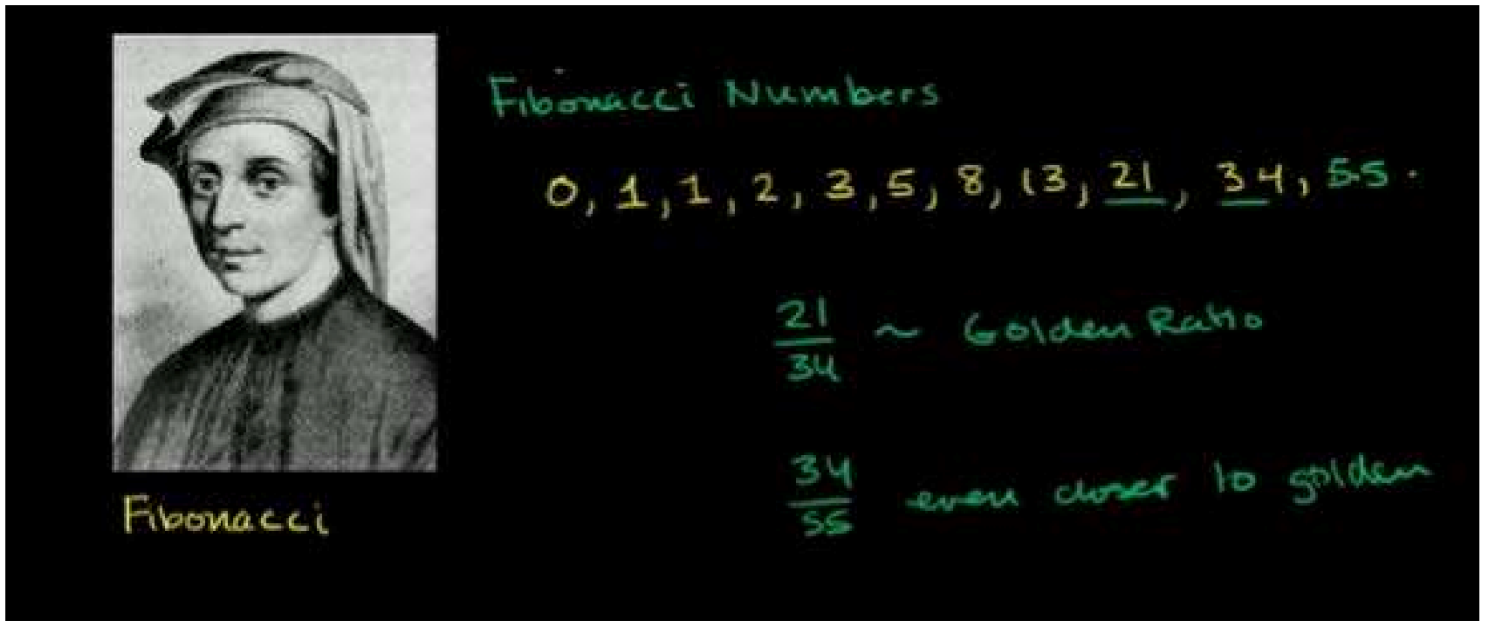
```c
#include <stdio.h>
int hcf(int n1, int n2);
int main() {
    int n1, n2;
    printf("Enter two positive integers: ");
    scanf("%d %d", &n1, &n2);
    printf("G.C.D of %d and %d is %d.", n1, n2, hcf(n1, n2));
    return 0;
}

int hcf(int n1, int n2) {
    if (n2 != 0)
        return hcf(n2, n1 % n2);
    else
        return n1;
}
```

Exercise: A Recursive `Fibonacci` Function

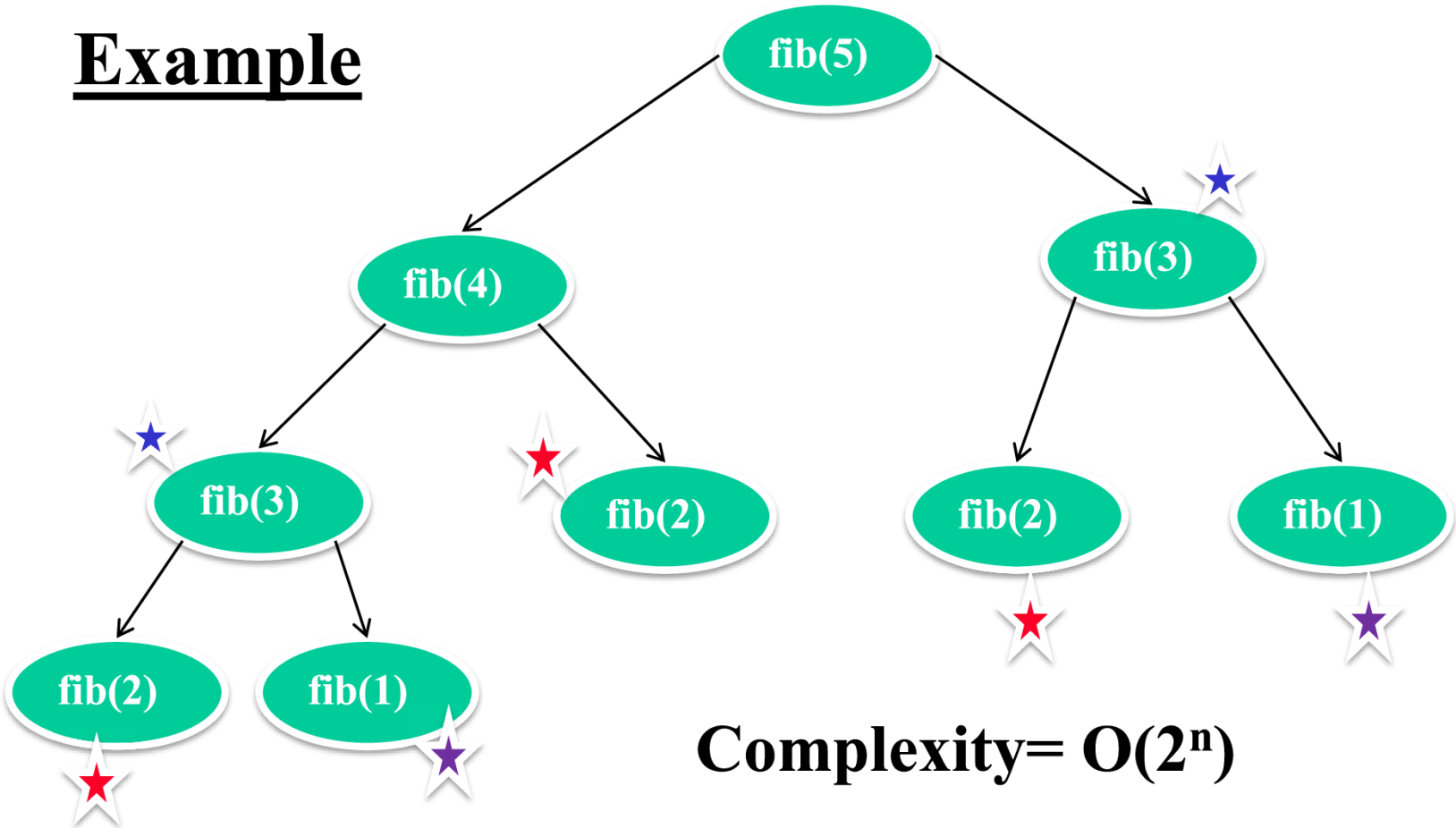
Write a **recursive function** for calculating the n-th **Fibonacci** number.

- F(0)=0
- F(1)=1
- F(n)=F(n-1)+F(n-2), for n>1



Exercise: A Recursive `Fibonacci` Function

Example



Complexity= $O(2^n)$

Repetition !!!

Exercise: A Recursive `Fibonacci` Function

- F(0)=0
- F(1)=1
- F(n)=F(n-1)+F(n-2), for n>1

Recursive Solution

```
int  fib(n) {
     If(n==1 or n==2)
        result = 1
     else
        result = fib(n-1)+fib(n-2)
     return result
     }
```

```c
#include<stdio.h>
int fib(int n){
 if (n <= 1)
   return n;
 return fib(n-1) + fib(n-2);
}
int main (){
 int n = 9;
 printf("%d", fib(n));
 return 0;
}
```

END