Recursion

Part I (Lecture 15)

Recursion

- A process by which a function calls itself repeatedly
  - Either directly.
    - X calls X
  - Or cyclically in a chain.
    - X calls Y, and Y calls X
- Used for repetitive computations in which each action is stated in terms of a previous result

  fact(n) = n * fact (n-1)

- For a problem to be written in recursive form, two conditions are to be satisfied:
  - It should be possible to express the problem in recursive form
    - Solution of the problem in terms of solution of the same problem on smaller sized data
  - The problem statement must include a stopping/terminating condition
    - The direct solution of the problem for a small enough size

$$\text{fact}(n) = \begin{cases} 1, & \text{if } n = 0 \\ n * \text{fact}(n-1), & \text{if } n > 0 \end{cases}$$

Stopping/Terminating condition

Recursive definition

- Examples:

  - Factorial:

    fact(0) = 1

    fact(n) = n * fact(n-1), if n > 0

  - GCD:

    gcd (m, m) = m

    gcd (m, n) = gcd (n, m%n)

  - Fibonacci series (1,1,2,3,5,8,13,21,….)

    fib (0) = 1

    fib (1) = 1

    fib (n) = fib (n-1) + fib (n-2), if n > 1

Factorial

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

Factorial Execution

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

Factorial Execution

fact(4)

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

Factorial Execution

```
fact(4)
   if   (4 = = 1) return (1);
   else return  (4 * fact(3));
```

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

Factorial Execution

```
fact(4)
   if   (4 = = 1) return (1);
   else return  (4 * fact(3));

            if   (3 = = 1) return (1);
            else return  (3 * fact(2));
```

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

Factorial Execution

```
fact(4)
   if   (4 = = 1) return (1);
   else return  (4 * fact(3));

          if   (3 = = 1) return (1);
          else return  (3 * fact(2));

                 if   (2 = = 1) return (1);
                 else return  (2 * fact(1));
```

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

Factorial Execution

```
fact(4)
   if   (4 = = 1) return (1);
   else return  (4 * fact(3));

        if   (3 = = 1) return (1);
        else return  (3 * fact(2));

             if   (2 = = 1) return (1);
             else return  (2 * fact(1));

                  if   (1 = = 1) return (1);
```

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

Factorial Execution

```
fact(4)
   if   (4 = = 1) return (1);
   else return  (4 * fact(3));

        if   (3 = = 1) return (1);
        else return  (3 * fact(2));

             if   (2 = = 1) return (1);
             else return  (2 * fact(1));      1

                  if   (1 = = 1) return (1);
```

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

Factorial Execution

```
fact(4)
   if   (4 = = 1) return (1);
   else return  (4 * fact(3));

        if   (3 = = 1) return (1);
        else return  (3 * fact(2));      2

             if   (2 = = 1) return (1);
             else return  (2 * fact(1));      1

                  if   (1 = = 1) return (1);
```

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

Factorial Execution

```
fact(4)
   if   (4 = = 1) return (1);
   else return  (4 * fact(3));                 6

         if   (3 = = 1) return (1);
         else return  (3 * fact(2));                 2

               if   (2 = = 1) return (1);
               else return  (2 * fact(1));               1

                     if   (1 = = 1) return (1);
```

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

Factorial Execution

24

fact(4)

```
if   (4 = = 1) return (1);
else return  (4 * fact(3));
```

6

```
if   (3 = = 1) return (1);
else return  (3 * fact(2));
```

2

```
if   (2 = = 1) return (1);
else return  (2 * fact(1));
```

1

```
if   (1 = = 1) return (1);
```

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

Example: Finding max in an array

```
int findMax(int A[ ], int n)
{
        int temp;
        if (n==1)
        {
                return A[0];
        }
        temp = findMax(A, n-1);
        if (A[n-1] > temp)
                return A[n-1];
        else return temp;
}
```

Terminating condition. Small size problem that you know how to solve directly without calling any functions

Recursive call. Find the max in the first n-1 elements (exact same problem, just solved on a smaller array).

Important things to remember

- Think how the whole problem (finding max of n elements in A) can be solved if you can solve the exact same problem on a smaller problem (finding max of first n-1 elements of the array). But then, do NOT think how the smaller problem will be solved, just call the function recursively and assume it will be solved.
- When you write a recursive function
    - First write the terminating/base condition
    - Then write the rest of the function
    - Always double-check that you have both

Back to Factorial: Look at the variable addresses (a slightly different program) !

```
int main()
{
    int  x,y;
    scanf("%d",&x);
    y = fact(x);
    printf  ("M: x= %d, y = %d\n", x,y);
    return 0;
}
int fact(int data)
{ int val = 1;
  printf("F: data = %d, &data = %u \n
    &val = %u\n", data, &data, &val);
  if (data>1) val = data*fact(data-1);
    return val;
}
```

Output

```
4
F: data = 4, &data = 3221224528
 &val = 3221224516
F: data = 3, &data = 3221224480
 &val = 3221224468
F: data = 2, &data = 3221224432
 &val = 3221224420
F: data = 1, &data = 3221224384
 &val = 3221224372
M: x= 4, y = 24
```

- The memory addresses for the variable data are different in different calls!
- They are not the same variable.
- Each function call will have its own set of variables, even if the name of the variable is the same as it is the same function being called
- Change made to one will not be seen by the calling function on return

```
int main()
{
    int  x,y;
    scanf("%d",&x);
    y = fact(x);
    printf  ("M: x= %d, y = %d\n", x,y);
    return 0;
}
int fact(int data)
{
   int val = 1, count = 0;
   if (data>1) val = data*fact(data-1);
   count++;
   printf("count = %d, data = %d\n",
count, data);
   return val;
}
```

Output

```
4
count = 1, data = 1
count = 1, data = 2
count = 1, data = 3
count = 1, data = 4
M: x= 4, y = 24
```

- Count did not change even though ++ done!
- Each call does it on its own copy, lost on return

Recursion

Part II (Lecture 16)

Fibonacci Numbers

Fibonacci recurrence:

fib(n) = 1 if n = 0 or 1;

= fib(n – 2) + fib(n – 1)

otherwise;

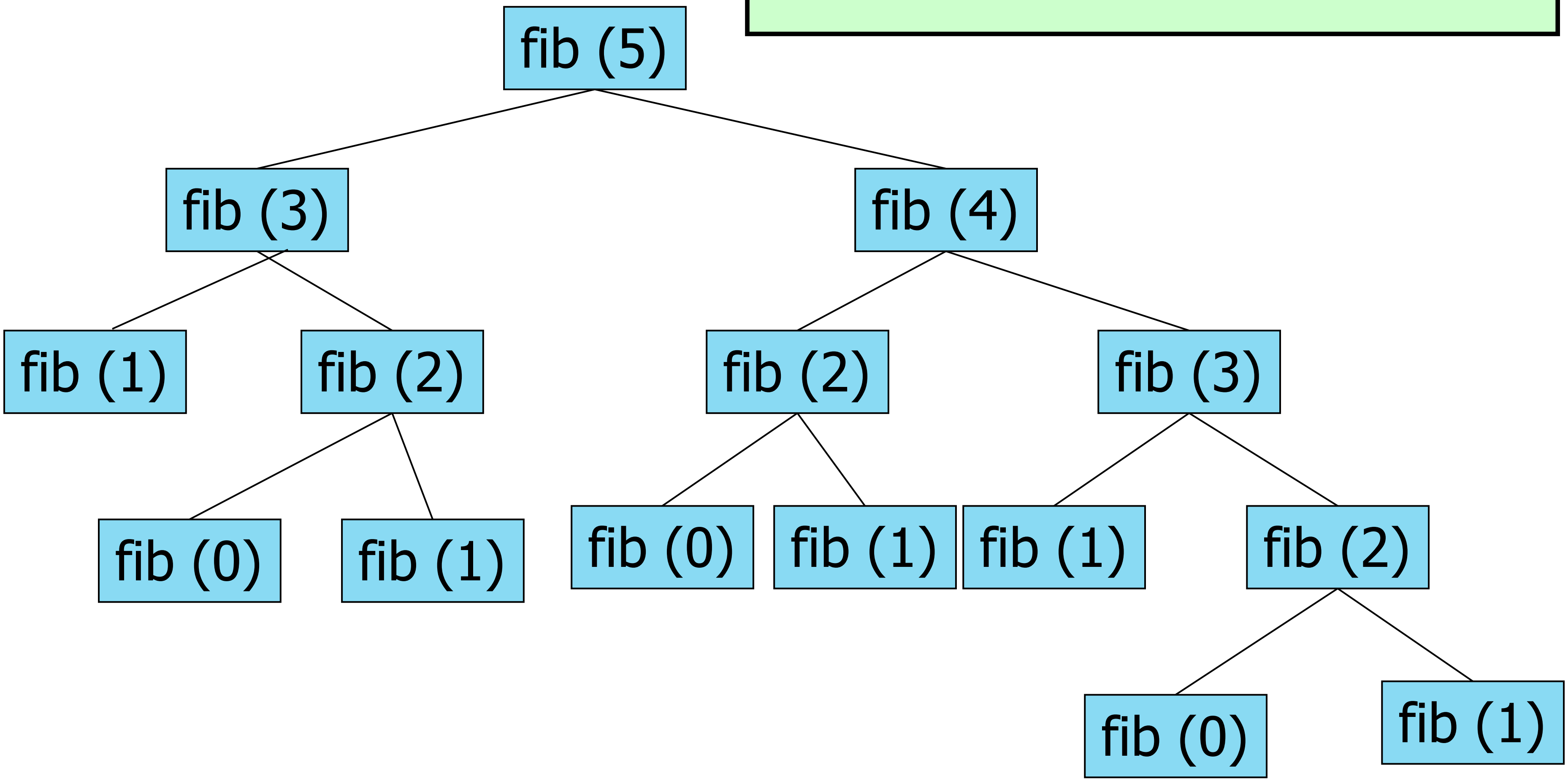
```
int fib (int n)      {
    if (n == 0 or n == 1)
        return 1;    [Base]
    return fib(n-2) + fib(n-1) ;
                              [Recursive]
 }
```

```
int fib (int n)  {
   if (n == 0 || n == 1)
         return 1;
   return fib(n-2) + fib(n-1) ;
}
```

Fibonacci recurrence:

fib(n) = 1 if n = 0 or 1;

= fib(n – 2) + fib(n – 1) otherwise;

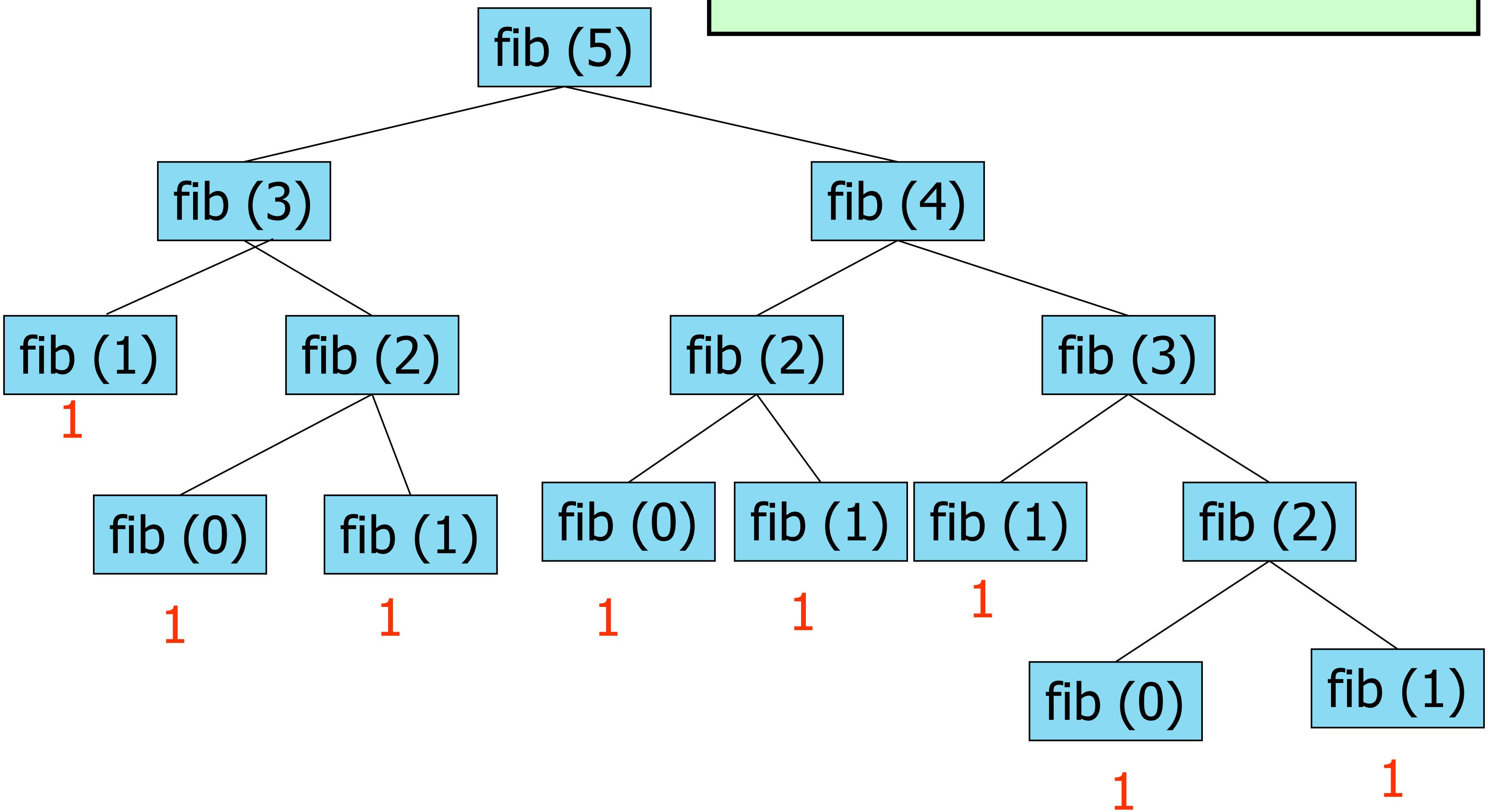


```
int fib (int n)  {
   if (n == 0 || n == 1)
         return 1;
   return fib(n-2) + fib(n-1) ;
}
```

Fibonacci recurrence:

fib(n) = 1 if n = 0 or 1;

= fib(n – 2) + fib(n – 1) otherwise;

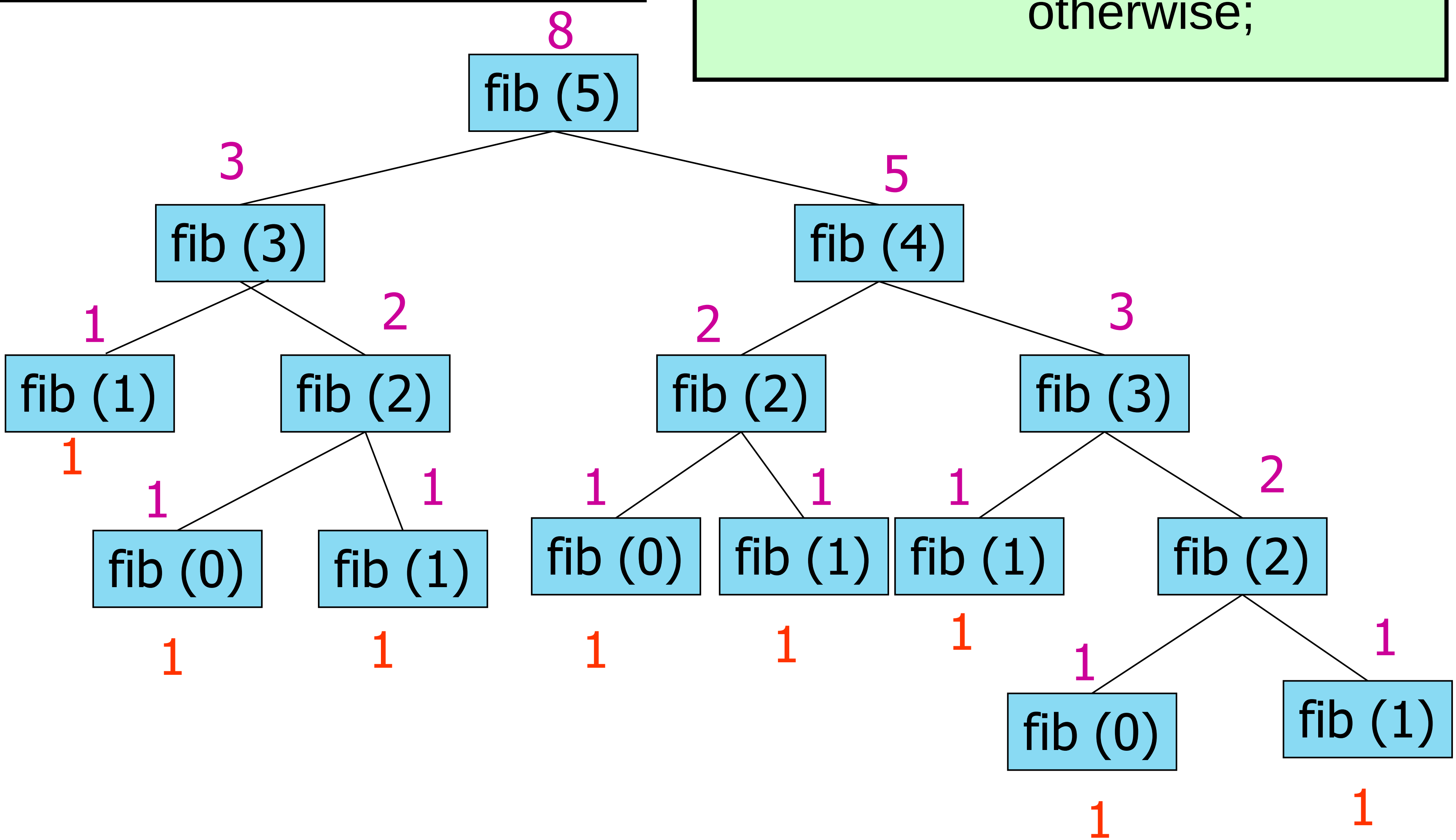


```
int fib (int n)  {
   if (n==0 || n==1)
         return 1;
   return fib(n-2) + fib(n-1) ;
}
```

Fibonacci recurrence:

fib(n) = 1 if n = 0 or 1;

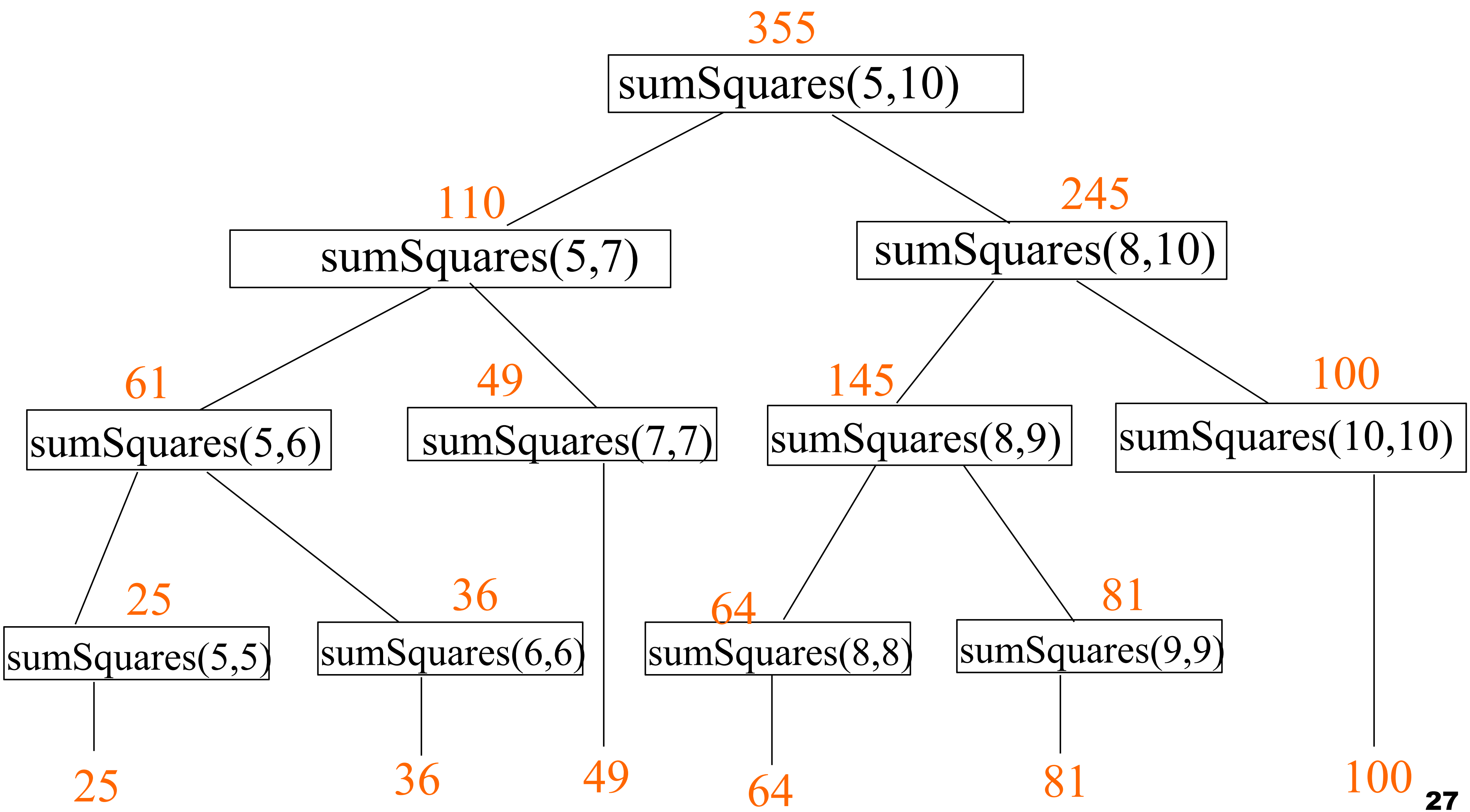
= fib(n – 2) + fib(n – 1) otherwise;



Example: Sum of Squares

```
int sumSquares (int m, int n)
{
    int middle ;
    if (m == n) return(m*m);
    else
    {
        middle = (m+n)/2;
        return (sumSquares(m,middle)
                    + sumSquares(middle+1,n));
     }
}
```

Annotated Call Tree



Example: Printing the digits of an Integer in Reverse

- Print the last digit, then print the remaining number in reverse
  - Ex: If integer is 743, then reversed is print 3 first, then print the reverse of 74

```
void printReversed(int i)
{
    if (i < 10)   {
       printf(“%d\n”, i); return;
    }
    else {
        printf(“%d”, i%10);
        printReversed(i/10);
    }
}
```

Counting Zeros in a Positive Integer

- Check last digit from right
  - If it is 0, number of zeros = 1 + number of zeroes in remaining part of the number
  - If it is non-0, number of zeros = number of zeroes in remaining part of the number

```
int zeros(int number)
{
    if(number<10) return 0;
    if (number%10 == 0)
            return(1+zeros(number/10));
    else
       return(zeros(number/10));
}
```

Example: Binary Search

- Searching for an element k in a sorted array A with n elements
- Idea:
  - Choose the middle element A[n/2]
  - If k == A[n/2], we are done
  - If k < A[n/2], search for k between A[0] and A[n/2 -1]
  - If k > A[n/2], search for k between A[n/2 + 1] and A[n-1]
  - Repeat until either k is found, or no more elements to search
- Requires less number of comparisons than linear search in the worst case ($\log_2 n$ instead of n)

```
int binsearch(int A[ ], int low, int high, int k)
{
     int mid;
     printf("low = %d, high = %d\n", low, high);
     if (low > high)
          return 0;
    mid = (low + high)/2;
    printf("mid = %d, A[%d] = %d\n\n", mid, mid, A[mid]);
    if (A[mid] == k)
          return 1;
    else {
          if (A[mid] > k)
                    return (binsearch(A, low, mid-1, k));
          else
                    return(binsearch(A, mid+1, high, k));
    }
}
```

```
int main()
{
    int A[25], n, k, i, found;

    scanf("%d", &n);
    for (i=0; i<n; i++) scanf("%d", &A[i]);
    scanf("%d", &k);
    found = binsearch(A, 0, n-1, k);
    if (found == 1)
        printf("%d is present in the array\n", k);
    else
        printf("%d is not present in the array\n", k);
}
```

Output

```
8
9 11 14 17 19 20 23 27
21
low = 0, high = 7
mid = 3, A[3] = 17

low = 4, high = 7
mid = 5, A[5] = 20

low = 6, high = 7
mid = 6, A[6] = 23

low = 6, high = 5
21 is not present in the array
```

```
8
9 11 14 17 19 20 23 27
14
low = 0, high = 7
mid = 3, A[3] = 17

low = 0, high = 2
mid = 1, A[1] = 11

low = 2, high = 2
mid = 2, A[2] = 14

14 is present in the array
```

Static Variables

```
int Fib (int, int);

int main()
{
    int n;
    scanf("%d", &n);
    if (n == 0 || n ==1)
      printf("F(%d) = %d \n", n, 1);
    else
      printf("F(%d) = %d \n", n,
    Fib(n,2));
    return 0;
}
```

```
int Fib(int n, int i)
{
    static int m1, m2;
    int res, temp;
    if (i==2) {m1 =1; m2=1;}
    if (n == i)  res = m1+ m2;
    else
     {   temp = m1;
          m1 = m1+m2;
          m2 = temp;
         res = Fib(n, i+1);
     }
    return res;
}
```

Static variables remain in existence rather than coming and going each time a function is activated

Static Variables: See the addresses!

```
int Fib(int n, int i)
{
  static int m1, m2;
  int res, temp;
  if (i==2) {m1 =1; m2=1;}
  printf("F: m1=%d, m2=%d, n=%d,
                  i=%d\n", m1,m2,n,i);
  printf("F: &m1=%u, &m2=%u\n",
                        &m1,&m2);
  printf("F: &res=%u, &temp=%u\n",
                  &res,&temp);
  if (n == i)  res = m1+ m2;
  else {  temp = m1;  m1 = m1+m2;
      m2 = temp;
      res = Fib(n, i+1);    }
  return res;
}
```

Output

```
5
F: m1=1, m2=1, n=5, i=2
F: &m1=134518656, &m2=134518660
F: &res=3221224516, &temp=3221224512
F: m1=2, m2=1, n=5, i=3
F: &m1=134518656, &m2=134518660
F: &res=3221224468, &temp=3221224464
F: m1=3, m2=2, n=5, i=4
F: &m1=134518656, &m2=134518660
F: &res=3221224420, &temp=3221224416
F: m1=5, m2=3, n=5, i=5
F: &m1=134518656, &m2=134518660
F: &res=3221224372, &temp=3221224368
F(5) = 8
```

Common Errors in Writing Recursive Functions

- Non-terminating Recursive Function (Infinite recursion)
  - No base case

```
int badFactorial(int x) {
  return x * badFactorial(x-1);
}
```

  - The base case is never reached

```
int anotherBadFactorial(int x) {
  if(x == 0)
    return 1;
  else
    return x*(x-1)*anotherBadFactorial(x-2);
    // When x is odd, base case never reached!!
}
```

```
int badSum2(int x)
{
  if(x==1) return 1;
  return(badSum2(x--));
}
```

Common Errors in Writing Recursive Functions

- Mixing up loops and recursion

```
int anotherBadFactorial(int x) {
   int i, fact = 0;
   if (x == 0)
       return 1;
   else {
       for (i=x; i>0; i=i-1) {
             fact = fact + x*anotherBadFactorial(x-1);
       }
       return fact;
   }
}
```

- In general, if you have recursive function calls within a loop, think carefully if you need it. Most recursive functions you will see in this course will not need this

Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance
  - Choice between performance (iteration) and good software engineering (recursion).

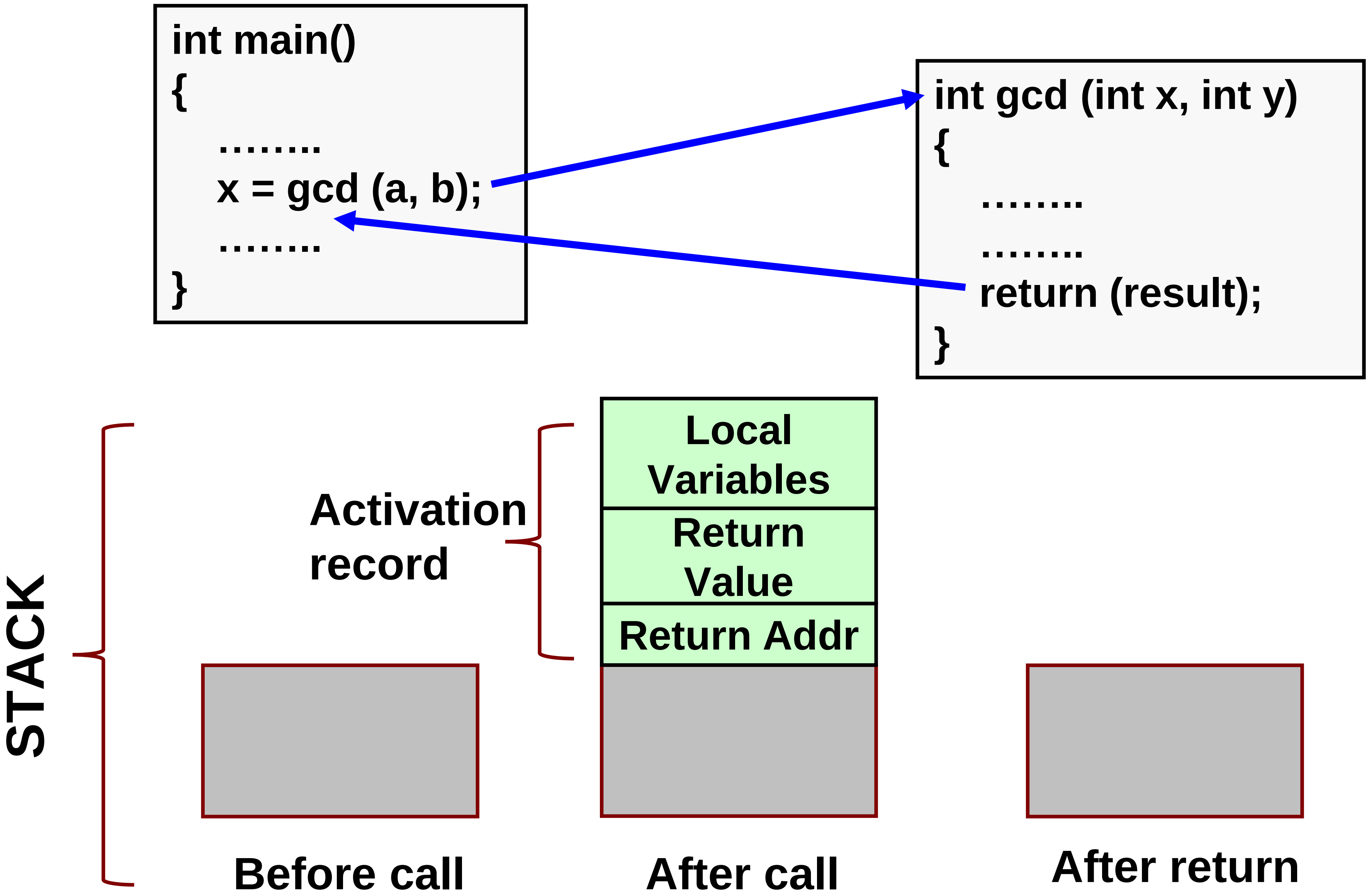
- Every recursive program can also be written without recursion
- Recursion is used for programming convenience, not for performance enhancement
- Sometimes, if the function being computed has a nice recursive form, then a recursive code may be more readable

Recursion

Part III (Lecture 17)

How are function calls implemented?

- The following applies in general, with minor variations that are implementation dependent
  - The system maintains a stack in memory
    - Stack is a last-in first-out structure
    - Two operations on stack, push and pop
  - Whenever there is a function call, the activation record gets pushed into the stack
    - Activation record consists of the return address in the calling program, the return value from the function, and the local variables inside the function

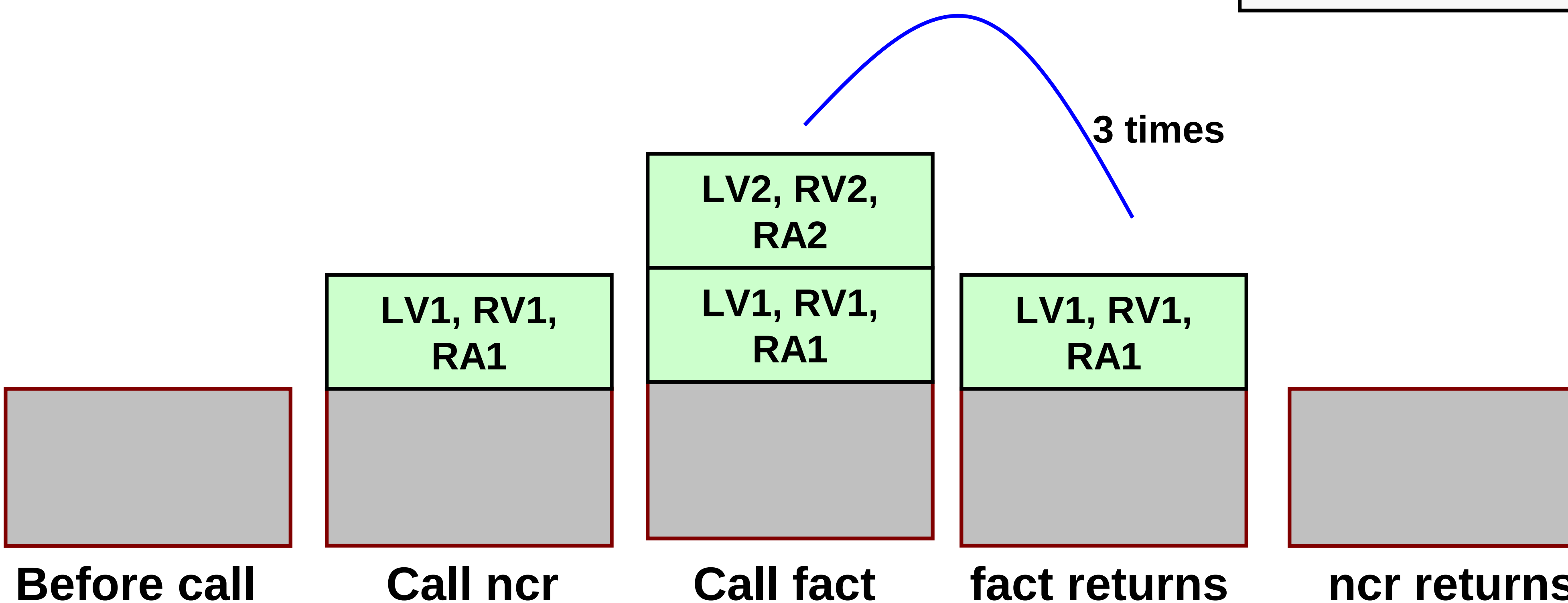

int main()
{
....
x = gcd (a, b);
....
}
int gcd (int x, int y)
{
....
....
return (result);
}
STACK
Activation record
Local Variables
Return Value
Return Addr
Before call
After call
After return

```
int main()
{
    ……..
    x = ncr (a, b);
    ……..
}
```

```
int ncr (int n, int r)
{
    return (fact(n)/
        fact(r)/fact(n-r));
}
```

3 times

```
int fact (int n)
{
    ………
    return (result);
}
```

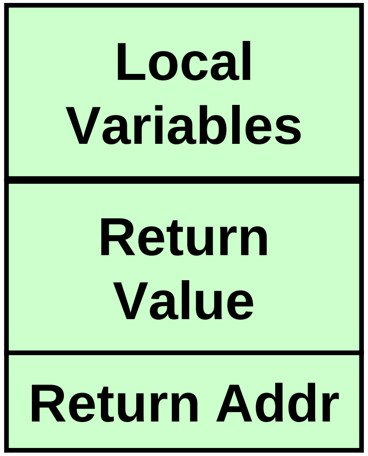


What happens for recursive calls?

- What we have seen ….
  - Space for activation record is allocated on the stack when a function call is made
  - Space allocated for activation record is de-allocated on the stack when the function returns
- In recursion, a function calls itself
  - Several function calls going on, with none of the function calls returning back
    - Space for activation records allocated on the stack continuously
    - Large stack space required

- Space for activation records are de-allocated, when the termination condition of recursion is reached

- We shall illustrate the process by an example of computing factorial
  - Activation record looks like:

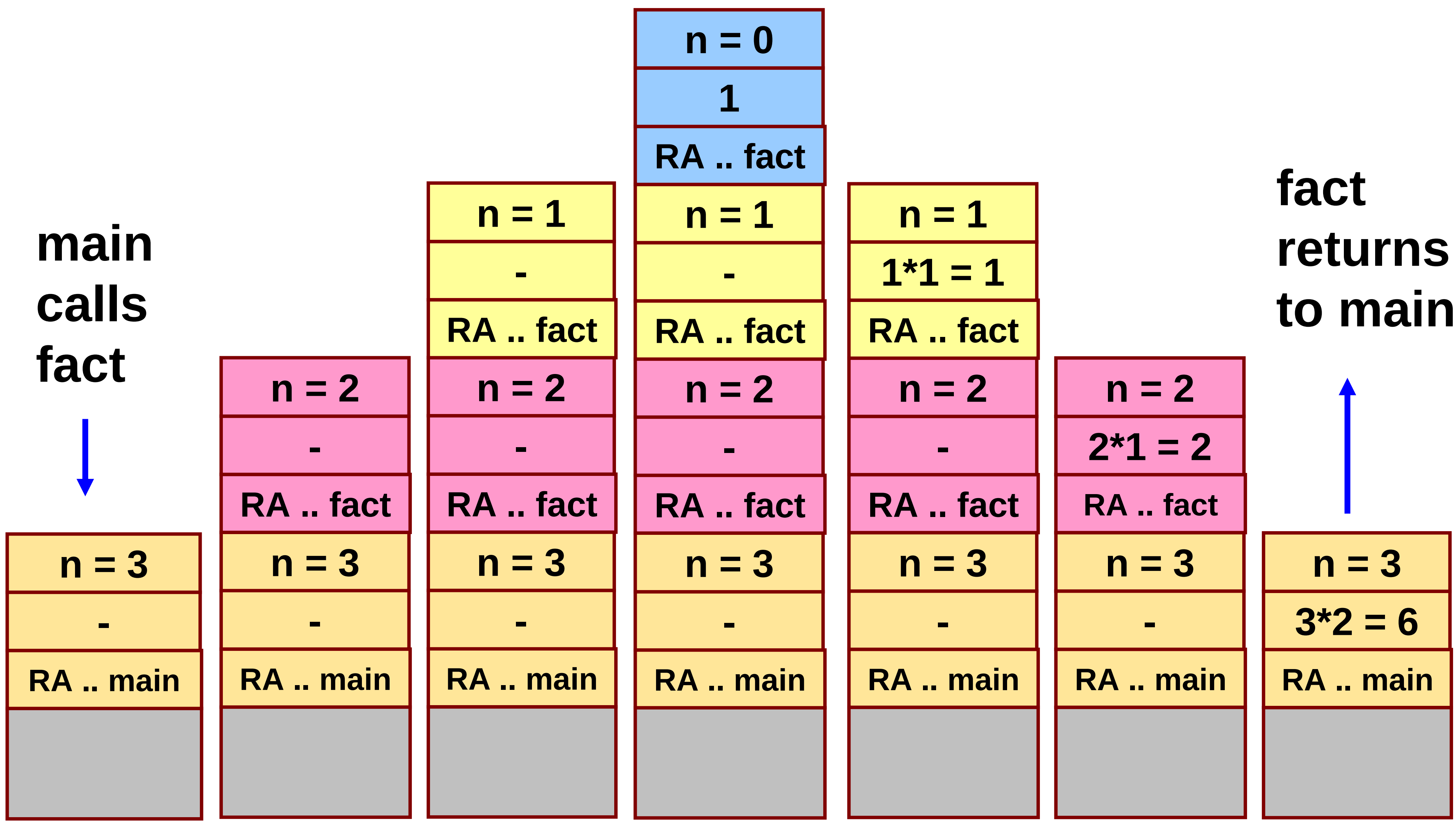


Example:: main() calls fact(3)

```
int main()
{
   int  n;
   n = 3;
   printf ("%d \n", fact(n) );
   return 0;
}
```

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

TRACE OF THE STACK DURING EXECUTION



Do Yourself

- Trace the activation records for the following version of Fibonacci sequence

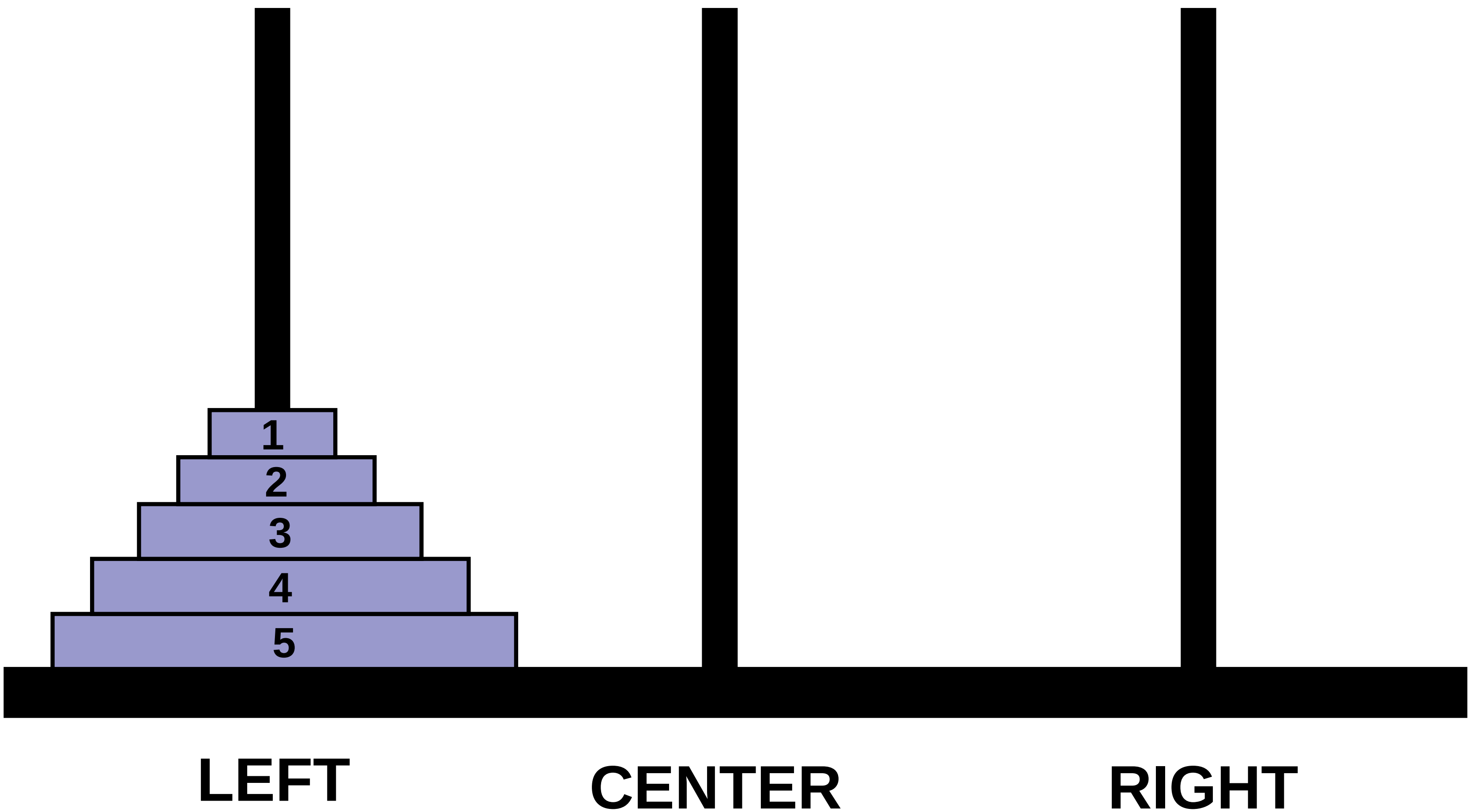
```
      int   f (int n)
      {
          int a, b;
          if  (n  < 2)   return (n);
          else  {
X           a = f(n-1);
            b = f(n-2);
Y           return (a+b);
          }
      }

      void main() {
          printf("Fib(4) is: %d \n", f(4));
main  }
```

Local Variables (n, a, b)
Return Value
Return Addr (either main, or X, or Y)

Additional Example

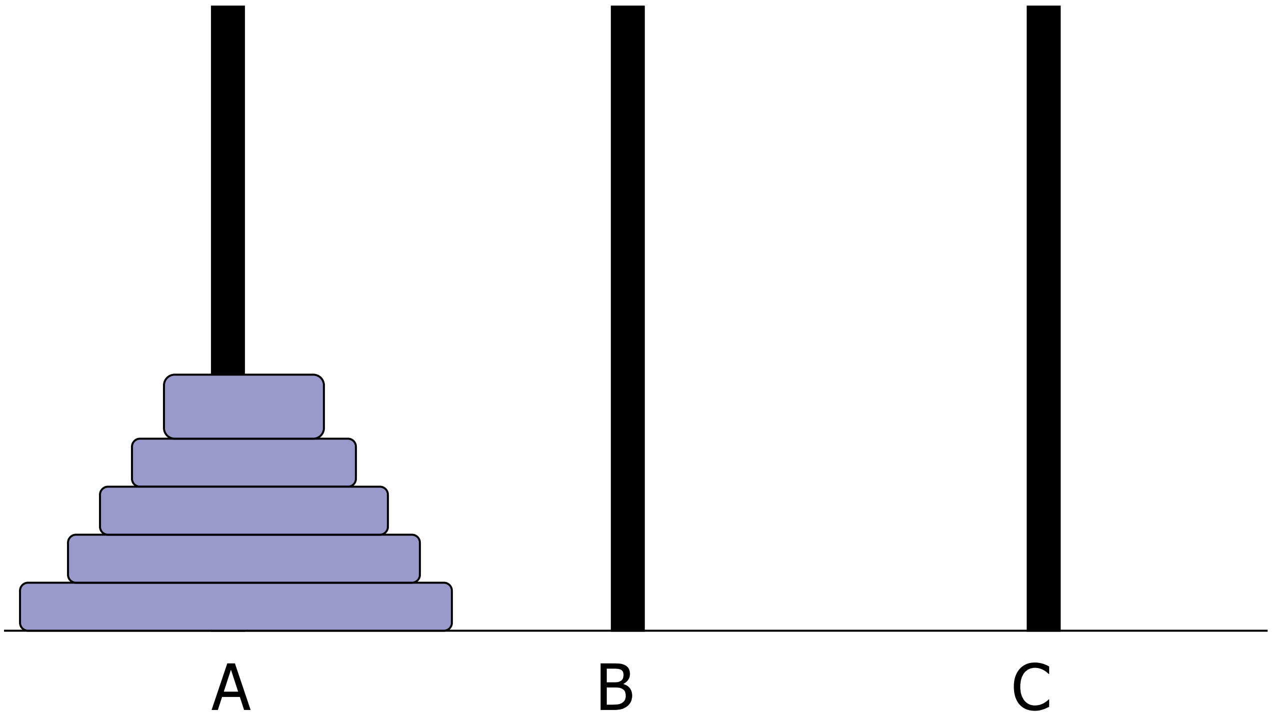
Towers of Hanoi Problem



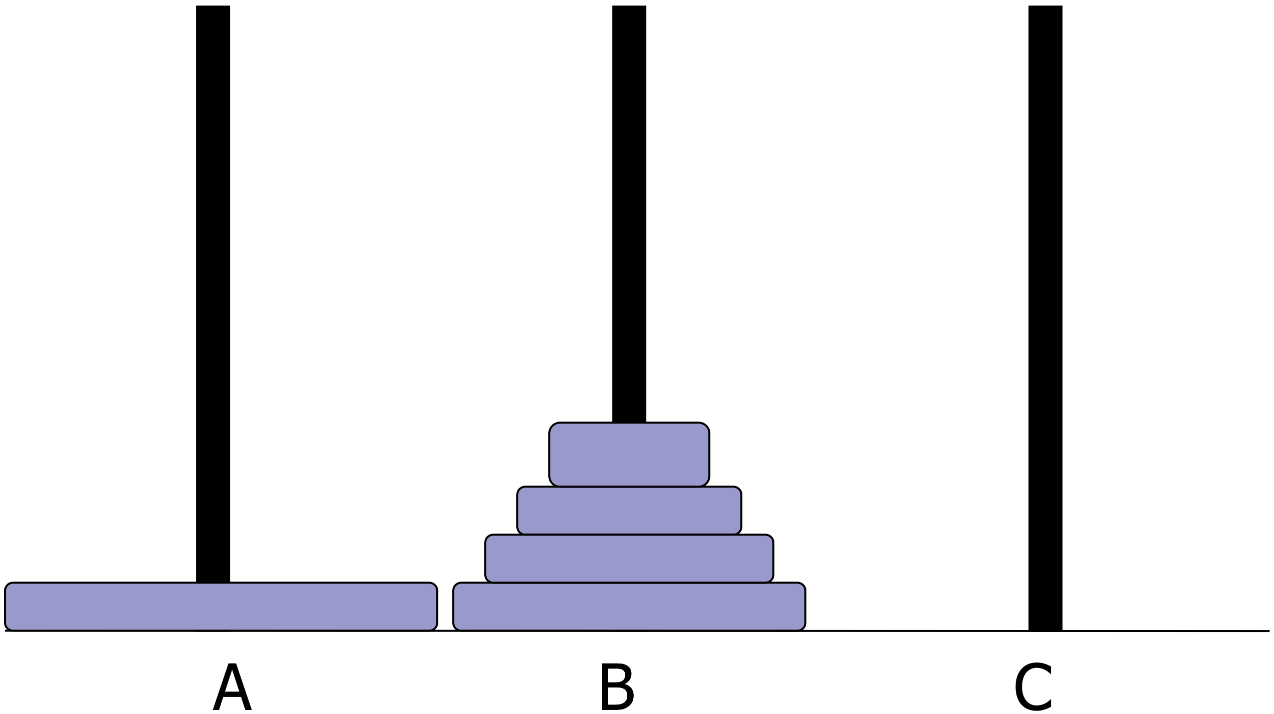
- Initially all the disks are stacked on the LEFT pole
- Required to transfer all the disks to the RIGHT pole
  - Only one disk can be moved at a time.
  - A larger disk cannot be placed on a smaller disk
- CENTER pole is used for temporary storage of disks

- Recursive statement of the general problem of n disks
  - Step 1:
    - Move the top (n-1) disks from LEFT to CENTER
  - Step 2:
    - Move the largest disk from LEFT to RIGHT
  - Step 3:
    - Move the (n-1) disks from CENTER to RIGHT

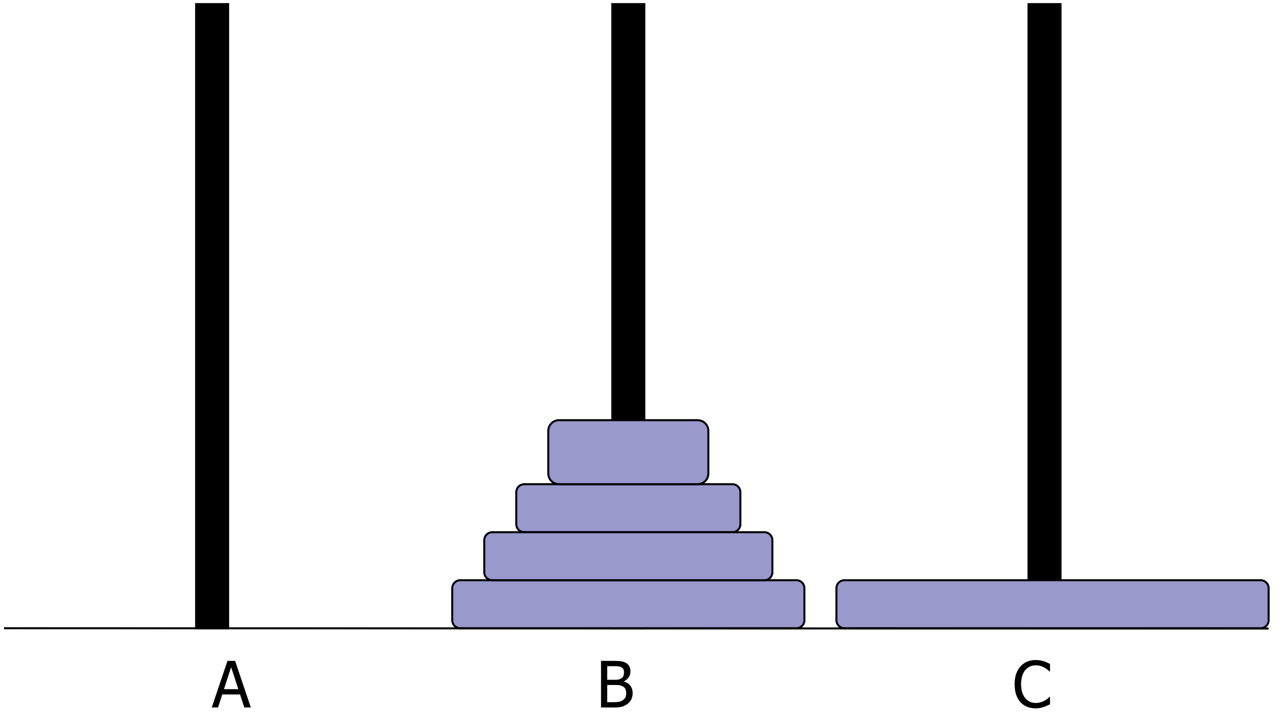
Tower of Hanoi



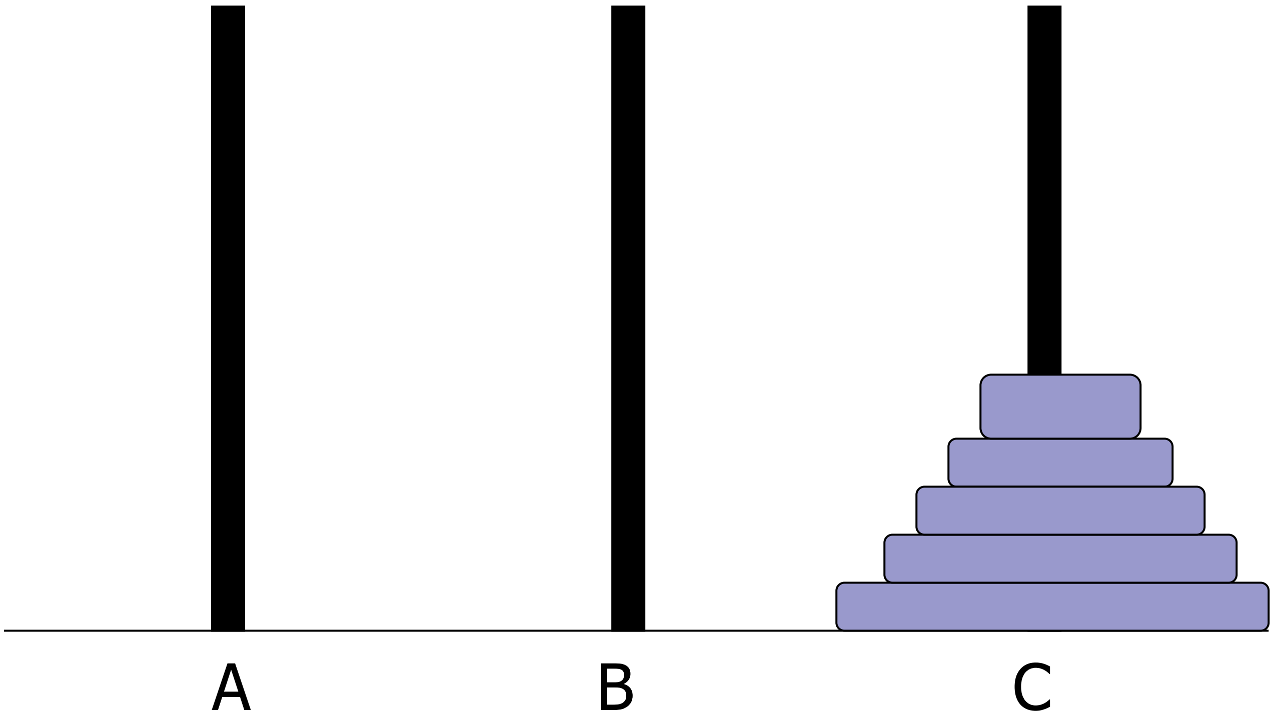
Tower of Hanoi



Tower of Hanoi



Tower of Hanoi



Towers of Hanoi function

```
void towers (int n, char from, char to, char aux)
{
 /* Base Condition */
 if (n==1)   {
          printf ("Disk 1 : %c -> &c \n", from, to) ;
          return ;
     }
    /* Recursive Condition */
        towers (n-1, from, aux, to) ;
        ……………………….
        ……………………….
}
```

Towers of Hanoi function

```
void towers (int n, char from, char to, char aux)
{
 /* Base Condition */
 if (n==1)   {
          printf ("Disk 1 : %c -> &c \n", from, to) ;
          return ;
     }
    /* Recursive Condition */
        towers (n-1, from, aux, to) ;
        printf ("Disk %d : %c -> %c\n", n, from, to) ;
        ……………………….
}
```

Towers of Hanoi function

```
void towers (int n, char from, char to, char aux)
{
 /* Base Condition */
 if (n==1)   {
          printf ("Disk 1 : %c -> %c \n", from, to) ;
          return ;
     }
    /* Recursive Condition */
        towers (n-1, from, aux, to) ;
        printf ("Disk %d : %c -> %c\n", n, from, to) ;
        towers (n-1, aux, to, from) ;
}
```

TOH runs

```
void towers(int n, char from, char to, char aux)
{ if (n==1)
 {  printf ("Disk 1 : %c -> %c \n", from, to) ;
    return ;
 }
   towers (n-1, from, aux, to) ;
   printf ("Disk %d : %c -> %c\n", n, from, to) ;
   towers (n-1, aux, to, from) ;
}
int main()
{ int n;
  scanf("%d", &n);
  towers(n,'A',‘C',‘B');
  return 0;
}
```

Output

```
3
Disk 1 : A -> C
Disk 2 : A -> B
Disk 1 : C -> B
Disk 3 : A -> C
Disk 1 : B -> A
Disk 2 : B -> C
Disk 1 : A -> C
```

More TOH runs

```
void towers(int n, char from, char to, char aux)
{ if (n==1)
 {  printf ("Disk 1 : %c -> %c \n", from, to) ;
    return ;
 }
   towers (n-1, from, aux, to) ;
   printf ("Disk %d : %c -> %c\n", n, from, to) ;
   towers (n-1, aux, to, from) ;
}
int main()
{ int n;
  scanf("%d", &n);
  towers(n,'A',‘C',‘B');
  return 0;
}
```

Output

```
4
Disk 1 : A -> B
Disk 2 : A -> C
Disk 1 : B -> C
Disk 3 : A -> B
Disk 1 : C -> A
Disk 2 : C -> B
Disk 1 : A -> B
Disk 4 : A -> C
Disk 1 : B -> C
Disk 2 : B -> A
Disk 1 : C -> A
Disk 3 : B -> C
Disk 1 : A -> B
Disk 2 : A -> C
Disk 1 : B -> C
```

Practice Problems

1. Write a recursive function to search for an element in an array
2. Write a recursive function to count the digits of a positive integer (do also for sum of digits)
3. Write a recursive function to reverse a null-terminated string
4. Write a recursive function to convert a decimal number to binary
5. Write a recursive function to check if a string is a palindrome or not
6. Write a recursive function to copy one array to another

Note:

- For each of the above, write the main functions to call the recursive function also
- Practice problems are just for practicing recursion, recursion is not necessarily the most efficient way of doing them