Week 5: Recursion

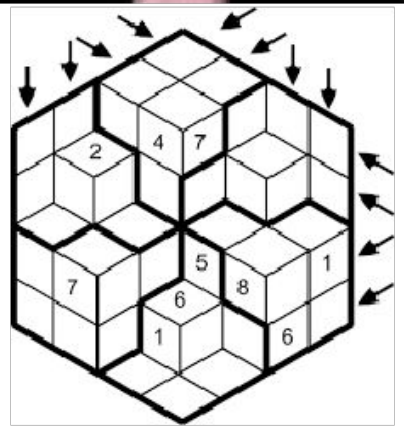
CS-250 Data Structure and Algorithms

DR. Mehwish Fatima | Assist. Professor
Department of AI & DS | SEECS, NUST

Mathematical Fractals & Recursion

- Natural objects exhibit scaling symmetry, a key characteristic associated with **fractals**.
  - They also tend to be “roughly” self-similar, appearing more or less the same at different scales of measurement.
  - Fractals are unpredictable in specific details
    - yet deterministic when viewed as a total pattern
    - in many ways this reflects what we observe in the small details & total pattern of life.

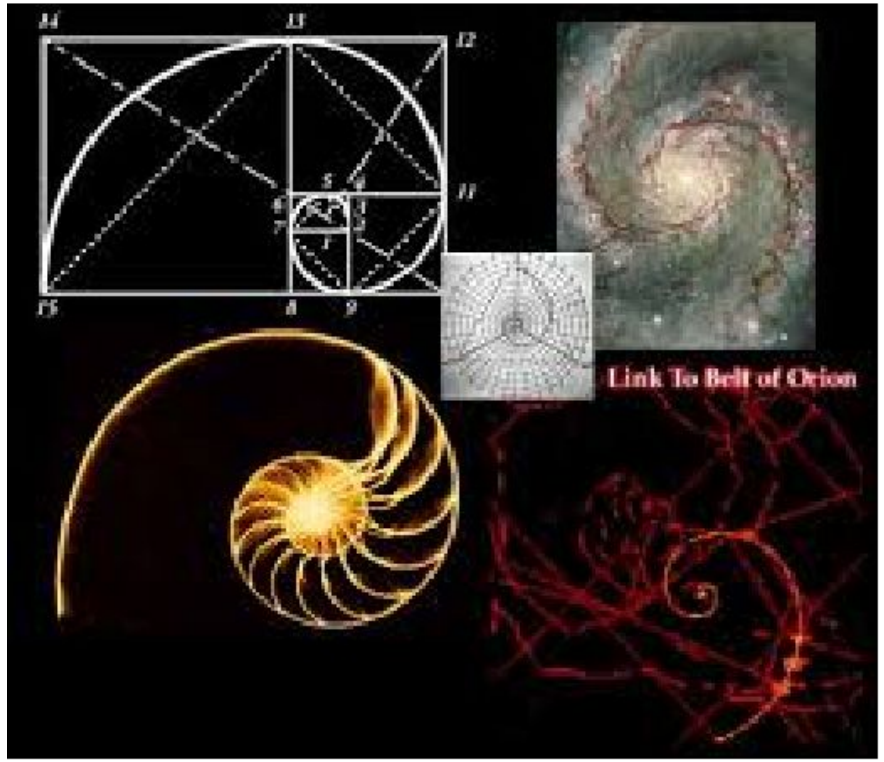
Recursion In Art

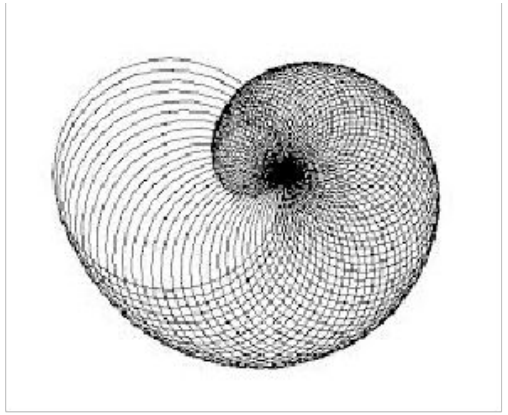


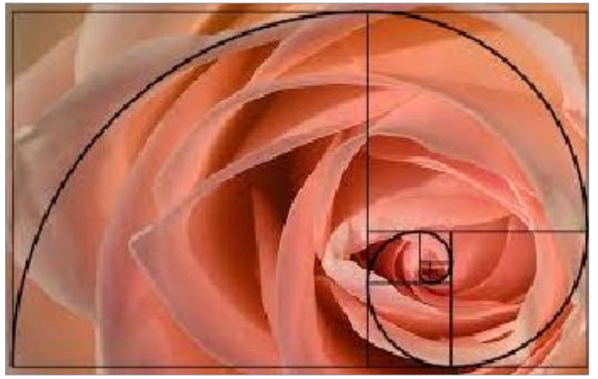
Recursion In Nature

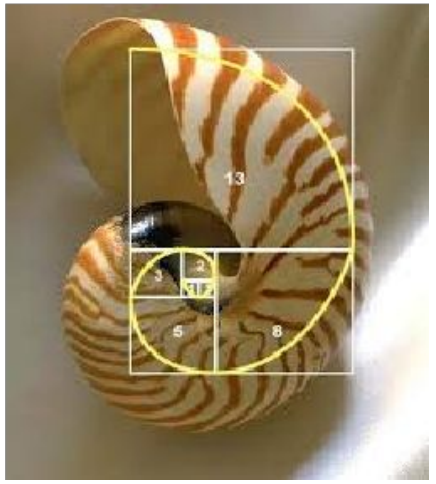
Recursion In Nature

Recursion in Math and Nature

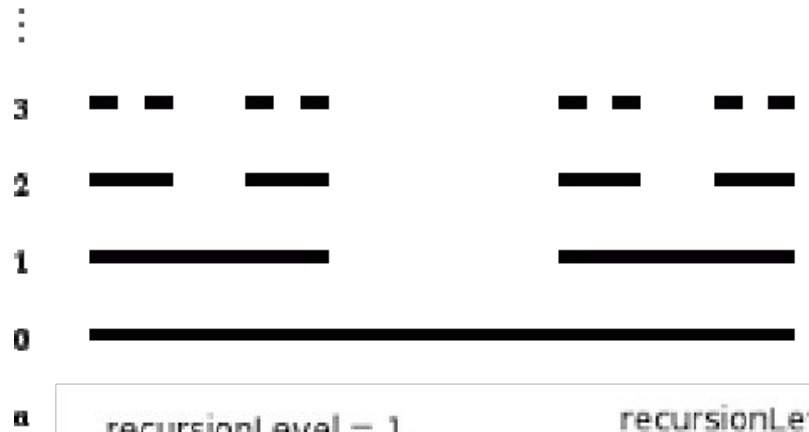


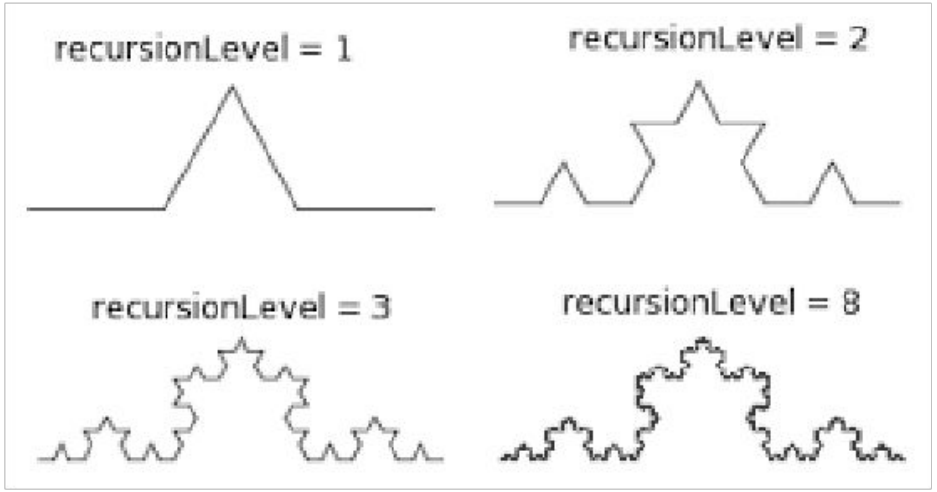


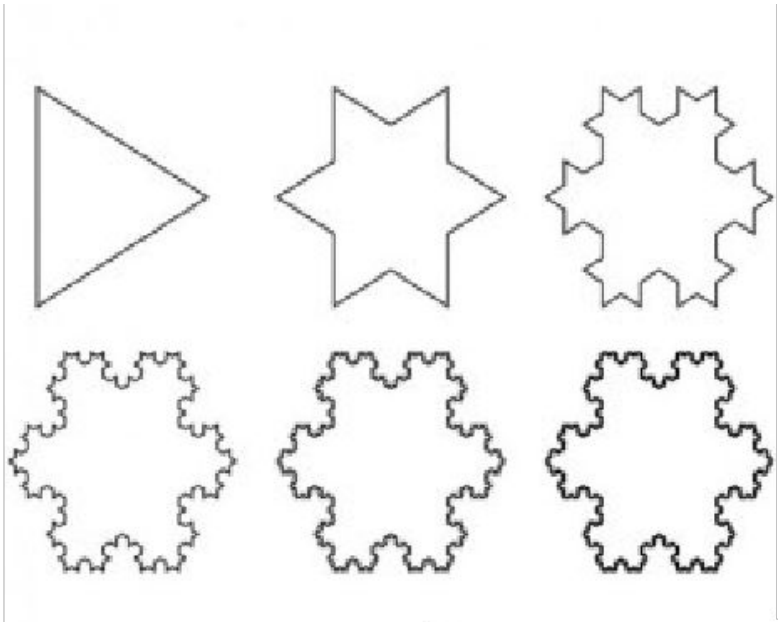


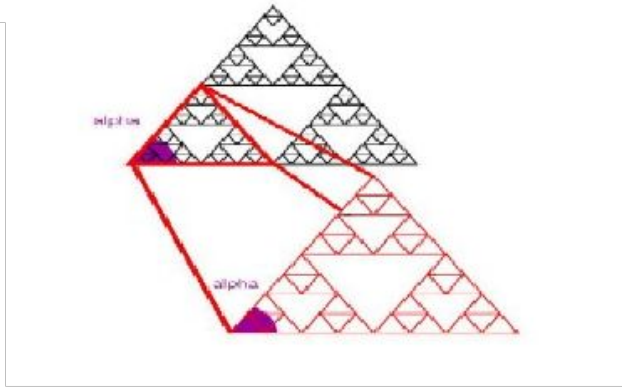


Recursion in Math and Signals









RECURSION

- Functions can be written in two ways:
  - Iterative
    - keep repeating until a task is “done”
      - *e.g.,* loop counter reaches limit
    - iterative algorithms are usually more efficient in their use of space and time
  - Recursive
    - Solve a large problem by breaking it up into smaller and smaller pieces until you can solve it; combine the results.
      - *e.g.,* recursive factorial function
    - Recursive algorithms are often shorter, more elegant, and easier to understand

RECURSION

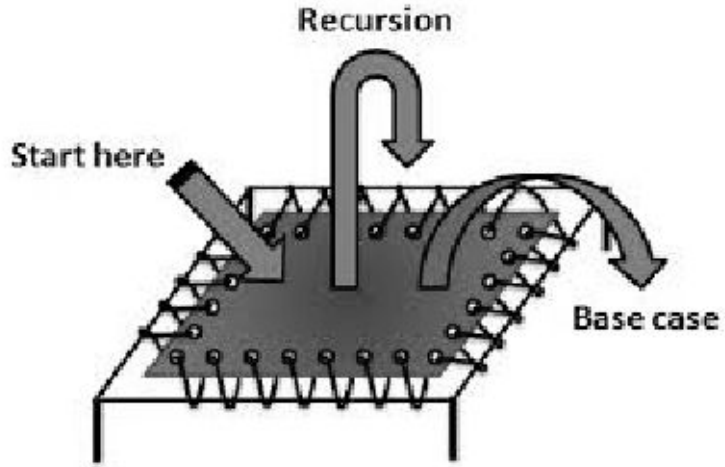
- Functions can be written in two ways:
  - Iterative
    - keep repeating until a task is “done”
      - *e.g.,* loop counter reaches limit
    - iterative algorithms are usually more efficient in their use of space and time
  - Recursive
    - Solve a large problem by breaking it up into smaller and smaller pieces until you can solve it; combine the results.
      - *e.g.,* recursive factorial function
    - Recursive algorithms are often shorter, more elegant, and easier to understand

For every recursive algorithm, there is an equivalent iterative algorithm.

RECURSIVE DEFINITIONS

- Recursion means having the characteristic of coming up again and again.
  - In a recursive definition, **an object is defined in terms of itself**.

- We can recursively defined sequences, functions and sets.
  - Example: The sequence $\{a_n\}$ of powers of 2 is given by
    $a_n = 2^n$ for n = 0, 1, 2, … .
  - The same sequence can also be defined recursively:
    - $a_0 = 1$
    - $a_{n+1} = 2a^n$ for n = 0, 1, 2, …



RECURSION

- A function that invokes itself by making a call to itself is called a recursive function.
  - Direct Recursion:
    - When a function calls itself
  - Indirect Recursion:
    - When A calls B and B calls A
- **Recursive definition**
  - An object is defined in terms of smaller version of itself.
  - A problem is decomposed into simpler sub-problems of the same kind.

PROPERTIES OF RECURSIVE FUNCTION

- A recursive function must have two properties:
  - There must be a certain (base) criteria for which function doesn't call itself.
  - Each time function does call itself (directly or indirectly), it must closer to the base criteria – Reduction Step.
- A recursive function with these two properties is said to be well-defined.

RECURSIVE FUNCTION

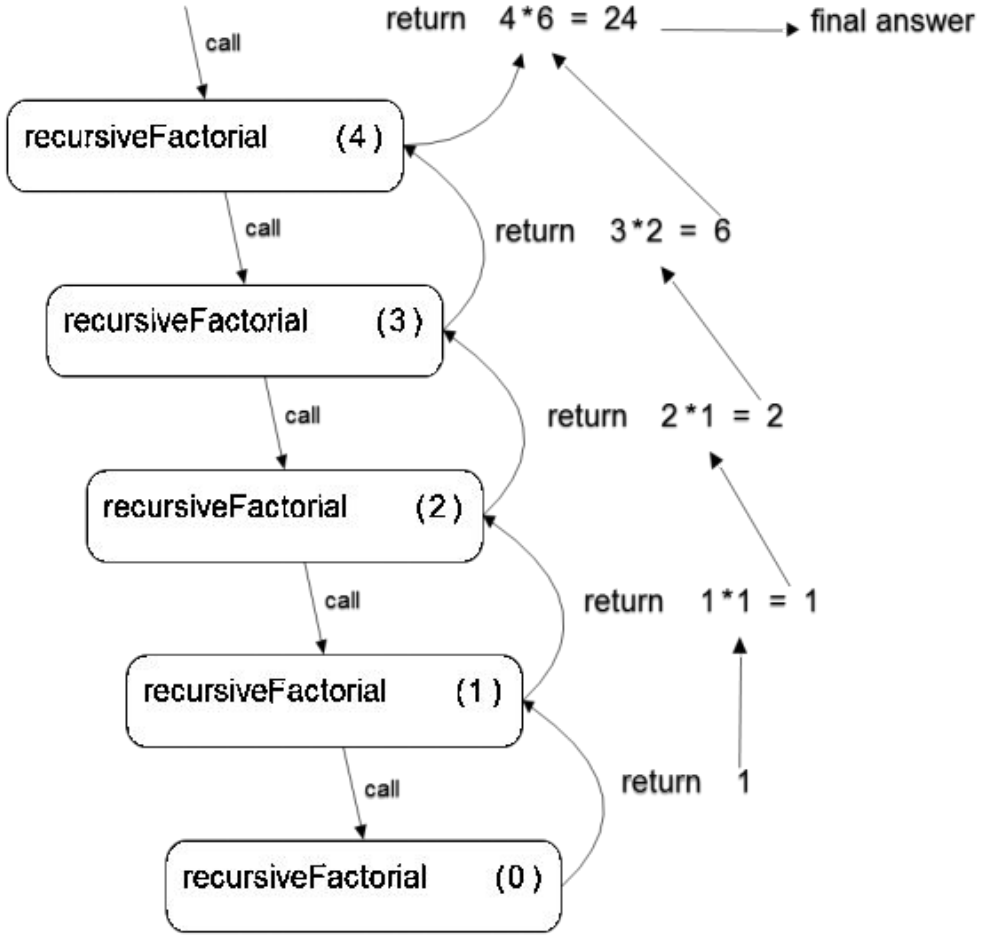
- Recursion is the process in which repetition is in a “self-similar” manner and has two parts
  - Base Case / Ground Case
    - Stops the recursion
    - There should be some base values.
  - Rules / General Case
    - Recursion or call to itself
    - In other words, with a recursive step, you apply the same algorithm on a scaled down problem and its process is repeated until the **end of condition** is reached.

FACTORIAL

- Base Case : **if n is 0 then n! = 1**

- Recursive Call: **if n > 0 then n! = n.(n-1)!**

```
int recursiveFactorial(int n) {                // recursive factorial function
  if (n == 0) return 1;                        // basis case
  else return n * recursiveFactorial(n–1);     // recursive case
}
```



Factorial

```
void main()
{
    int answer = factorial(3);
    cout << answer;
}
```

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

factorial(0) fact= 1

factorial(1) fact= 1* factorial(0)

factorial(2) fact= 2* factorial(1)

factorial(3) fact= 3* factorial(2)

Main() answer=factorial(3)

FIBONACCI SEQUENCE

- Following is the Fibonacci Series
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89……..
- Fibonacci series start with 0 and 1 as first two numbers and any **subsequent number is sum of previous two numbers**
- 2 = 1+1
- 3 = 2+1
- 5 = 3+2
- 8 = 5+3
- 21 = 13+8

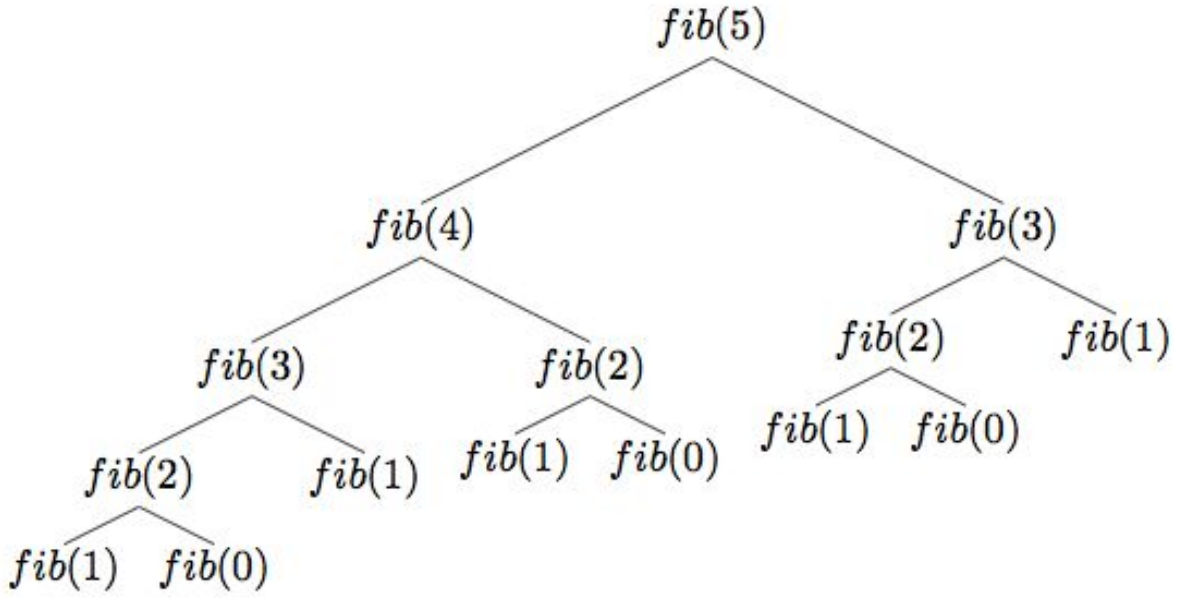
$$
\begin{aligned}
n_0 &= 0 \\
n_1 &= 1 \\
n_2 &= n_1 + n_0 = 1 + 0 = 1 \\
n_3 &= n_2 + n_1 = 1 + 1 = 2 \\
n_4 &= n_3 + n_2 = 2 + 1 = 3 \\
n_5 &= n_4 + n_3 = 3 + 2 = 5 \\
n_6 &= n_5 + n_4 = 5 + 3 = 8 \\
n_7 &= n_6 + n_5 = 8 + 5 = 13 \\
n_8 &= n_7 + n_6 = 13 + 8 = 21.
\end{aligned}
$$

NTH FIBONACCI NUMBER

- Base Case : **if n = 1 or n=0 then $F_n = n$**

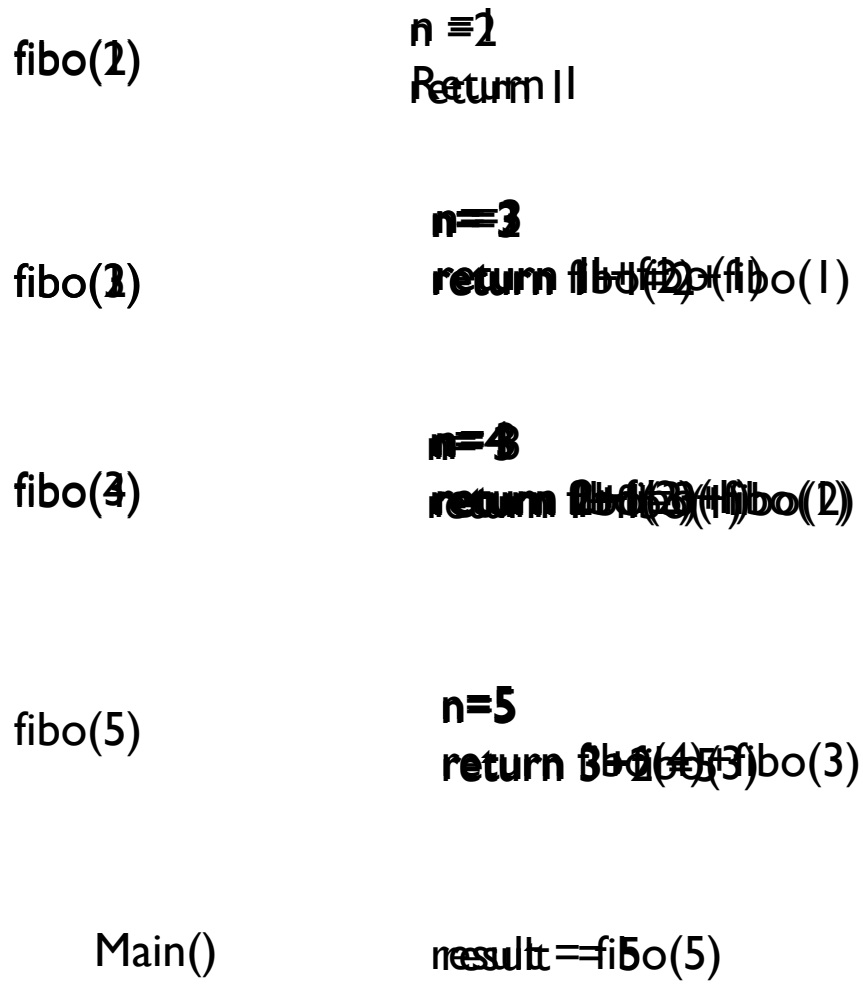
- Recursive Call: **if n > 1 then $F_n = F_{n-2} + F_{n-1}$**

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



```
void main(){
      int result= fibo(5);
}
```

```
int fibo(int n)
{
      if  ( n==1 or n == 2)
        return 1;
      else
        return fibo(n-1)+fibo(n-2);
}
```



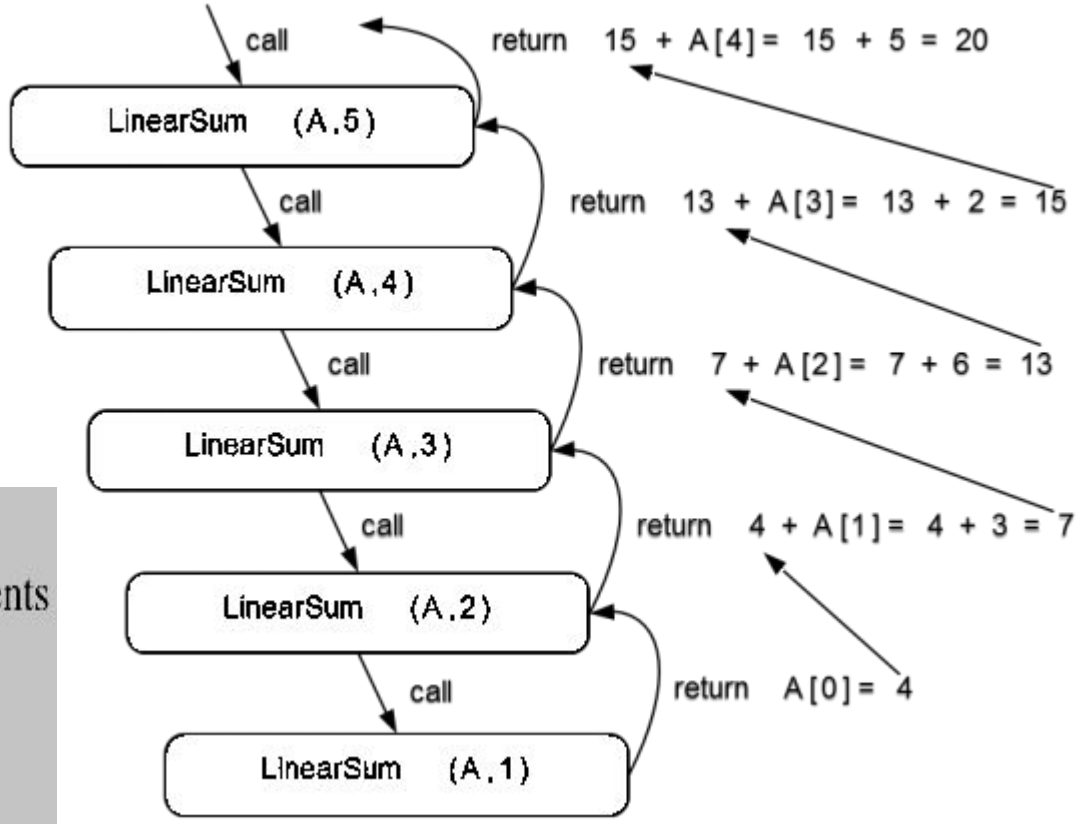
KEY POINTS

- If you are not careful with the program logic, you may miss a base case and go off into an infinite recursion.
  - This is similar to an infinite loop!
  - **Example:** call to factorial with N < 0
  - Either you must ensure that factorial is never, ever called with a negative N, or you must build in a check somehow.

- Moral
  - When you are designing your recursive calls, make sure that at least one of the base cases MUST be reached eventually.

TYPES OF RECURSION

- Linear Recursion
  - Function makes one recursive call each time it is invoked
  - Example: Summing Array elements recursively



Algorithm LinearSum(A,n):

Input: A integer array A and an integer $n \geq 1$, such that A has at least n elements

Output: The sum of the first n integers in A

```
if n = 1 then
    return A[0]
else
    return LinearSum(A, n − 1) + A[n − 1]
```

TYPES OF RECURSION

- Tail Recursion
  - Tail recursion occurs when a linearly recursive method makes its recursive call as its last step.
  - Such methods can be easily converted to non-recursive methods (which saves on some resources).
- Algorithm ReverseArray(A, i, j): An array A and nonnegative integer indices i and j
  - Output: The reversal of the elements in A starting at index i and ending at j
  - if i < j then
    - Swap A[i] and A[j]
    - ReverseArray(A, i + 1, j - 1)
  - return

TYPES OF RECURSION

- Binary Recursion
  - Function makes two recursive calls each time it is invoked
  - Example summing the array using binary sum

Here i is the start index of array, and n is the size of array. **Remember n is not last index**

Algorithm BinarySum(A, i, n):

Input: An array A and integers i and n

Output: The sum of the n integers in A starting at index i

```
if n = 1 then
    return A[i]
return BinarySum(A, i, ⌈n/2⌉) + BinarySum(A, i + ⌈n/2⌉, ⌊n/2⌋)
```

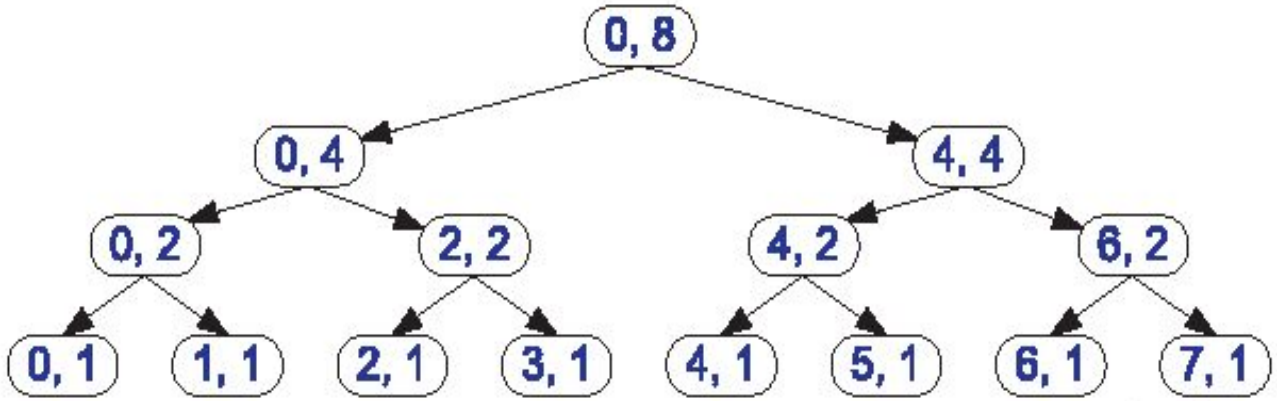


Figure 3.20: Recursion trace for the execution of BinarySum$(0, 8)$.

BINARY SEARCH

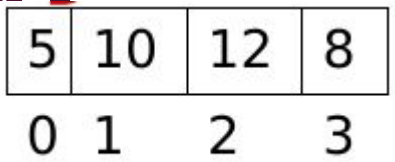
Algorithm: **RECURSIVE_BINARY_SEARCH**(A[], int key, int low, int high)

```
1.    if (low>high)
2.    return -1 //KEY_NOT_FOUND”
3.     else
4.        mid = (low+high)/2;
5.        if (A[mid] > key)
6.            return RECURSIVE_BINARY_SEARCH (A, key, low, mid-1)
7.        else if (A[mid] < key)
8.            return RECURSIVE_BINARY_SEARCH (A, key, mid+1, high)
9.        else
10.           return mid;
11.       End If
12.   End If
```

EXERCISE

- Find the maximum number of array
  - Iterative
  - Recursive
- Find the minimum number of array
  - Iterative
  - Recursive

Max of Array




```
void main(){
      int list[4]={5,10,12,8};
      int large = Largest(list,0,3);
}
```

```
int Largest(const int list[], int low, int
      high){
      int max;
      if(low==high)
        return list[low];
      else
      {
        max = Largest(list,low+1,high);
        if(list[low]>=max)
              return list[low];
        else
              return max;
      }
}
```

Largest(list,3,3)

Low=3, high = 3
max = list[3] → return 8

Largest(list,2,3)

Low=2, high = 3
max = Largest(list,3,3)
Low=2, high =3
max =8 → list[2]>max return
12

Largest(list,1,3)

Low=1, high = 3
max = Largest(list,2,3)
Low=1, high =3
max =12 → list[1]<max return
12

Largest(list,0,3)

Low=0, high = 3
max = Largest(list,1,3)
Low=0, high =3
max = 12 → list[0] <max return
12

Main()

Large = 12

Large = Largest(list,0,3)

RECURSION

- Very useful technique
  - Definition of mathematical functions
  - Definition of data structures
    - Recursive structures are naturally processed by recursive functions!

RECURSION

- Recursively defined functions
  - Factorial
  - Fibonacci
  - GCD by Euclid's algorithm
  - Fourier Transform
  - Games
  - Towers of Hanoi
  - Chess
  - Pathfinding

http://chessprogramming.wikispaces.com/Recursion

TOWER OF HANOI & N-QUEENS

- Tower of Hanoi
  - A logical puzzle game that can be solved very effectively with recursion.
  - Three pegs. On each of these pegs is a series of disks decreasing in size from the bottom of the peg to the top of the peg.
- The goal is to move all these disks to another peg following these rules:
  - Only one disk can be moved at a time.
  - Only the topmost disk of any given peg can be moved to another peg.
  - larger disk cannot be placed on top a smaller disk.
- The simplest way of doing this is via Divide and Conquer.

https://www.youtube.com/watch?v=Q3U6PRZDjTA

https://www.youtube.com/watch?v=0DeznFqrgAI

TOWER OF HANOI

- */* Params: diskSize refers to the maximum number of disks we wish to move.*
- *If we have 5 disks and we wish to move all five disks to another peg then this value should be 5.*
- *5 = largest disk and 0 = smallest disk.*
- *source is the peg where the disks currently exist.*
- *dest is the peg we want the disks to be moved to.*
- *spare is the peg that temporally stores pegs.*
- *This is necessary in order for us to shuffle disks around without violating the rules.*/*

TOWER OF HANOI

```
void hanoi(int diskSize, int source, int dest, int spare)
{
        //This is our standard termination case.
         We get to here when we are operating on the smallest possible disk.
       if(diskSize == 0)
       {
             print("Move disk " +diskSize +" from peg " +source +" to peg " +dest )
       }
        else    {
                    /*Move all disks smaller than this one over to the spare.  So if
diskSize is 5, we move 4 disks to the spare.  This leaves us with 1 disk on the
source peg.  Note the placement of the params.  We are now using the dest
peg as the spare peg.  This causes each recursion to ping-pong the spare and
dest pegs. */
```

TOWER OF HANOI

- hanoi(diskSize - 1, source, spare, dest);
  - *//Move the remaining disk to the destination peg.*
- print("Move disk " +diskSize +" from peg "+source +" to peg " +dest);
  - *//Move the disks we just moved to the spare back over to the dest peg.*
- hanoi(diskSize - 1, spare, dest, source); }}
- Public static void main (String[] args)
  - *//Move all 3 disks from peg 0 to peg 1 using peg 2 as a temporary.*
- hanoi(2, 0, 1, 2); return 0;}

https://www.includehelp.com/data-structure-tutorial/tower-of-hanoi-using-recursion.aspx#google_vignette

Recursion vs. Iteration

RECURSION	ITERATIONS
Recursive function – is a function that is partially defined by itself.	Iterative Instructions –are loop based repetitions of a process.
Recursion Uses selection structure.	Iteration uses repetition structure.
Infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on some condition.(base case)	An infinite loop occurs with iteration if the loop-condition test never becomes false.
Recursion terminates when a base case is recognized.	Iteration terminates when the loop-condition fails.
Recursion is usually slower then iteration due to overhead of maintaining stack.	Iteration does not use stack so it's faster than recursion.
Recursion uses more memory than iteration.	Iteration consume less memory.
Infinite recursion can crash the system.	infinite looping uses CPU cycles repeatedly.
Recursion makes code smaller.	Iteration makes code longer.

READING

- N-queens Problem
  - Example: The standard 8 by 8 Queen's problem asks how to place 8 queens on an ordinary chess board so that none of them can hit any other in one move

- http://www.geeksforgeeks.org/backtracking-set-3-n-queen-problem/

- http://en.wikibooks.org/wiki/Algorithm_Implementation/Miscellaneous/N-Queens