Lecture 2: Recursion

Read: Chpt.2, Carrano.

Motivation: Many object/structure can be defined naturally using recursive definitions, and recursive definition can be implemented easily in programming languages that support recursion.

Q: What is recursive definition?

A definition used to define an object/structure in terms of its own (object/structure).

Format:

- **Base case**: Define the object directly when the object size is small.
- **Recursive case**: Define the object in terms of its own (object) but with a smaller size.

Implementation of Recursive Definition:

Recursive Definition		**Recursive Algorithm**
Base case(s)	⟶	Terminating condition(s)
Recursive defn.	⟶	Recursive function call(s)

Warning: The base case must be *reachable* in order to terminate the recursive definition.

Examples of Recursive Definitions:

1. Factorial Function:

Defn: Given a non-negative integer $n \geq 0$.

$0! = 1,$

$n! = 1*2*3*\ldots* (n-1)*n,\ n > 0.$

Recursive Definition of Factorial Functions:

Defn: Given a non-negative integer $n \geq 0$.

$0! = 1,$ (Base case)

$n! = 1*2*3*\ldots* (n-1)*n$

$= n*[(n-1)*\ldots\ldots(n-1)*n]$

$= n*(n-1)!,\ n > 0.$ (Recursive case)

Example:

Iterative computation of 3!:

$3! = 1*2*3$

$= 6$

Recursive Computation of 3!:

$3! = 3*2!$

$= 3*(2*1!)$

$= 3*(2*(1*0!))$

$= 3*(2*(1*1))$

$= 3*(2*1)$

$= 3*2$

$= 6$

Computing Factorial Function Iteratively:

```
// Compute the factorial function of nonnegative integer n
// Precondition: n is a nonnegative integer
// Postcondition: Return n!
    int factorial(int n)          // iterative program
    {
       if (n == 0)
             return 1;
       else
            {   int  product = 1;
                for (int index = 2; index <= n; index++)
                      product = product * index;
            }
       return product;
    }
```

Computing Factorial Function Recursively:

```
    int rfactorial(int n)         // recursive program
    {
       if (n == 0)                // base case
             return 1;
       else                       // Recursive case
             return n*rfactorial(n-1);
    }
```

2. Fibonacci Sequence Numbers:

Consider the following sequence of integers.

f_0	f_1	f_2	f_3	f_4	f_5	f_6	…	f_n	…
0	1	1	2	3	5	8		?	

Defn: Given a non-negative integer $n \geq 0$.

$f_0 = 0$, $f_1 = 1$,

$f_n = f_{n-1} + f_{n-2}$, $n > 1$.

Example:

Iterative Computation of f_4:

$f_0 = 0$,

$f_1 = 1$,

$f_2 = f_1 + f_0 = 1$,

$f_3 = f_2 + f_1 = 1 + 1 = 2$,

$f_4 = f_3 + f_2 = 2 + 1 = 3$

Recursive Computation of f_4:

$f_4 = f_3 + f_2$,

$= (f_2 + f_1) + f_2$

$= ((f_1 + f_0) + f_1) + f_2$

$= ((1 + f_0) + f_1) + f_2$

$= ((1 + 0) + f_1) + f_2$

$= (1 + 1) + f_2$

$= 2 + (f_1 + f_0)$

$= 2 + (1 + f_0)$

$= 2 + (1 + 0)$

$= 2 + 1$

$= 3$

Computing Fibonacci Number Iteratively:

```
// Compute the n-term Fibonacci number for nonnegative
// integer n
// Precondition: n is a nonnegative integer
// Postcondition: Return nth Fibonacci number
int fib(int n)
{  int first;              // initialize f_{n-2}
   int second;             // initialize f_{n-1}
   int next;               // for computing f_n, n > 1
   int count;              // counter for computing f_2, f_3, …, f_n
   if (n == 0)             // base case: f_0 = 0
       return 0;
   else
       {  if (n == 1)      // base case: f_1 = 0
             return 1;
          else
             {  first = 0;
                second = 1;
                for (count = 2; count <= n; count++)
                    {  next = first + second;
                       first = second;
                       second = next;
                    }
                return next;
             }
       }
}
```

Computing Fibonacci Number Recursively:

```
int rfib(int n)                  // recursive program
{
      if (n == 0)                // base case
            return 0;
      else
            {  if (n == 1)       // another base case
                  return 1;
               else                // recursive case
                  return rfib(n-1) + rfib(n-2);
            }
 }
```

3. Recursive Definition of (Rooted) Tree:

Defn. Given a set T of n objects, $n \geq 0$.
If $n = 0$, then T is a tree.
If $n > 0$, then there is a distinct element r in T, called the root of T, such that $T - \{r\}$ can be partitioned into 0 or more trees $T_1, T_2, \ldots, T_k$, $k \geq 0$.

Basic Concepts of Tree:

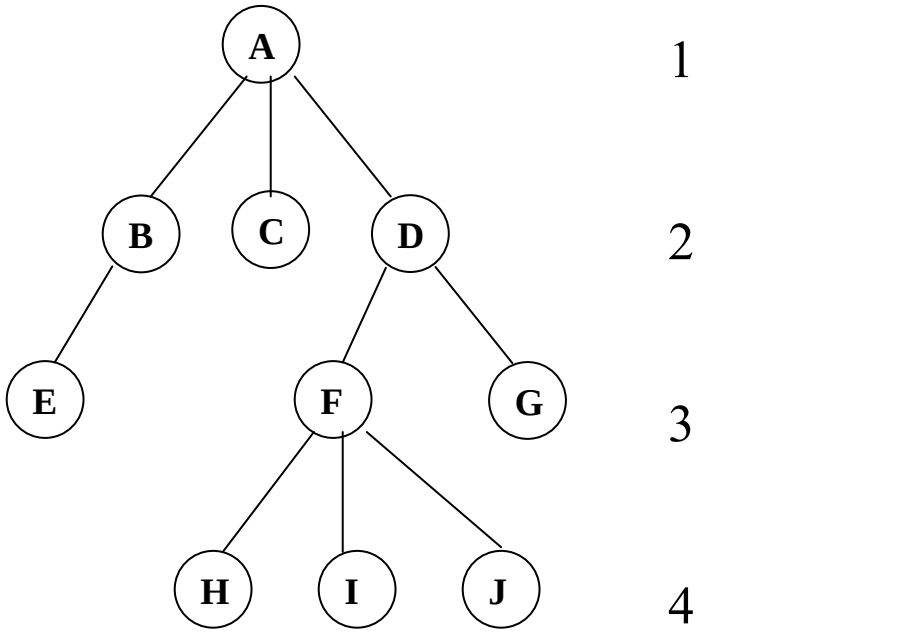
- Each tree T_i, $1 \leq i \leq k$, is called a subtree of r.
- The roots of the subtrees of r are ***children*** *of r*.
- The root r is the ***parent*** of the roots of its subtrees.
- Objects in T are ***nodes*** in the tree T.
- Nodes with the same parents are ***siblings***.
- Nodes with no children are ***leaves***.
- A ***path of length m*** (in a tree) from node x to node y is a sequence of m+1 nodes $x = n_0, n_1, n_2, ..., n_k = y$ such that n_i is the parent of n_{i+1} for all $i = 0, 1, ..., m-1$. (Observe that there is a unique path from the root to each node in the tree. Also, there is a path of zero length from any node to itself.)
- If there exists a path from x to y, then x is an ***ancestor*** of y and y is a ***descendant*** of x.
- For any given node x, the ***depth*** **(*level*)** of x is the number of nodes on the unique path from the root to x, and the ***height*** of x is the number of nodes of a longest path from x to a leaf.
- The ***height of a tree*** is the height of its root.

Graphical Representation:

Node $\Leftrightarrow$ Object

Line from node x to node y
$\Leftrightarrow$ Parent-child relation

Example: A tree T.



A is the ***root*** of T;
H, I, J are ***children*** of F;
D is the ***parent*** of F, G;
B, C, D are ***siblings***;
C, E, G, H, I, J are ***leaves***;
(A,D,F,H) is a path of length 3;
height of T is 4.

4. Recursive Definition of k-ary Trees, k ≥ 2:

Given a set T of n objects, n ≥ 0, and a fixed positive integer k ≥ 2.
If n = 0, then T is a k-ary tree.
If n > 0, then there is a distinct element r in T, called the root of T, such that T−{r} can be partitioned into at most k k-ary trees T_1, T_2, …, T_k.

Observe that when k = 2, T is a binary tree.

Remarks:

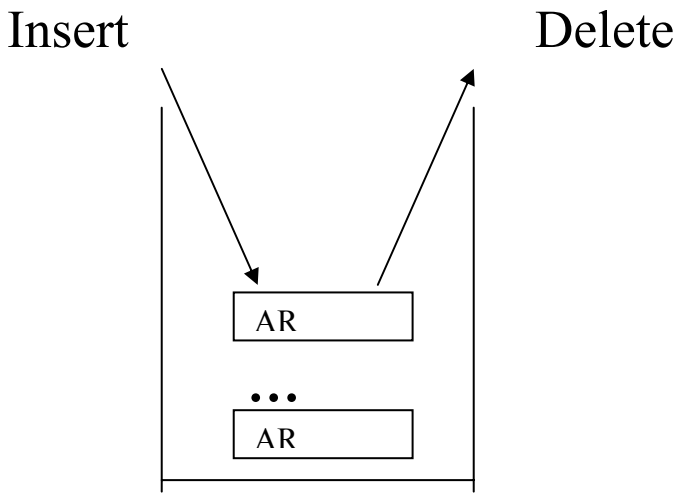
- Recursive algorithms are powerful computational tool in problem solving because they are easy to state, comprehend, implement, and they also support the implementation of Divide-and-Conquer.
- **Warning:** Be aware of potentially large running time (exponential complexity) and infinite loop!

The Four Characteristics of Recursive Algorithms:

1. The algorithm computes a solution to a general problem by combining the solutions of one or more identical, but smaller, subproblems.
2. The algorithm invokes itself (recursive call) to compute a solution to the subproblems.
3. There exist one or more base cases for which solution(s) can be computed directly without any further recursion.
4. The base case(s) must be reachable to guarantee termination of the recursive algorithm.

Executing recursive algorithm using a stack:

Whenever a function is called, an activation record (AR) is created to store the current status of that function, which include the values of its parameters, contents of registers, the function's return value, local variables, and the address of the instruction to which execution is to be continued when it finishes execution. This AR will be pushed (added) onto a stack, which is a (**L**ast-**I**n-**F**irst-**O**ut, **LIFO**) data structure in which insertion and deletion can only be carried out at one end.



Stack

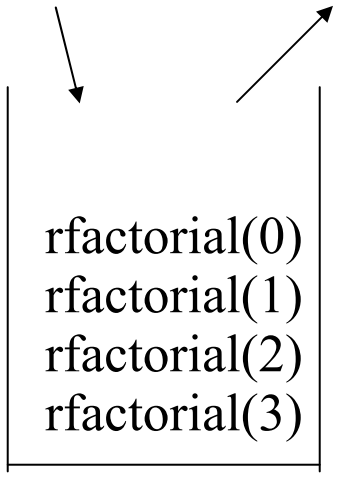
Remark: By simulating the action of a stack, any recursive program can be transformed into an iterated algorithm with no recursion.

Example: Computing 3! using the rfactorial.

```
3!                      // call rfactorial(3)
=  (3*2!)               // call rfactorial(2)
=  (3*(2*1!))           // call rfactorial(1)
=  (3*(2*(1*0!)))       // call rfactorial(0)
=  (3*(2*(1*(1))))      // 0! = 1, return 1
=  (3*(2*(1)))          // 1! = 1, return 1*1 = 1
=  (3*(2))              // 2! = 2, return 2*1 = 2
=  6                    // return 3*2 = 6
```

Stack Structure:

Insert



Stack

Delete stack items:

0! = 1
1! = 1*0! =1
2! = 2*1! =2
3! = 3*2! =6

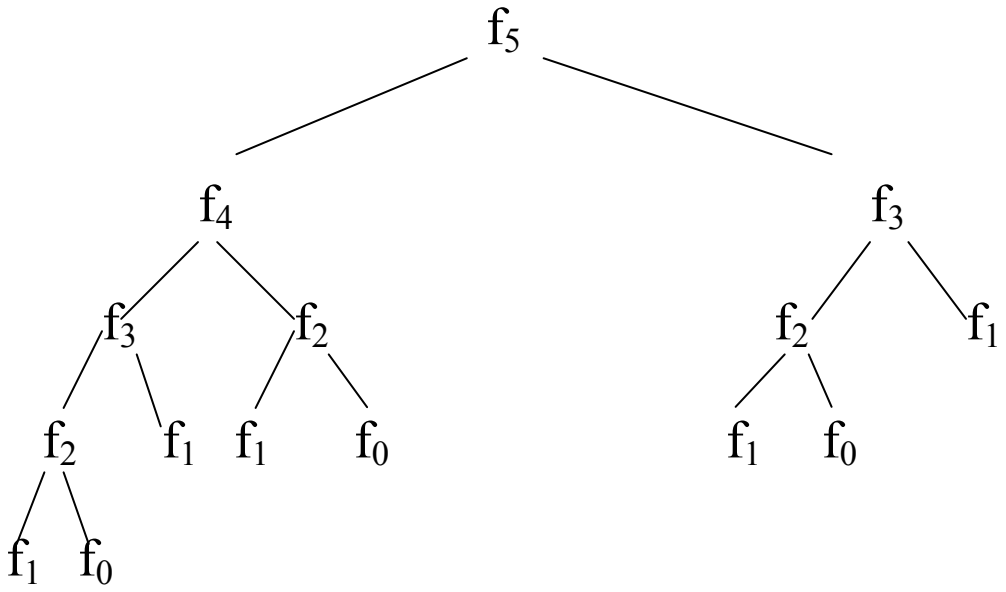
Efficiency of Recursive Algorithms:

Tracing a Recursive Algorithm using Recursion Tree:

A recursion tree is a k-ary tree T used to model the execution of a recursive function such that

- Each node in T corresponds to an execution of the function.
- T has two types of nodes:
  - Non-leaf nodes: Recursive call to the function.
  - Leaf nodes: Terminating condition is reached.

Consider the recursion tree for the computation of f_5 using rfib.



Observe that each (non-leaf) node corresponds to a recursive call to rfib and there are many re-computations in the recursive program! In general, it will require exponentially, 2^n, that many steps in computing f_n.

Using Recursion Tree with Box Diagrams:

A recursion tree with boxes and transitions is a recursion tree T such that

- Each node in T is replaced with a box with information on current function
- Each box contains:
    1. Formal parameters
    2. Local variables
    3. A placeholder for the value returned by the function call
    4. Value of the function.
- Each transition corresponds to a recursive call. If more than one recursive calls in the function, each transition can be labeled with the corresponding call.

Consider the following problem.

Given a pair of new-born (M/F) rabbits, we assume that, after two months, this pair of rabbits will give birth to another pair of rabbits (M/F). For simplicity, we assume that the rabbits will never die and this process will go on indefinitely.

Q: How many pairs of rabbits do we have in n months?

Let a_n be the number of pairs of rabbits we have after n months.

Observe that

$a_0 = 0$, $a_1 = 1$, $a_2 = 1$ (before breeding starts), $a_3 = 2$,

…

$a_n = a_{n-1} + a_{n-2}$, $n > 2$.

Example:

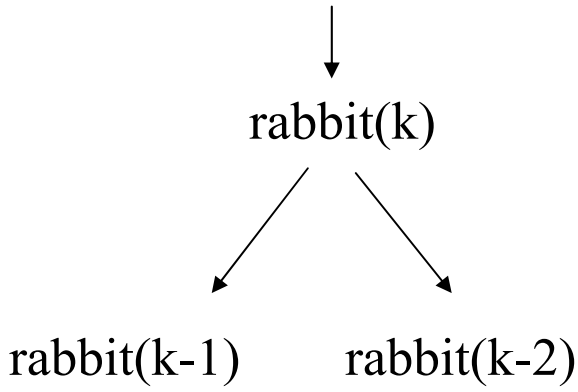
Month: 0	1	2	3	4	5	6	7	…
Pair Rabbits:	0	1	1	2	3	5	8	

Observe that a_n is simply the nth term Fibonacci sequence number we computed before!

```
// Compute the number of pairs of rabbits after n months
// Precondition: n is a positive integer
// Postcondition: Return # of pairs of rabbits after n months
   int rabbit(int n)
   {
        if (n <= 2)                    // base case
             return 1;
        else
             return rabbit(n-1) + rabbit(n-2);

   }  // end rabbit
```

General Recursion Tree Structure:

For k > 2:



Non-Leaf Node:

```
n

rabbit(n)

return
   rabbit(n-1) +
      rabbit(n-2)
```

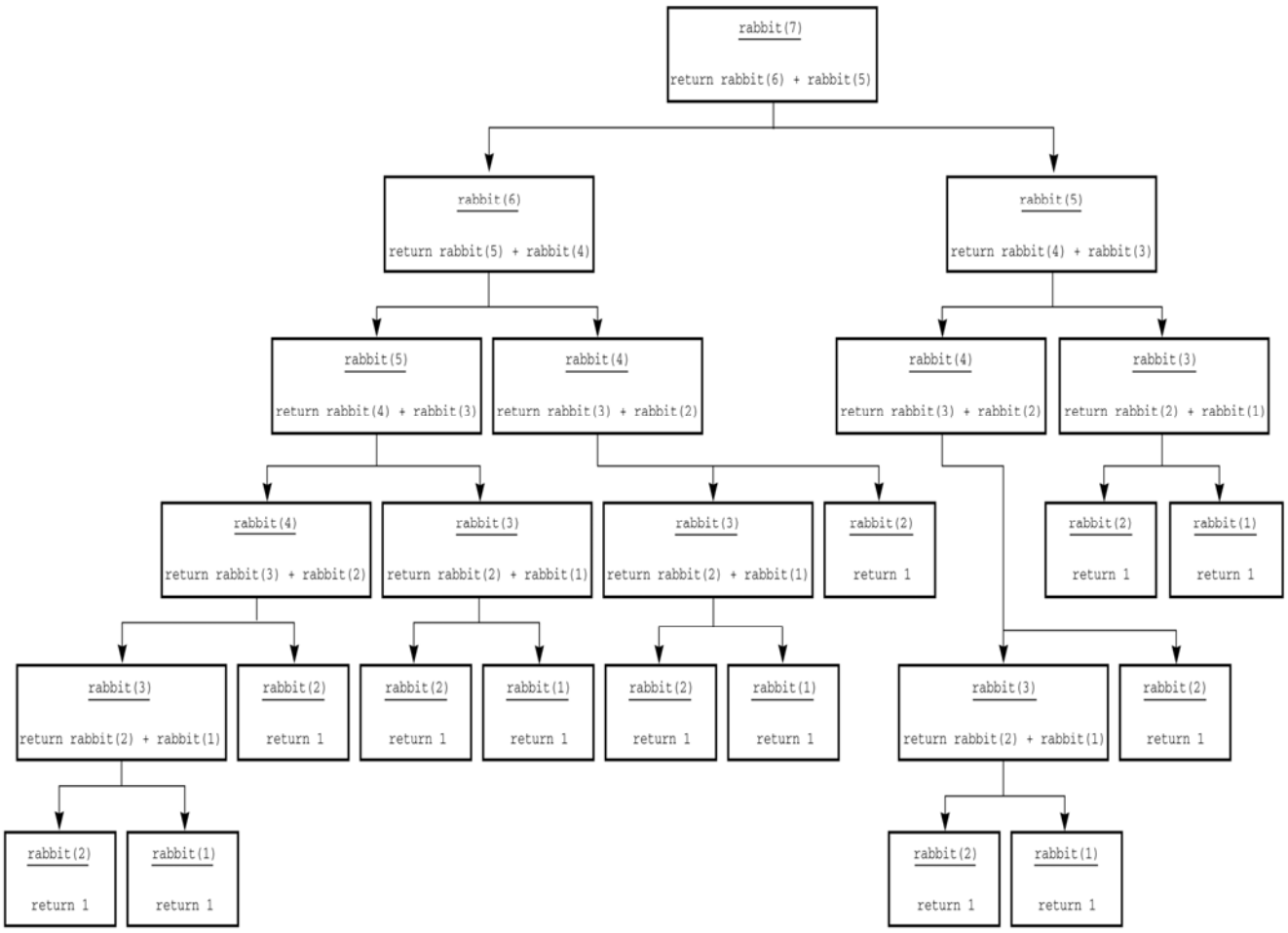
rabbit(n)

Leaf-Node:

```
n

rabbit(n)

return 1
```

rabbit(1)

Example: Recursion Tree for Computing rabbit(7) (P.88).



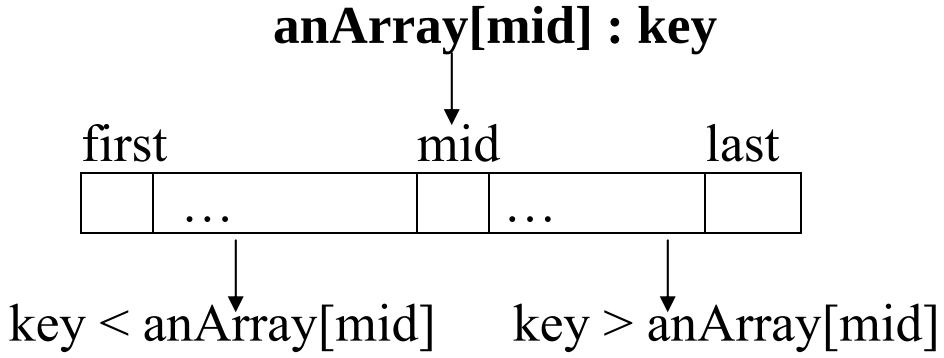
More Examples on Recursion:

1. Binary Searching an Ordered Array:

Given a sorted array anArray[first..last] and a key.
Return array index k, first ≤ k ≤ last, such that anArray[k] = key, if exists; otherwise, return -1.

Prototype:

int bsearch(const int anArray[], int first, int last, int key);



Recursive Algorithm:

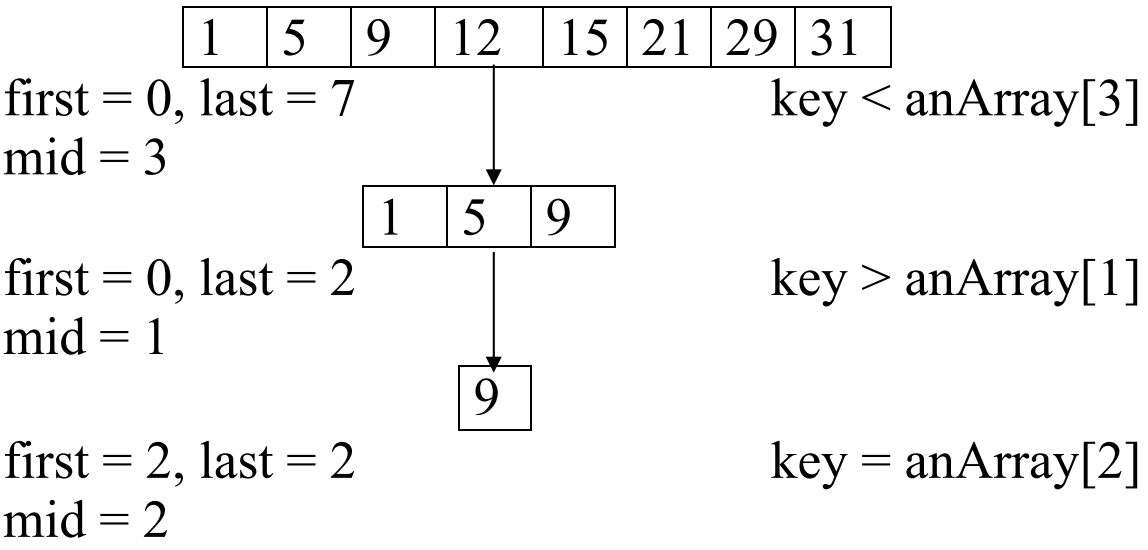
```
mid = (first+last)/2;
if key = anArray[mid]
   return mid;
else if key < anArray[mid]
   bsearch(anArray, first, mid-1, key);
else                                    // if x > anArray[mid]
   bsearch(anArray, mid+1, last, key);
```

Warning: Where is the terminating condition?

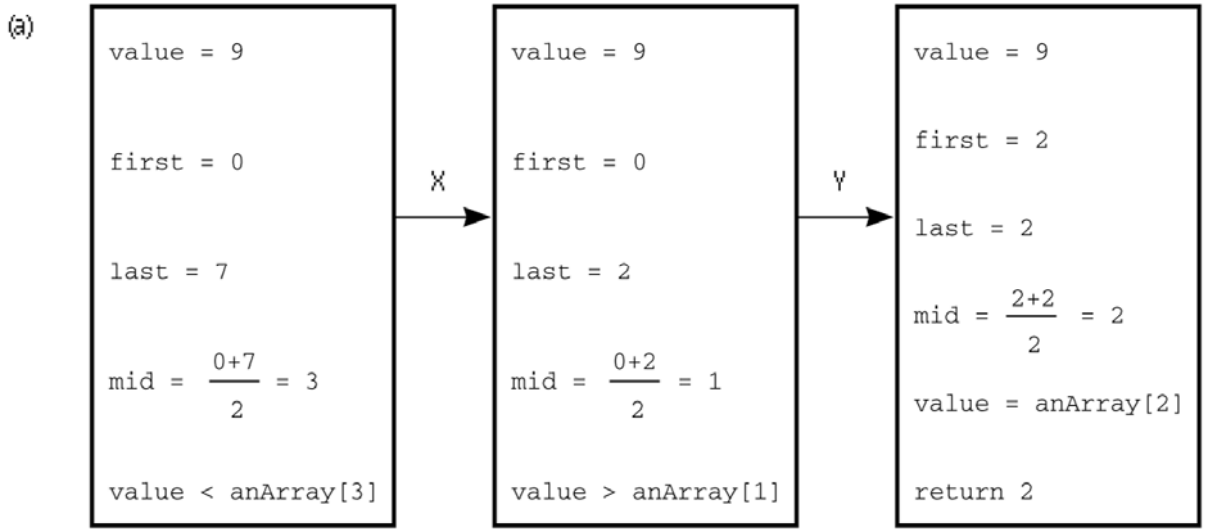
Recursive Binary Search Algorithm:

```
int bsearch(const int anArray[], int first, int last, int key)
// Use binary search to search an integer array from
// anArray[first] to anArray[last] for integer key.
// Precondition:  0 <= first, last <= size – 1, where size is
// the max size of array.
// Postcondition: If key is found, return array index; else
// return -1.
{
   int index;

   if (first > last)                    // base case; key not found
      index = -1;
   else
   {
      int mid = (first+last)/2;      // compute mid for dividing

      if (key == anArray[mid])      //     key found
      {
         index = mid;
      }
      else if (key < anArray [mid])   // search left sub-array
               index = bsearch(anArray, first, mid−1, key);
            else                                 // search right sub-array
               index = bsearch(anArray, mid+1, last, key);
   }
   return index;
} // end bsearch
```

Example: Using binary search to search an array anArray = [1, 5, 9, 12, 15, 21, 29, 31] with key = 9.



Recursion Tree for Binary Search:



Iterative Binary Search Algorithm:

```
int bsearch(const int anArray[], int first, int last, int key)
{  while (first <= last)
   {  int mid = (first+last)/2;
      if (anArray[mid] = key)
          return mid;
      else if (key < anArray[mid])
                last = mid–1;
            else
                first = mid+1;
   }
   return –1;
}  // end bsearch
```

2. Computing Ackermann's Function:

Let N be the set of nonnegative integers.

Ackermann's function $A : N \times N \rightarrow N$ is defined by

$A(0, n) = n + 1,$

$A(m, 0) = A(m-1, 1), \text{ if } m > 0,$

$A(m, n) = A(m-1, A(m, n-1)), \text{ if } m, n > 0.$

This is an extremely fast growing function often found in analyzing the performance of data structures and counting.

Example:

$A(0, 0) = 1,$
$A(0, 1) = 2,$
$A(0, 2) = 3,$
$A(0, 3) = 4,$
…

$A(1, 0) = A(0, 1) = 2,$
$A(1, 1) = A(0, A(1, 0)) = A(0, 2) = 3,$
$A(1, 2) = A(0, A(1, 1)) = A(0, 3) = 4,$
$A(1, 3) = 5,$
…

$A(2, 0) = A(1, 1) = 3,$
$A(2, 1) = A(1, A(2, 0)) = A(1, 3) = 5,$
$A(2, 2) = A(1, A(2, 1)) = A(1, 5) = 7,$
$A(2, 3) = 9,$
…
$A(3,0) = A(2, 1) = 5,$
$A(3, 1) = A(2, A(3,0)) = A(2, 5) = 13,$
$A(3, 2) = A(2, A(3, 1)) = A(2, 13) = 29,$
$A(3, 3) = A(2, A(3, 2)) = A(2, 29) = 61,$
…

```
int Ackermann(int m, int n)
{
   int value;
   if (m ==0)
      value = 1;
   else
      {  if (n == 0)
            value = Ackermann(m-1, 1);
         else
             value = Ackermann(m-1, Ackermann(m, n-1));
      }
   return value;
}  // end Ackermann
```

Remark: The inverse of the Ackerman's function is an extremely slow growing function that governs the performance of many important data structures and algorithms.

3. Tower of Hanoi Problem:

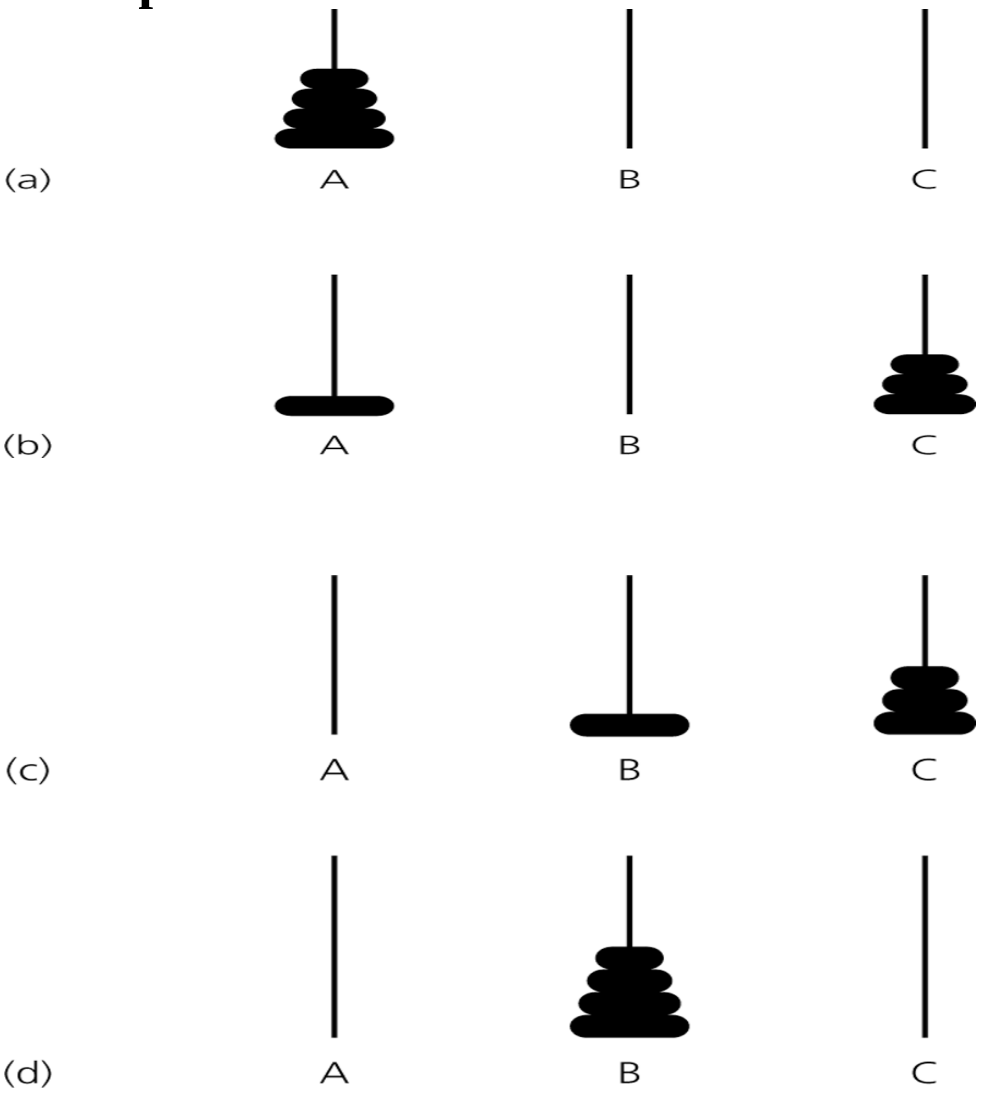
Given 3 poles (labeled A, B, and C) and n different sized disks sorted on pole A such that the smallest (largest) disk is on top (bottom).

Q: How fast can we move all the disks from pole A to pole B such that

(1) We can move only one disk at a time.

(2) No larger disk can be on top of a smaller one.

Example:



Let H_n be the # of moves required to move n disk from pole i to peg j, $i \neq j$.

Observe that

$H_1 = 1,$

$H_n = H_{n-1} + 1 + H_{n-1}$

$= 2H_{n-1} + 1, n > 1.$

$H_n = 2H_{n-1} + 1$

$= 2(2H_{n-2} + 1) + 1$

$= 2^2H_{n-2} + 2 + 1$

$= 2^2(2H_{n-3} + 1) + 2 + 1$

$= 2^3H_{n-3} + 2^2 + 2^1 + 2^0$

$= \ldots$

$= 2^{n-1}H_1 + 2^{n-2} + 2^{n-3} + \ldots + 2^1 + 2^0$

$= 2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 2^1 + 2^0$

$= 2^n - 1$

In the original Tower of Hanoi problem, n = 64.
Hence,

$H_n = 2^{64} - 1$

$= 18,446,744,073,709,551,615$

If we can move one disk per second, it will still take us more than 500 billion years to move 64 disks from pole A to pole B!

```
Towers(int count, char source, char destination, char spare)
{
   if (count == 1)
      {  cout  << "Move top disk from pole " << source
               << "to pole " << destination << endl;
      }
   else
      {  Towers(count-1, source, spare, destination);
         Towers(1, source, destination, spare);
         Towers(count-1, spare, destination, source);
      }
}  // end Towers
```

Exponential Behavior of Recursive Algorithms:

1. **Caused by Re-computations:**
   Exponential complexity can be eliminated by removing/minimizing redundant computations.
   Example: Fibonacci sequence number.
2. **Caused by Solution Requirement:**
   A problem may be so complicated that any solution to the problem will require exponential number of computational steps to obtain, no matter what kind of algorithms (recursive or iterative) we use.
   Example: Towers of Hanoi Problem
   $2^n - 1$ is the min # moves to move n disks.

Divide-and-Conquer and Recursive Algorithm:

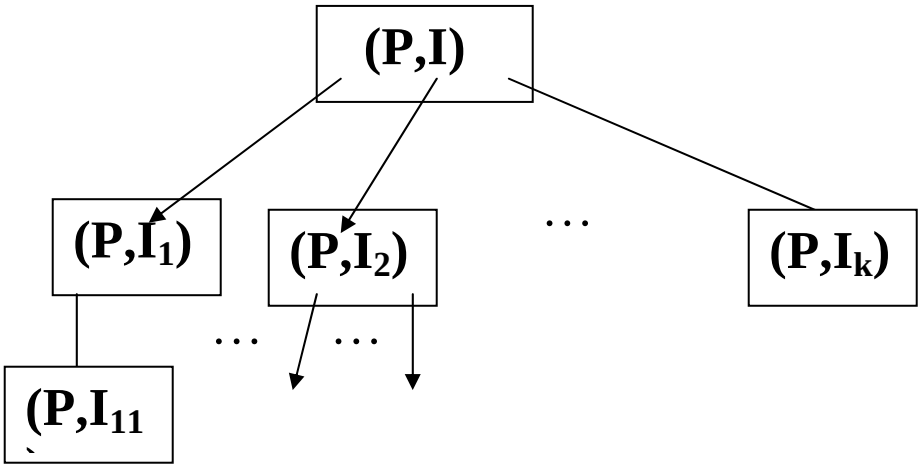
Given a general problem P with input I.

```
if the problem (P,I) can be solved directly
    then  solve (P,I)
    else  divide P into k subproblems of the same type;
          solve each subproblem recursively;
          combine solutions of subproblems to obtain the
                solution to the original problem;
endif;
```

General Format of DAC Algorithm:

```
Algorithm: DAC(P,I)
    if |I| is small enough to be solved
        then solve (P,I) directly    // from base case
        else divide (P,I) into (P,I1), (P,I2), …, (P,Ik));
             combine(DAC(P,I1), …, DAC(P,Ik))
    endif;
```

DAC Algorithm:



Examples:

1. Computing SumSquares of integers.

Input: Given two positive integers m and n with $m < n$.

Output: Compute $m^2 + (m+1)^2 + (m+2)^2 + \ldots + (n-1)^2 + n^2$

General approach:

$(m+1)^2 + \ldots + t^2 \mid (t+1)^2 + \ldots + (n-1)^2 + n^2$, $m \le t < n$.

Define a function sumSquares(m,n) as follows:
sumSquares(m,m) = m^2,
sumSquares(m,n) = sumSquares(m,t) + sumSquares(t+1,n), $m < n$.

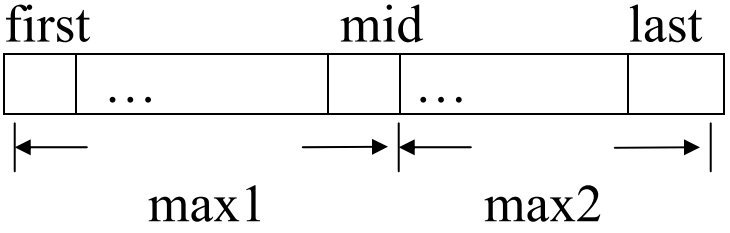
Algorithm 1: $m^2 \mid (m+1)^2 + \ldots + (n-1)^2 + n^2$
SumSquares1(m,m)= m^2,
SumSquares1(m,n) = m^2 + sumSquares1(m+1,n), for $m < n$.

Algorithm 2: $m^2 + (m+1)^2 + \ldots + (n-1)^2 \mid n^2$
SumSquares2(m,m)= m^2,
SumSquares2(m,n) = sumSquares2(m,n-1) + n^2, for $m < n$.

Algorithm 3: $m^2 + \ldots + mid^2 \mid (mid+1)^2 + \ldots + (n-1)^2 + n^2$
SumSquares3(m,m)= m^2,
SumSquares3=sumSquares3(m,mid)+sumSquares3(mid+1,n), for $m < n$.

```
int sumSquares1(int m, int n)
{
   if (m == n)            // base case
        return m*m;
   else
        return m*m + sumSquares1(m+1,n);
}  //end sumSquares1

int sumSquares2(int m, int n)
{
   if (m == n)            // base case
        return m*m;
   else
        return sumSquares2(m,n-1) + n*n;
}  //end sumSquares2


int sumSquares3(int m, int n)
{  int mid;

   if (m == n)            // base case
        return m*m;
   else
   {
        mid = (m+n)/2;    // compute midpoint for dividing
        return sumSquares3(m,mid)+sumSquares3(mid+1,n);
   }
}  //end sumSquares3
```

2. Computing the maximum integer in an array.



maxArray = max{max1,max2}

```
int maxInteger(const int anArray[], int first, int last)
{
   int mid;
   int max1, max2;

   if (first == last)            // base case
       return anArray[first];
   else
   {
       mid = (first + last)/2;
       max1 = maxInteger(anArray, first, mid);
       max2 = maxInteger(anArray, mid+1, last);

       if (max1 > max2)
          return max1;
       else return max2;
   }
} // end maxInteger
```

HW. Compute the maximum and minimum of anArray.