Vietnam National University of HCMC
International University
School of Computer Science and Engineering

Data Structures and Algorithms
★ Recursion ★

Dr Vi Chi Thanh - vcthanh@hcmiu.edu.vn
https://vichithanh.github.io

SCAN ME

Week by week topics (*)

1. Overview, DSA, OOP and Java
2. Arrays
3. Sorting
4. Queue, Stack
5. List
6. Recursion

Mid-Term

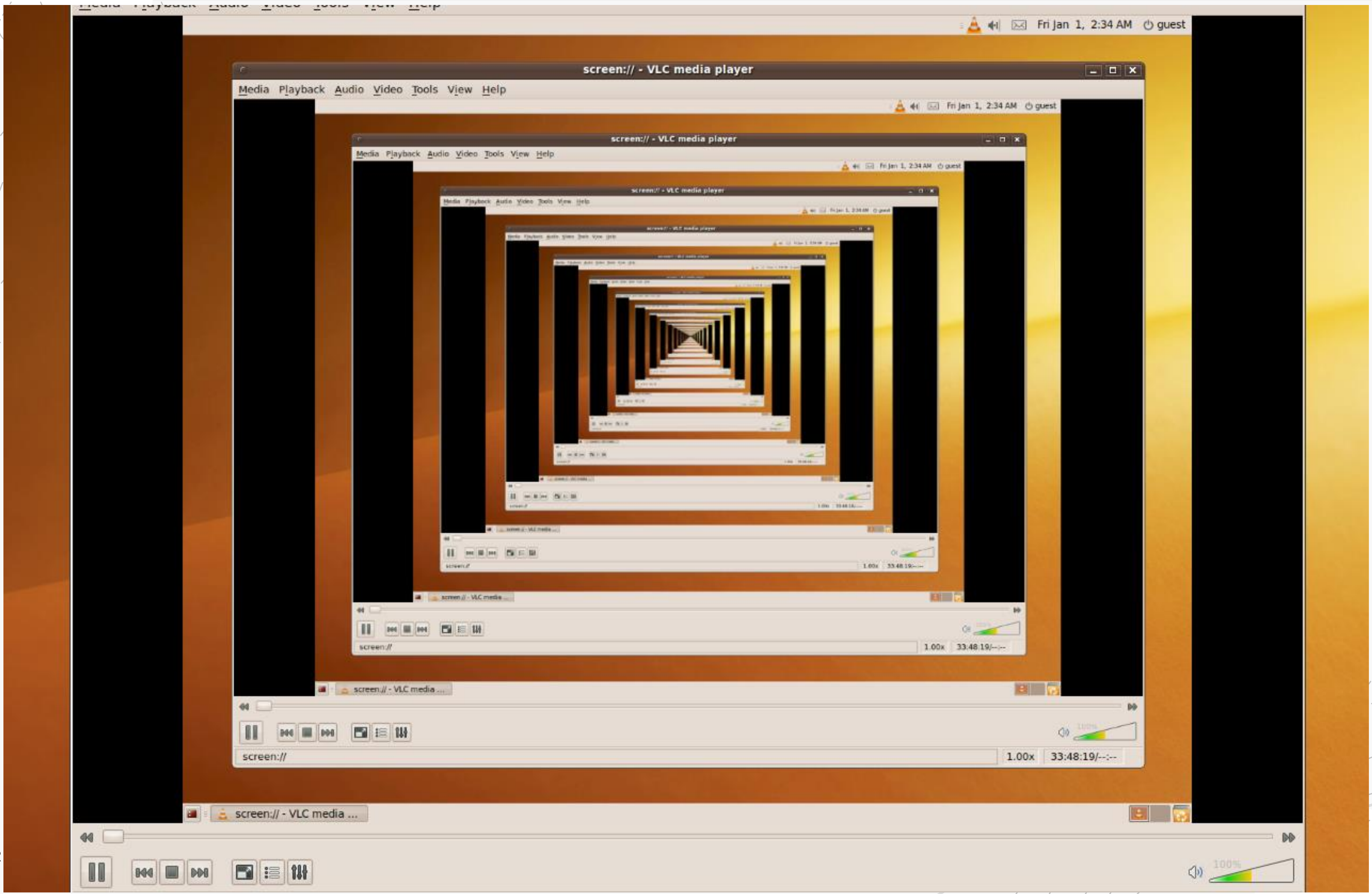
7. Advanced Sorting
8. Binary Tree
9. Hash Table
10. Graphs
11. Graphs Adv.

Final-Exam

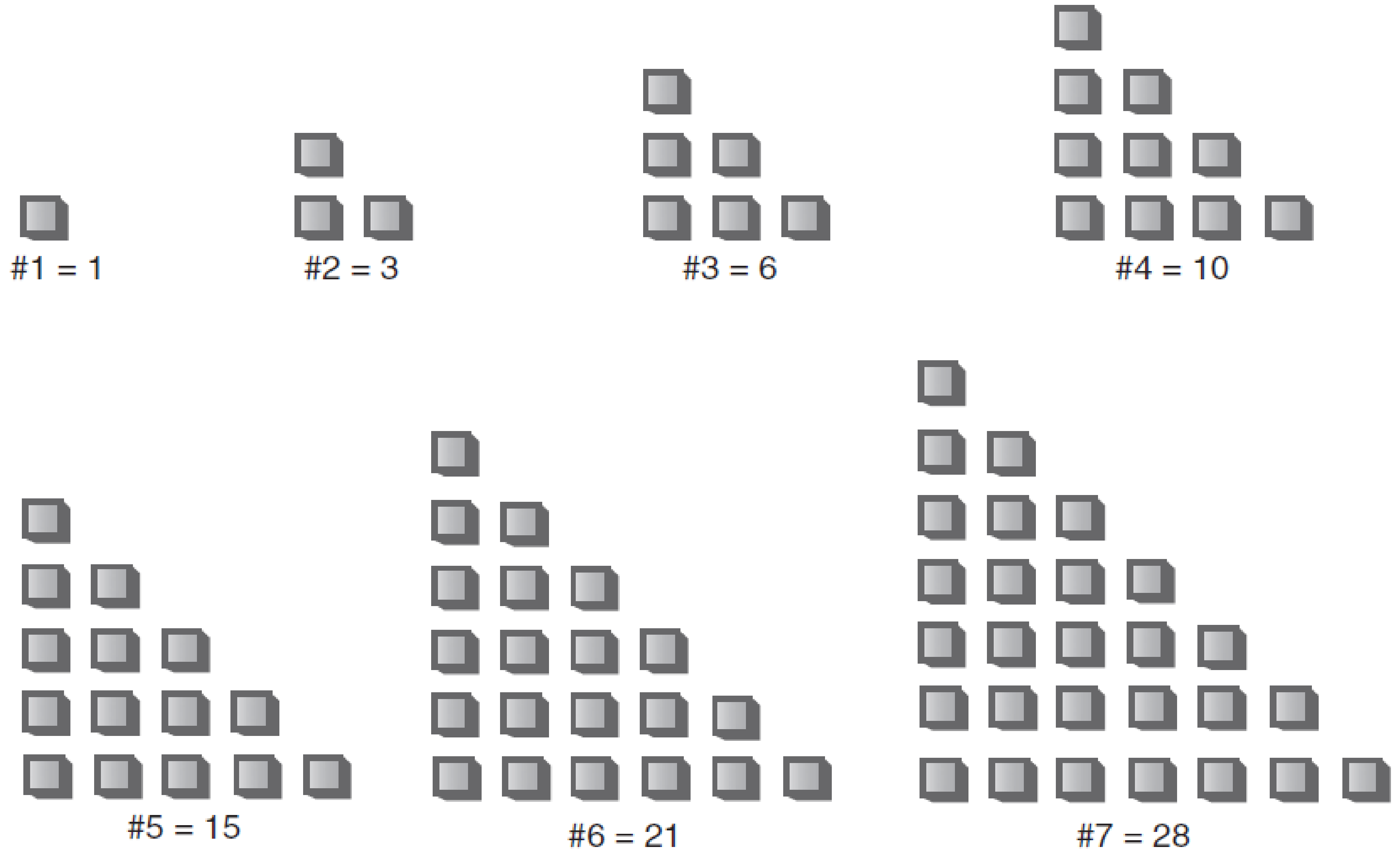
10 LABS

Objectives

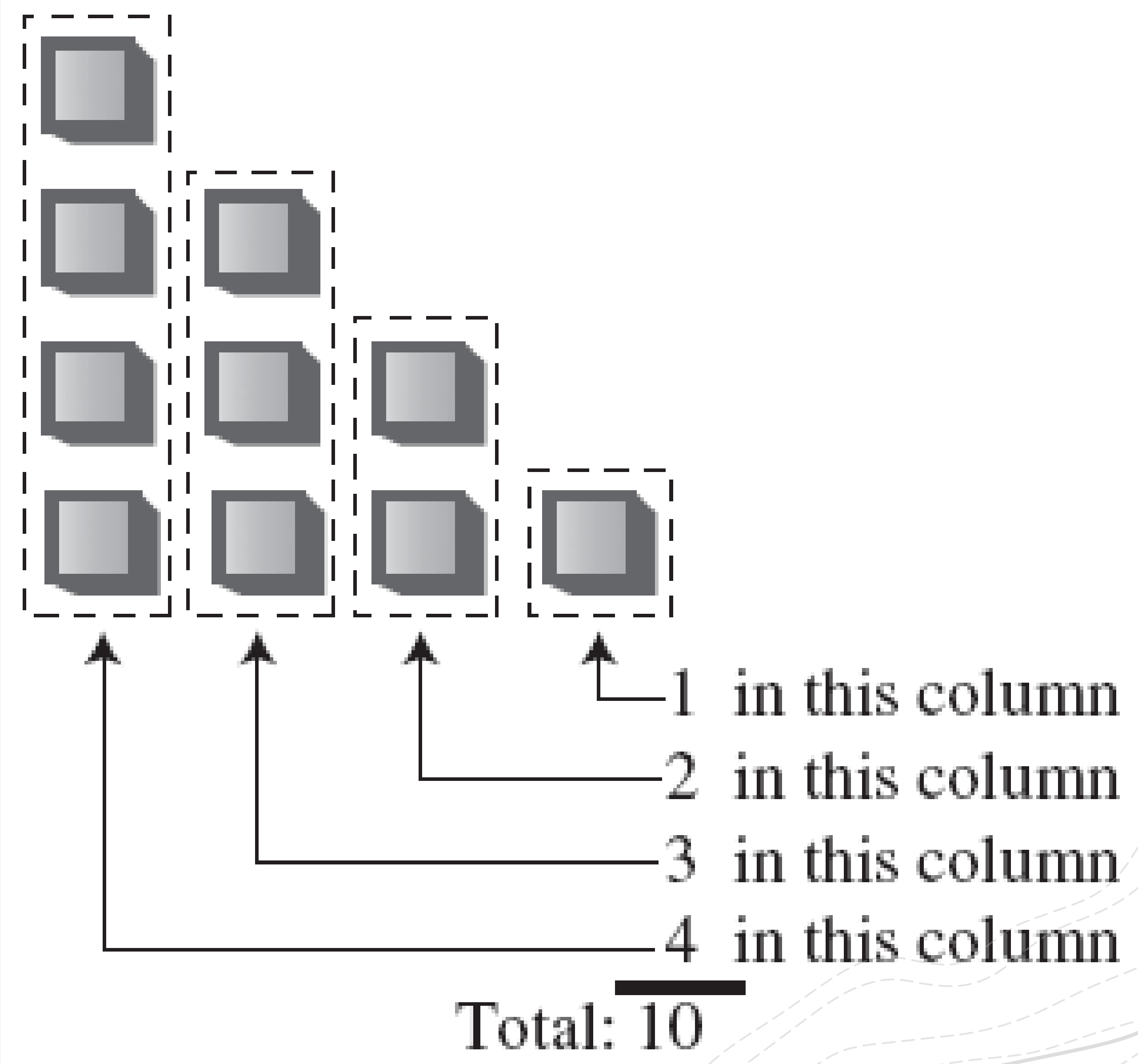
1. Introduction
2. Triangular Numbers
3. Recursive definition
4. Recursive characteristics
5. Factorials
6. Recursion implementation & Stacks
7. Recursive binary search
8. Tower of Hanoi
9. Merge sort
10. Tail recursion
11. Non-tail recursion
12. Indirect recursion
13. Nested recursion
14. Excessive recursion


screen:// - VLC media player
Media Playback Audio Video Tools View Help
Fri Jan 1, 2:34 AM
guest
screen:// - VLC media ...
screen://
1.00x
33:48:19/--:--

Triangular Numbers: Examples



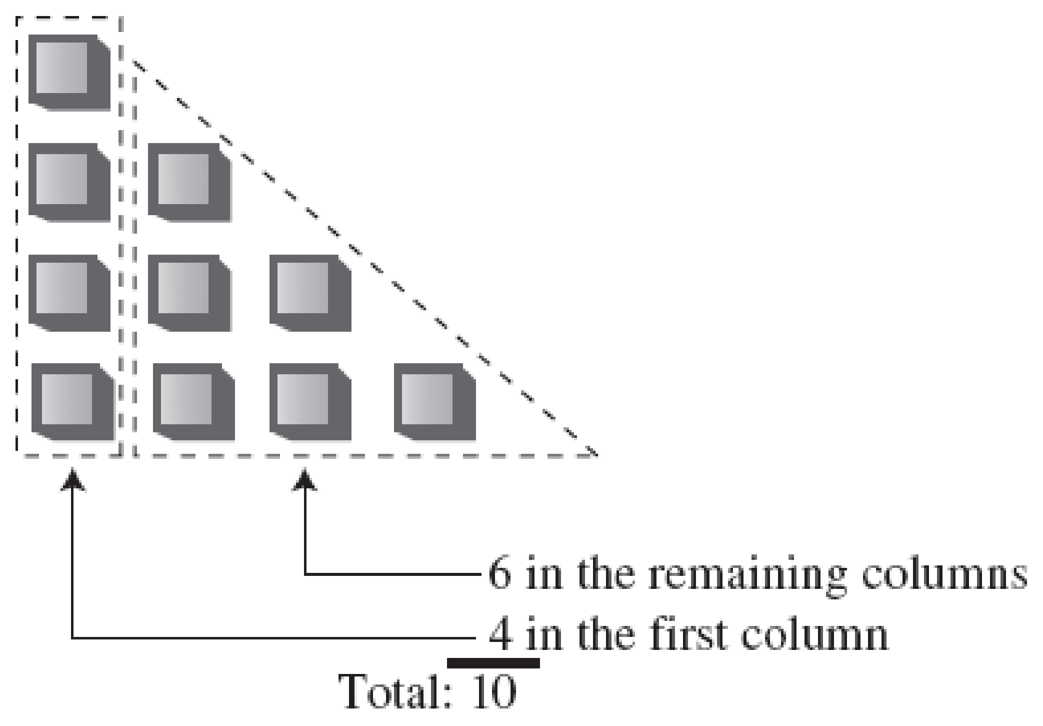
Finding nth Term using a Loop



```
int triangle(int n)
   {
   int total = 0;

   while(n > 0)           // until n is 1
      {
      total = total + n;  // add n (column height) to total
      --n;                // decrement column height
      }
   return total;
   }
```

Finding nth Term using Recursion



- Value of the nth term is the SUM of:
  - The first column (row): n
  - The SUM of the rest columns (rows)

Recursive method

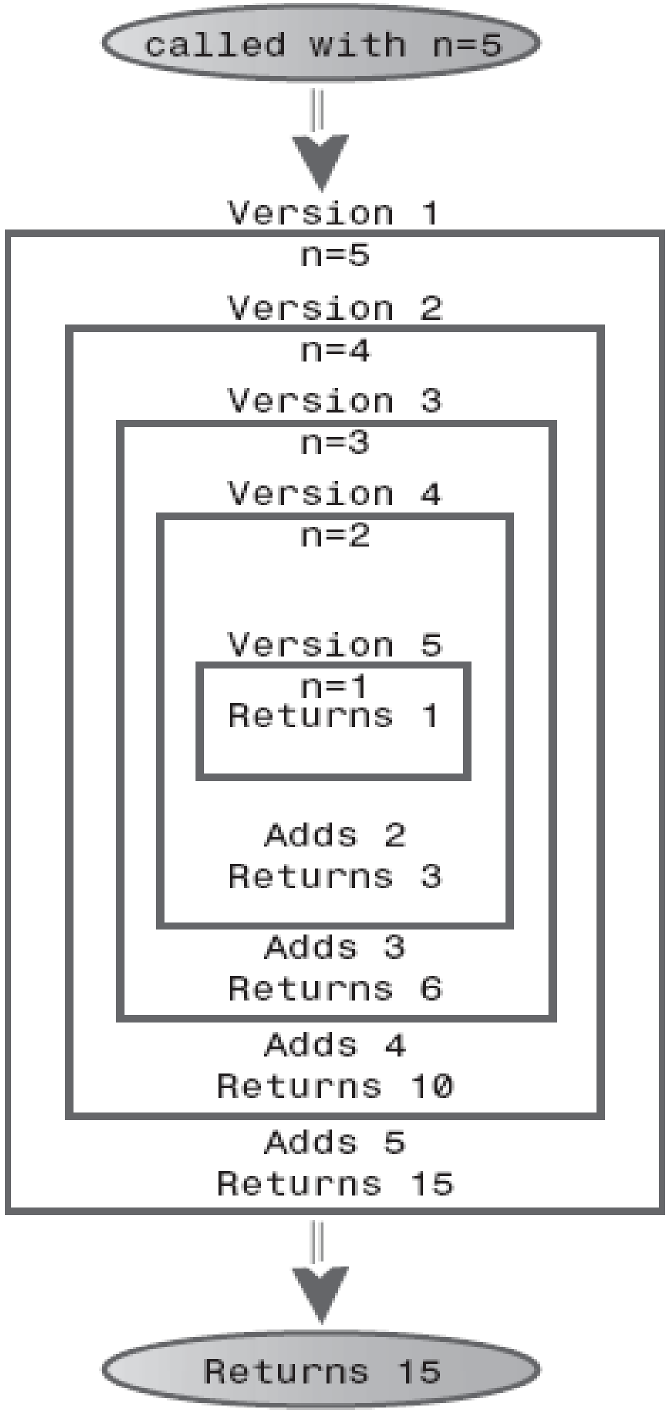
```
int triangle(int n)
   {
   return( n + triangle(n-1) );  // (incomplete version)
   }
```

Recursive method

- Now, it is complete with stopping condition

```
int triangle(int n)
   {
   if(n==1)
      return 1;
   else
      return( n + triangle(n-1) );
   }
```

- See ***triangle.java*** program



Definition

- When a function calls itself, this is known as recursion.
- Is a way to archive repetition, such as **while-loop** and **for-loop**
- This is an important theme in Computer Science that crops up time & time again.
- Can sometimes lead to very simple and elegant programs.

Characteristics of Recursive Program/ Algorithms

- There are three basic rules for developing recursive algorithms.
  - Know how to take one step.
  - Break each problem down into one step plus a smaller problem.
  - Know how and when to stop.
- Example for recursive program:
  - Factorial of a natural number

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

```
public class FactorialCalculator {

    public static int factorial(int n) {
        if (n < 0) {
            throw new IllegalArgumentException("Factorial is not
                      defined for negative numbers.");
        } else if (n == 0) {
            return 1;
        } else {
            return n * factorial(n - 1);
        }
    }
}
```

Recursion Characteristics

- There is some version of the problem that is simple enough that the routine can solve it, and return, without calling itself
- Is recursion efficient?
  - No
  - Address of calling methods must be remembered (in stack)
  - Intermediate arguments must also be stored

Example: Factorials

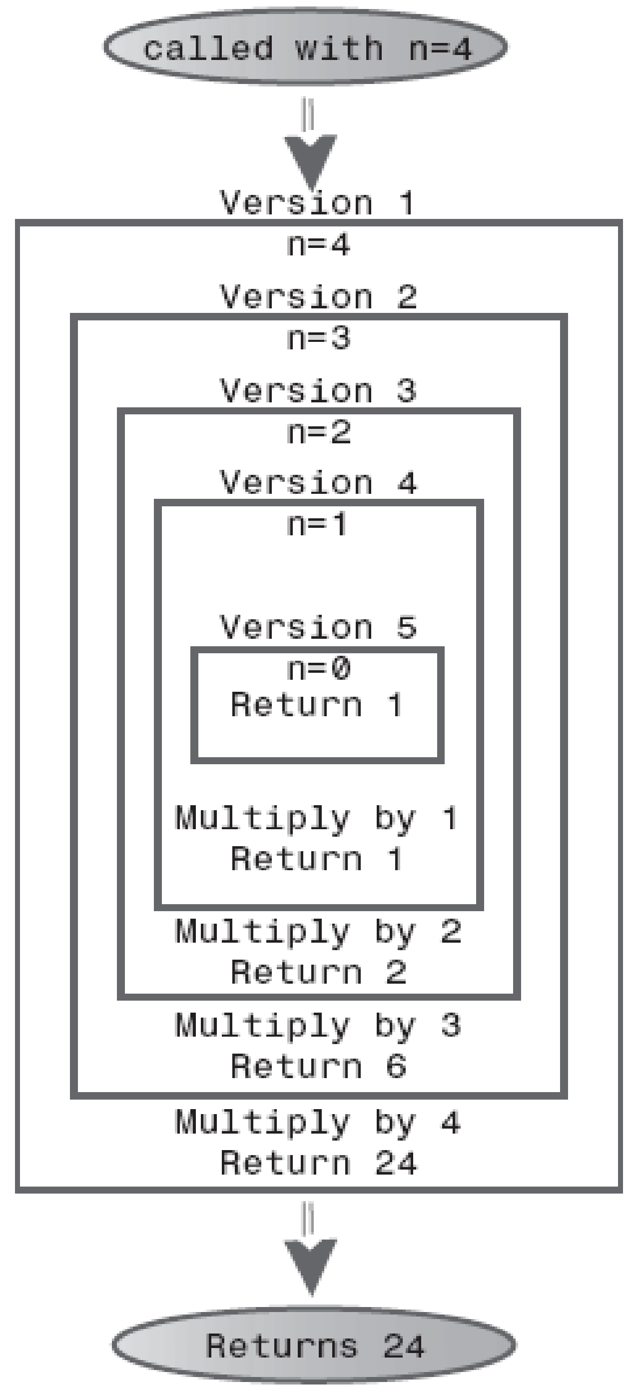
Number	Calculation	Factorial
0	by definition	1
1	1 * 1	1
2	2 * 1	2
3	3 * 2	6
4	4 * 6	24
5	5 * 24	120
6	6 * 120	720
7	7 * 720	5,040
8	8 * 5,040	40,320
9	9 * 40,320	362,880

Computing factorial by simple iteration

- See ***Factorial1.java***

Factorials

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

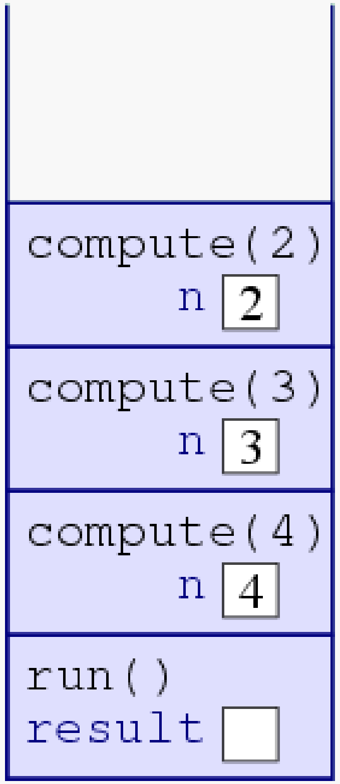


Computing factorial by recursion

- See ***Factorial2.java***
- Computing factorial by simulating recursion using a stack
  - ***Factorial3.java***

Recursion & stack

- Recursion is usually implemented by stacks



Method calls and recursion implementation

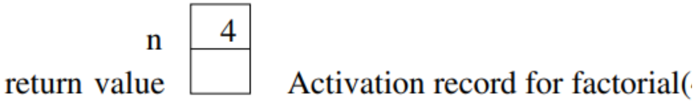
- Each time a method is called, an activation record (AR) is allocated for it in the memory. A recursive function that calls itself times N must allocate N activation records.
- This record usually contains the following information:
  - Parameters and local variables used in the called method.
  - A dynamic link, which is a pointer to the caller's activation record.
  - Return address to resume control by the caller, the address of the caller’s instruction immediately following the call.
  - Return value for a method not declared as void. Because the size of the activation record may vary from one call to another, the returned value is placed right above the activation record of the caller.

Method calls and recursion implementation

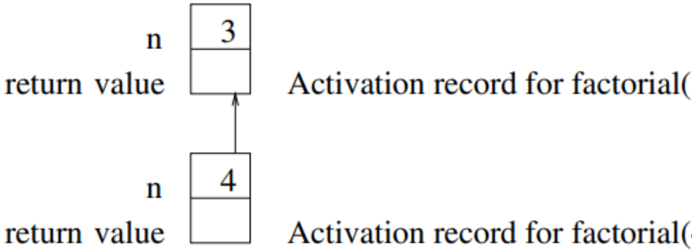
- Each new activation record is placed on the top of the run-time stack
- When a method terminates, its activation record is removed from the top of the run-time stack
- Thus, the first AR placed onto the stack is the last one removed.

Example:

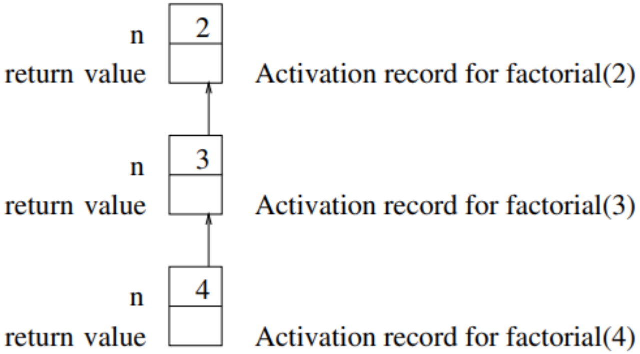
- **factorial(4)**



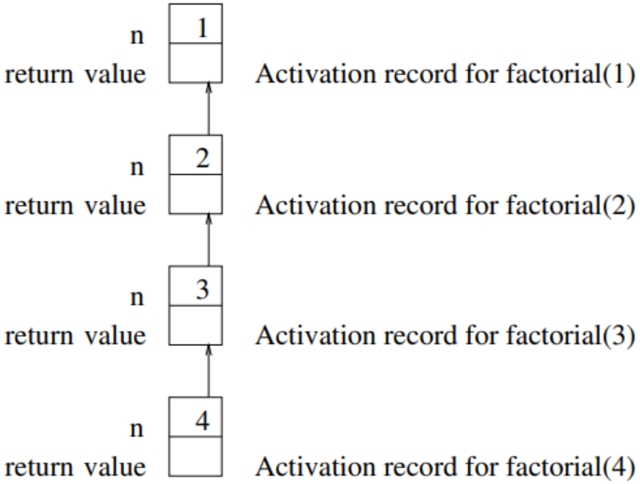
- **factorial(4)**: call to factorial(3)



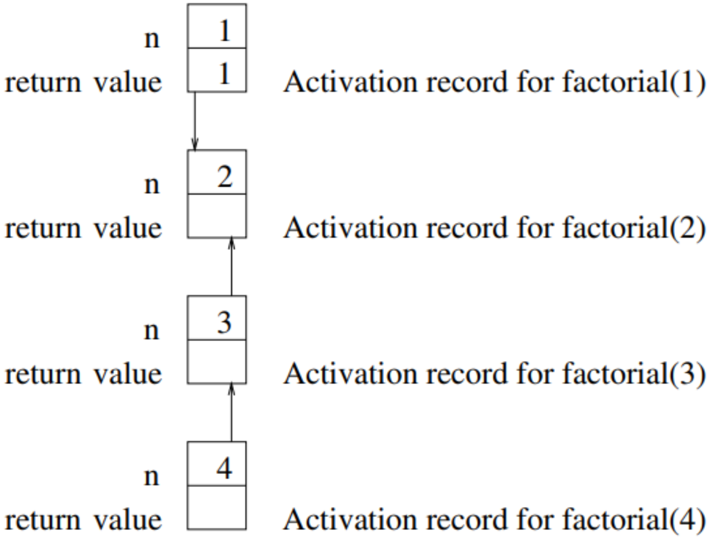
- **factorial(4):** call to factorial(2)



- **factorial(4):** call to factorial(1)



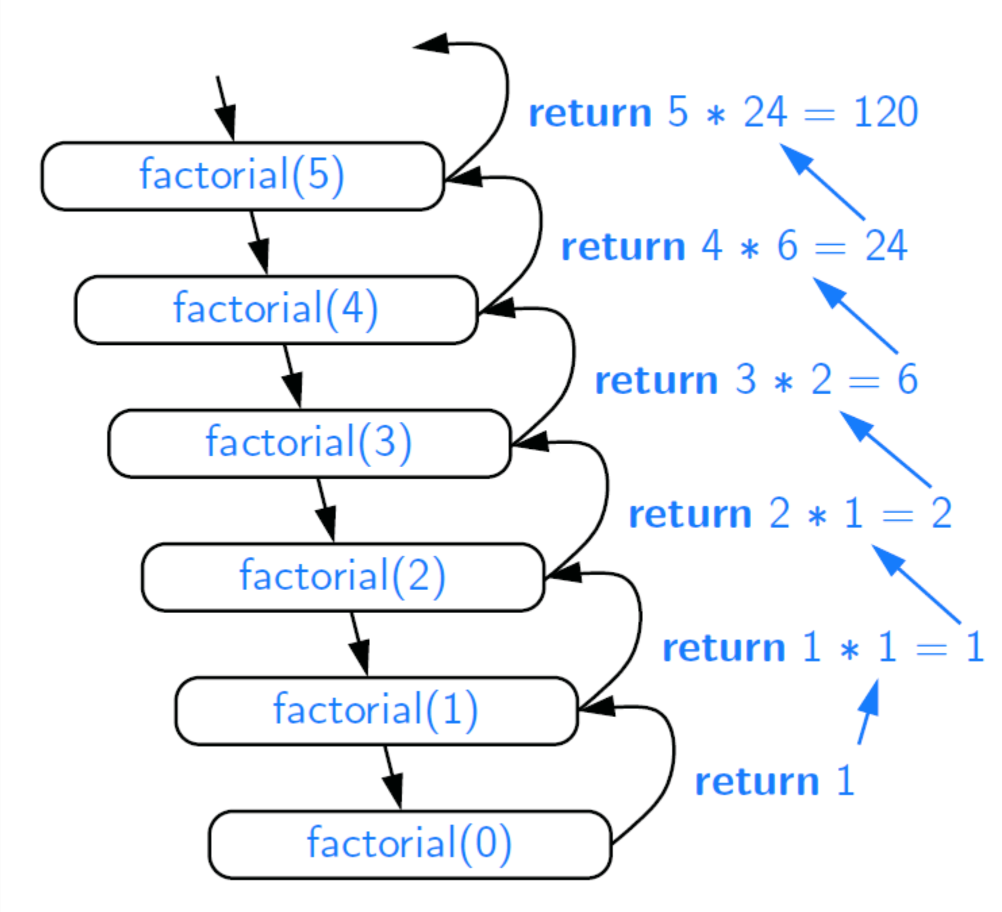
- **factorial(4):** factorial(1) is the base case, so it returns ‘1’.



Anatomy of a recursive call

- Example: Factorial of a number

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



Binary Search: Recursion vs. Loop

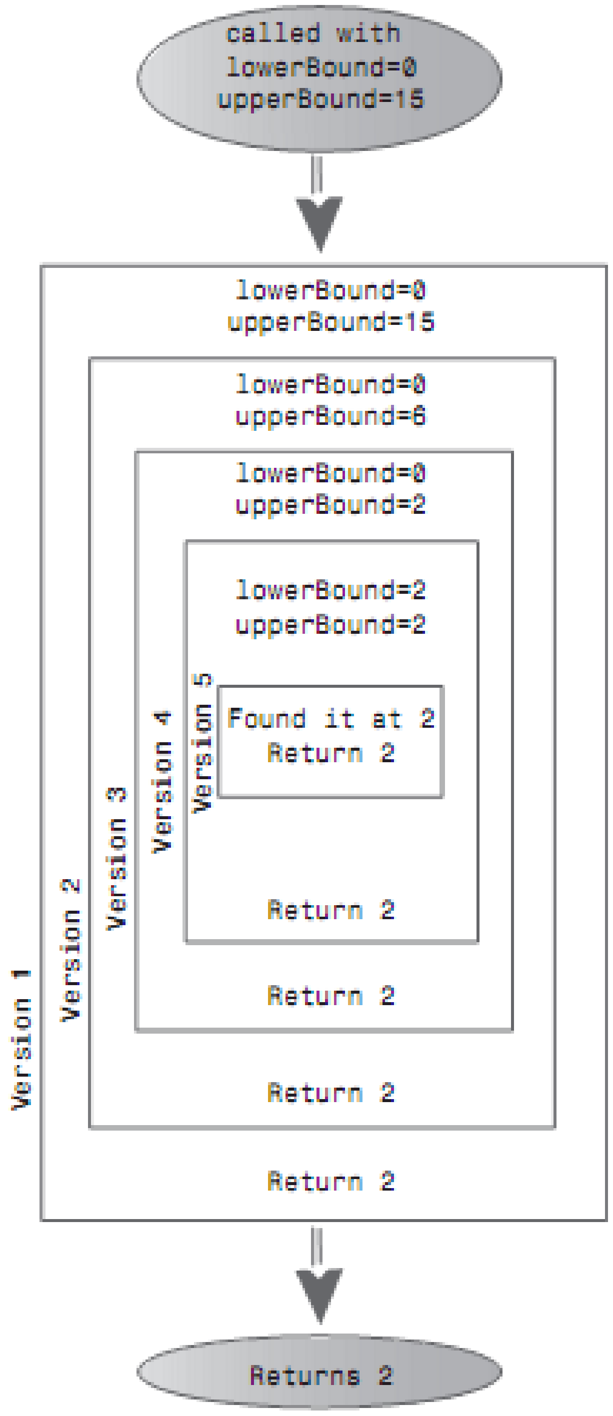
```
public int find(long searchKey)
   {
   int lowerBound = 0;
   int upperBound = nElems-1;
   int curIn;

   while(true)
      {
      curIn = (lowerBound + upperBound ) / 2;
      if(a[curIn]==searchKey)
         return curIn;              // found it
      else if(lowerBound > upperBound)
         return nElems;             // can't find it
      else                          // divide range
         {
         if(a[curIn] < searchKey)
            lowerBound = curIn + 1; // it's in upper half
         else
            upperBound = curIn - 1; // it's in lower half
```

Binary Search: Recursion vs. Loop

```
private int recFind(long searchKey, int lowerBound,
                                       int upperBound)
   {
   int curIn;

   curIn = (lowerBound + upperBound ) / 2;
   if(a[curIn]==searchKey)
      return curIn;              // found it
   else if(lowerBound > upperBound)
      return nElems;             // can't find it
   else                          // divide range
      {
      if(a[curIn] < searchKey)   // it's in upper half
         return recFind(searchKey, curIn+1, upperBound);
      else                       // it's in lower half
         return recFind(searchKey, lowerBound, curIn-1);
      }  // end else divide range
   }  // end recFind()
```



Recursive binary search implementation

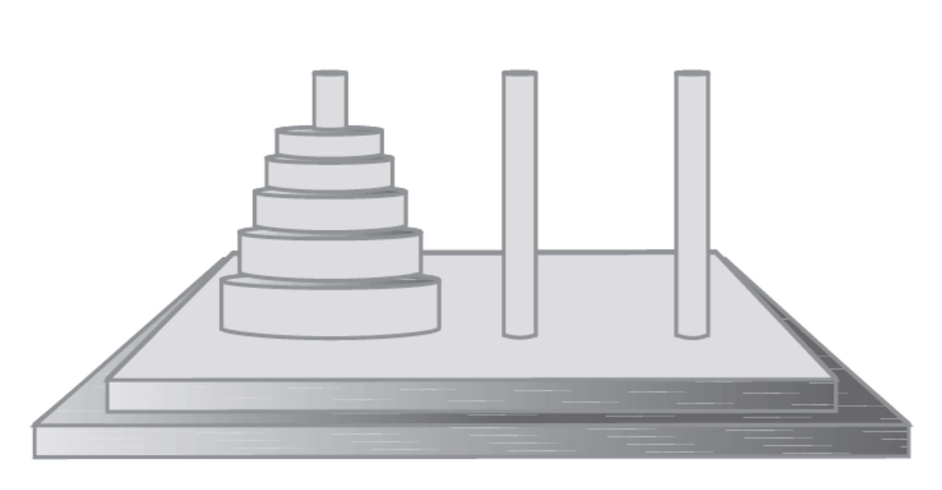
- ***BinarySearchApp.java***
- Trace the recursion by printing *lowerBound* and *upperBound* at each call and exit

Divide-and-conquer

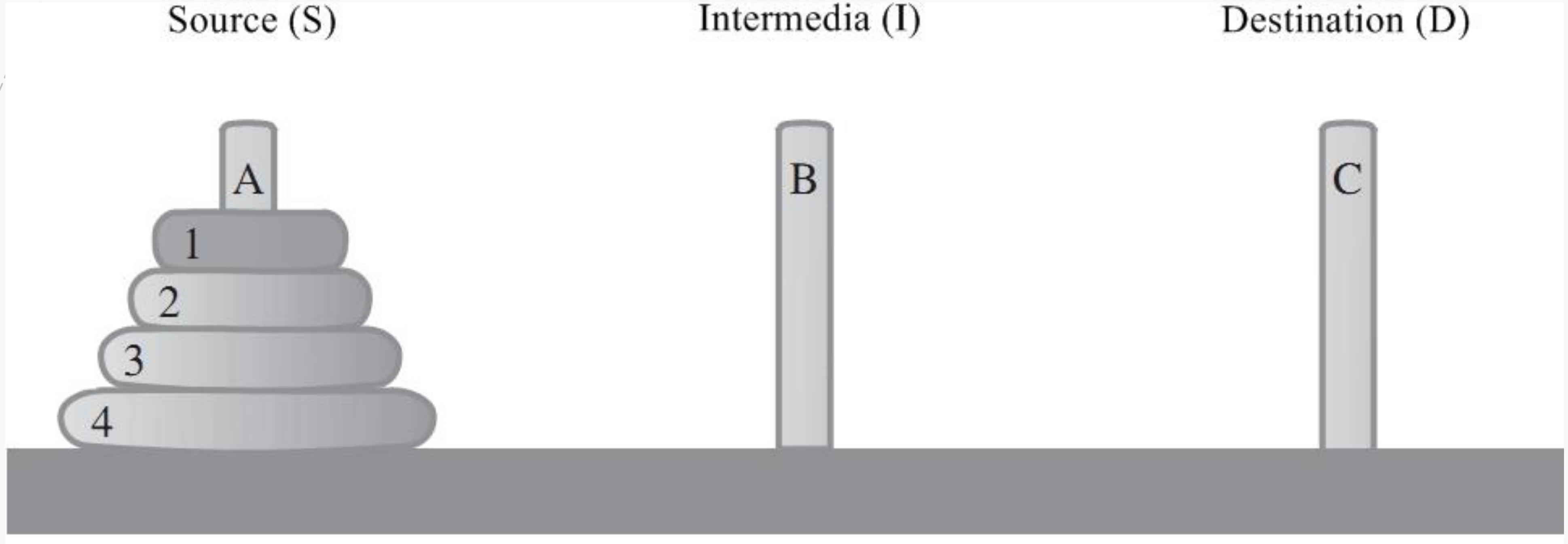
- Divide problems into two smaller problems
  - Solve each one separately (divide again)
  - Usually have 2 recursive calls in main method: one for each half
- Can be non-recursive
- Examples
  - The Towers of Hanoi
  - MergeSort

Towers of Hanoi

- An ancient puzzle consisting of a number of disks placed on three columns (A, B, C)
- Objectives
  - Transfer all disks from column A to column C
- Rules
  - Only one disk can be moved at a time
  - No disk can be placed on a disk that is smaller than itself

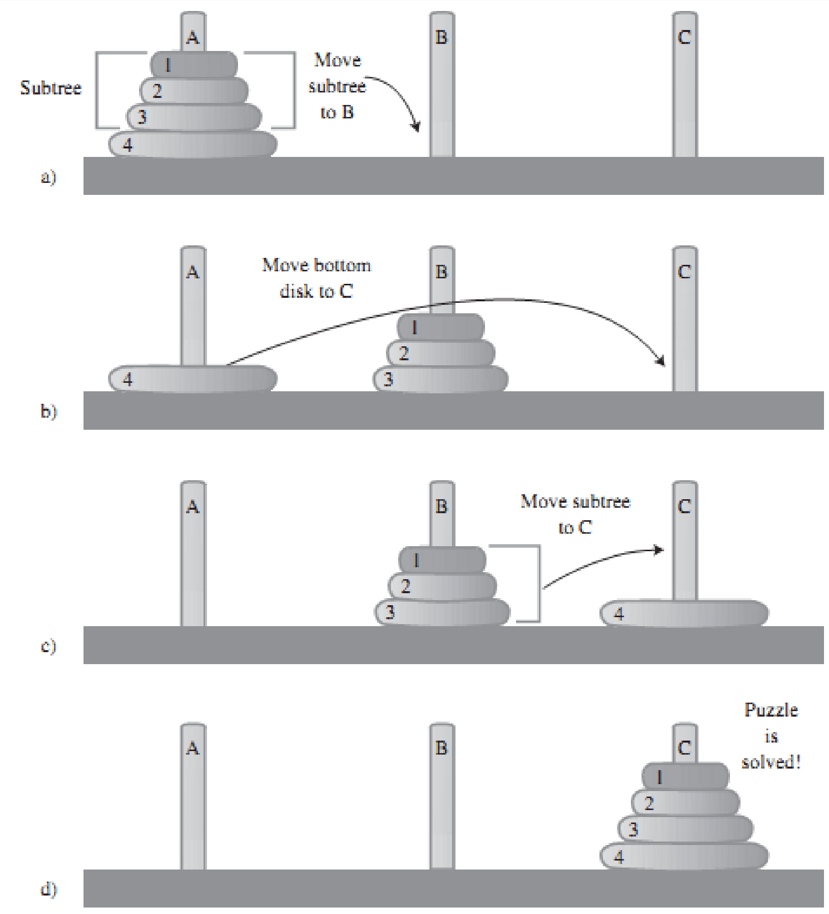


Algorithm



- Move first n-1 subtree from S to I
- Move the largest disk from S to D
- Move the subtree from I to D

Algorithm


A
B
C
Subtree
1
2
3
4
Move
subtree
to B
a)
Move bottom
disk to C
b)
Move subtree
to C
c)
Puzzle
is
solved!
d)

Implementation

```
public static void doTowers(int topN,
                            char from, char inter, char to)
   {
   if(topN==1)
      System.out.println("Disk 1 from " + from + " to "+ to);
   else
      {
      doTowers(topN-1, from, to, inter);  // from-->inter

      System.out.println("Disk " + topN +
                         " from " + from + " to "+ to);
      doTowers(topN-1, inter, from, to);  // inter-->to
      }
   }
```

Implementation

- **TowersApp.java**
- Include a counter and print the number of recursive steps for different number of disks

Classification of recursive functions by number of recursive calls

- Considering the maximum number of recursive calls that may be started from within the body of a single activation:
- **Linear recursion**: Only 1 recursive call (to itself) inside the recursive function (e.g., binary search, factorial).
- **Binary recursion**: There exactly 2 recursive calls (to itself) inside the recursive function (e.g., Fibonacci number) .
- **Multiple recursion**: There are 3 or more recursive calls (to itself) inside the recursive function (e.g., "Sierpinski triangle").

Tail recursion

- A recursion is a tail recursion if:
  - Any recursive call that is made from one context is the very last operation in that context,
  - with the return value of the recursive call (if any) immediately returned by the enclosing recursion.

```
public static int factorial(int n) {
        return factorialHelper(n, 1);
    }

    private static int factorialHelper(int n, int result) {
        if (n == 0) {
            return result;
        } else {
            return factorialHelper(n - 1, result * n);
        }
    }
```

```
public class FactorialCalculator {

    public static int factorial(int n) {
        if (n < 0) {
            throw new IllegalArgumentException("Factorial is not
                       defined for negative numbers.");
        } else if (n == 0) {
            return 1;
        } else {
            return n * factorial(n - 1);
        }
    }
}
```

Indirect recursion

- If ***f()*** calls itself, it is direct recursive
- If ***f()*** calls ***g()***, and ***g()*** calls ***f()***. It is indirect recursion. The chain of intermediate calls can be of an arbitrary length, as in:

$$f() \rightarrow f_1() \rightarrow f_2() \rightarrow \ldots \rightarrow f_n() \rightarrow f()$$

```java
public class IndirectRecursionExample {

    public static void functionA(int n) {
        if (n > 0) {
            System.out.print(n + " ");
            functionB(n - 1);
        }
    }

    public static void functionB(int n) {
        if (n > 0) {
            System.out.print(n + " ");
            functionA(n - 1);
        }
    }

    public static void main(String[] args) {
        int n = 5;
        System.out.print("Indirect Recursion Output: ");
        functionA(n);
    }
}
```

```
Indirect Recursion Output: 5 4 3 2 1 1 2 3 4 5
```

Nested recursion

- A function is not only defined in terms of itself but also is used as one of the parameters
- Examples: Ackermann function
  - A(0, y) = y + 1
  - A(x, 0) = A(x - 1, 1)
  - A(x, y) = A(x - 1, A(x, y - 1))

$$A(n, m) = \begin{cases} m + 1 & \text{if } n = 0 \\ A(n - 1, 1) & \text{if } n > 0, m = 0 \\ A(n - 1, A(n, m - 1)) & \text{otherwise} \end{cases}$$

This function is interesting because of its value grows rapidly, even for small inputs.

A(3,1) = 24 - 3

A(4,1) = 265536 - 3

Eliminate recursion

- Some algorithms are naturally in recursive form (merge sort, Hanoi Tower, etc.)
- But recursion is not efficient

  → try to transform to non-recursive approach

Example: Fibonacci sequence

- A well-known example of a recursive function is the Fibonacci sequence.
- The first term is 0, the second term is 1 and each successive term is defined to be the sum of the two previous terms, i.e. :
  - fib(0) is 0
  - fib(1) is 1
  - fib(2) is 1
  - fib(n) is fib(n-1) + fib(n-2)
  - 1, 1, 2, 3, 5, 8, 13, 21, ...

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n \geq 2 \end{cases}$$

Example: Fibonacci sequence

```java
public class FibonacciSequence {
    public static int fibonacci(int n) {
        if (n <= 0) {
            return 0;
        } else if (n == 1) {
            return 1;
        } else {
            return fibonacci(n - 1) + fibonacci(n - 2);
        }
    }
    public static void main(String[] args) {
        int n = 10;
        for (int i = 0; i <= n; i++) {
            System.out.print("fibonacci(" + i + "): " + fibonacci(i) + "\n");
        }
    }
}
```

Execution:

```
fibonacci(0): 0
fibonacci(1): 1
fibonacci(2): 1
fibonacci(3): 2
fibonacci(4): 3
fibonacci(5): 5
fibonacci(6): 8
fibonacci(7): 13
fibonacci(8): 21
fibonacci(9): 34
fibonacci(10): 55
```

Excessive recursion

- Some recursive methods repeats the computations for some parameters, which results in long computation time even for simple cases.
- For example, consider the Fibonacci sequence.
- In Java it can be implemented recursively as:

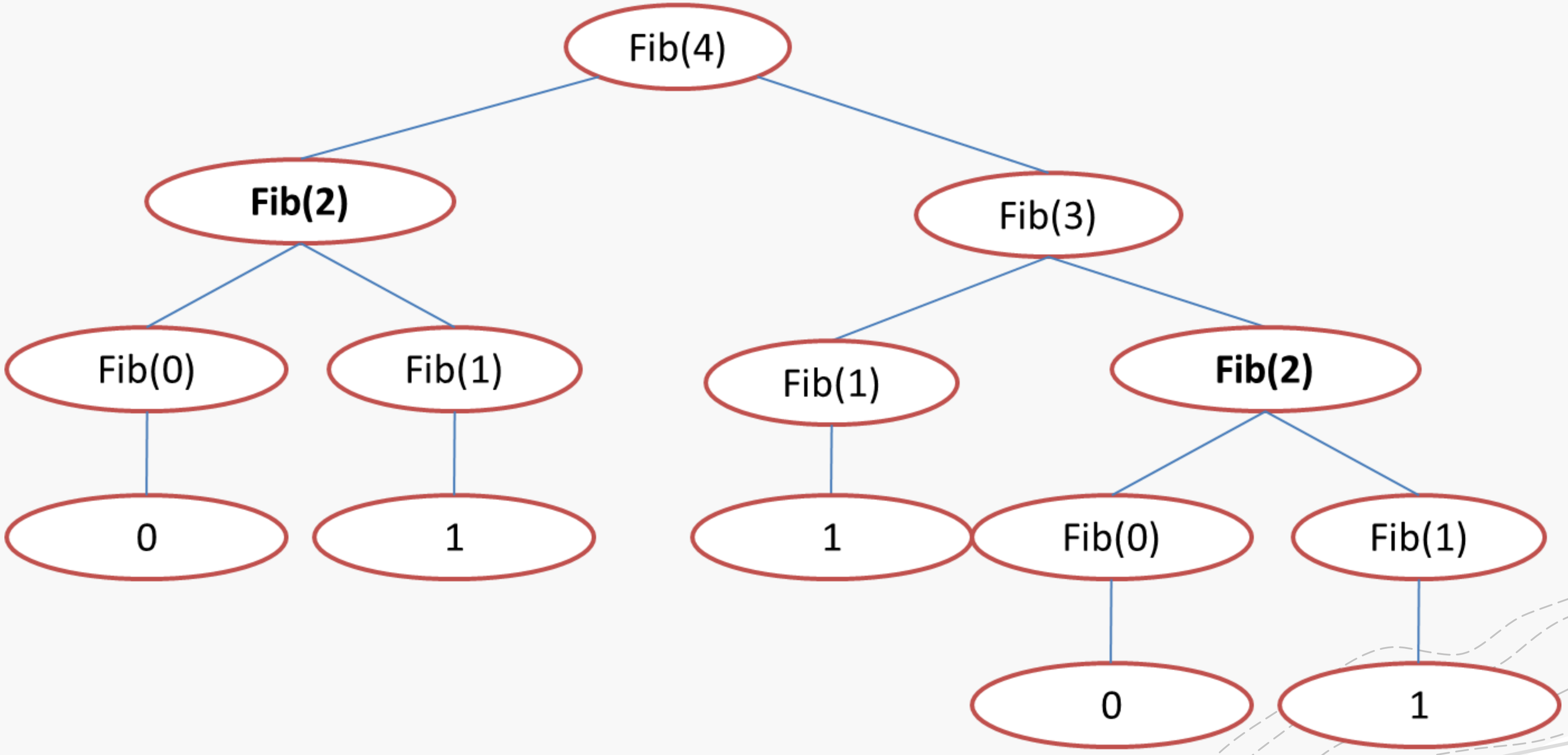
$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n \geq 2 \end{cases}$$

```
/** Returns the nth Fibonacci number (inefficiently). */
public static long fibonacciBad(int n) {
      if (n <= 1)
      return n;
      else
      return fibonacciBad(n-2) + fibonacciBad(n-1);
}
```

This implementation looks very natural but extremely inefficient! (**O(2^n)**)

Excessive recursion

The tree of calls for fibo(4)



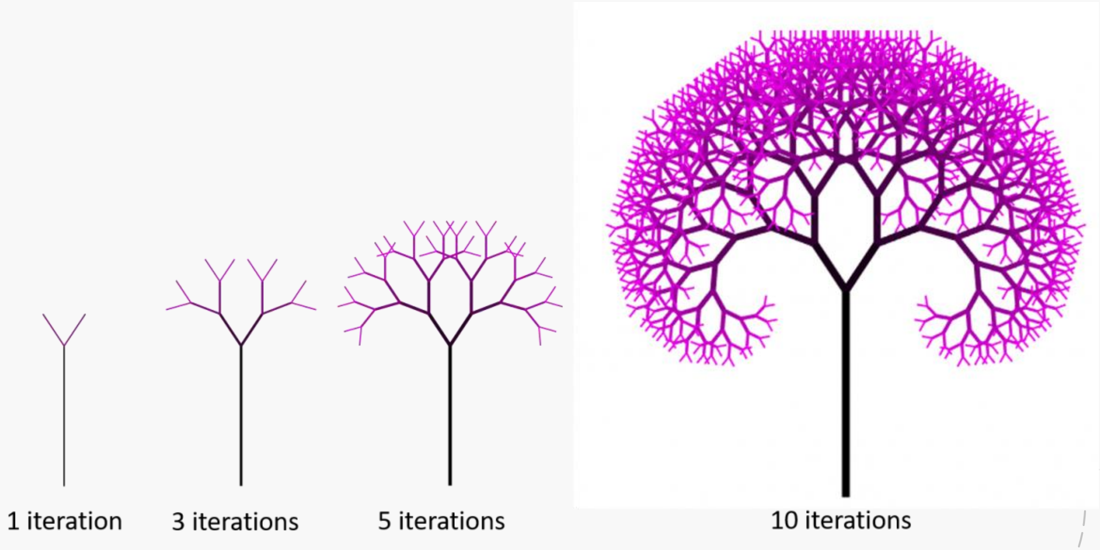
Excessive recursion

- Better Fibonacci implementation: define a recursive method that
  - returns an array with
  - two consecutive Fibonacci numbers $\{F_n, F_{n-1}\}$
  - using the convention $F_{-1} = 0$.

```
/** Returns array containing the pair of Fibonacci numbers, F(n) and F(n-1). */
public static long[ ] fibonacciGood(int n) {
       if (n <= 1) {
               long[ ] answer = {n, 0};
               return answer;
       } else {
               long[ ] temp = fibonacciGood(n - 1); // returns {Fn-1, Fn-2}
               long[ ] answer = {temp[0] + temp[1], temp[0]}; // we want {Fn, Fn-1}
               return answer;
       }
}
```

More Examples – Drawing fractals

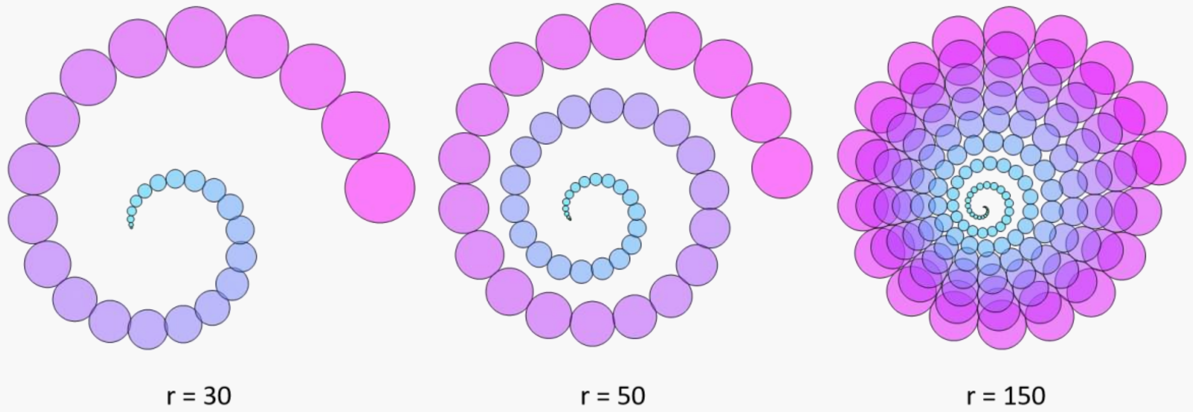
- Inspired by the growth patterns of natural trees.
- Each branch of a fractal tree is divided into smaller branches
- Smaller branches are divided into even smaller branches, and so on, creating a recursive structure.



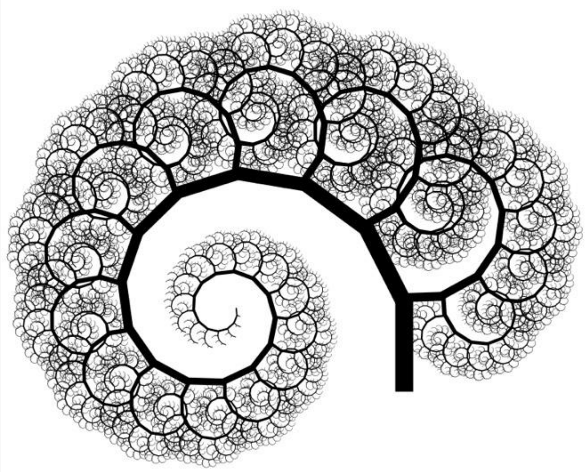
Fractal Tree

More Examples – Drawing fractals

- Starts as a basic geometric shape, often a line or an arc
- Then applying a series of scaling and rotation operations to create smaller copies
- These smaller copies are positioned in a specific arrangement (often following a spiral-like pattern)
- The process is repeated at smaller and smaller scales.

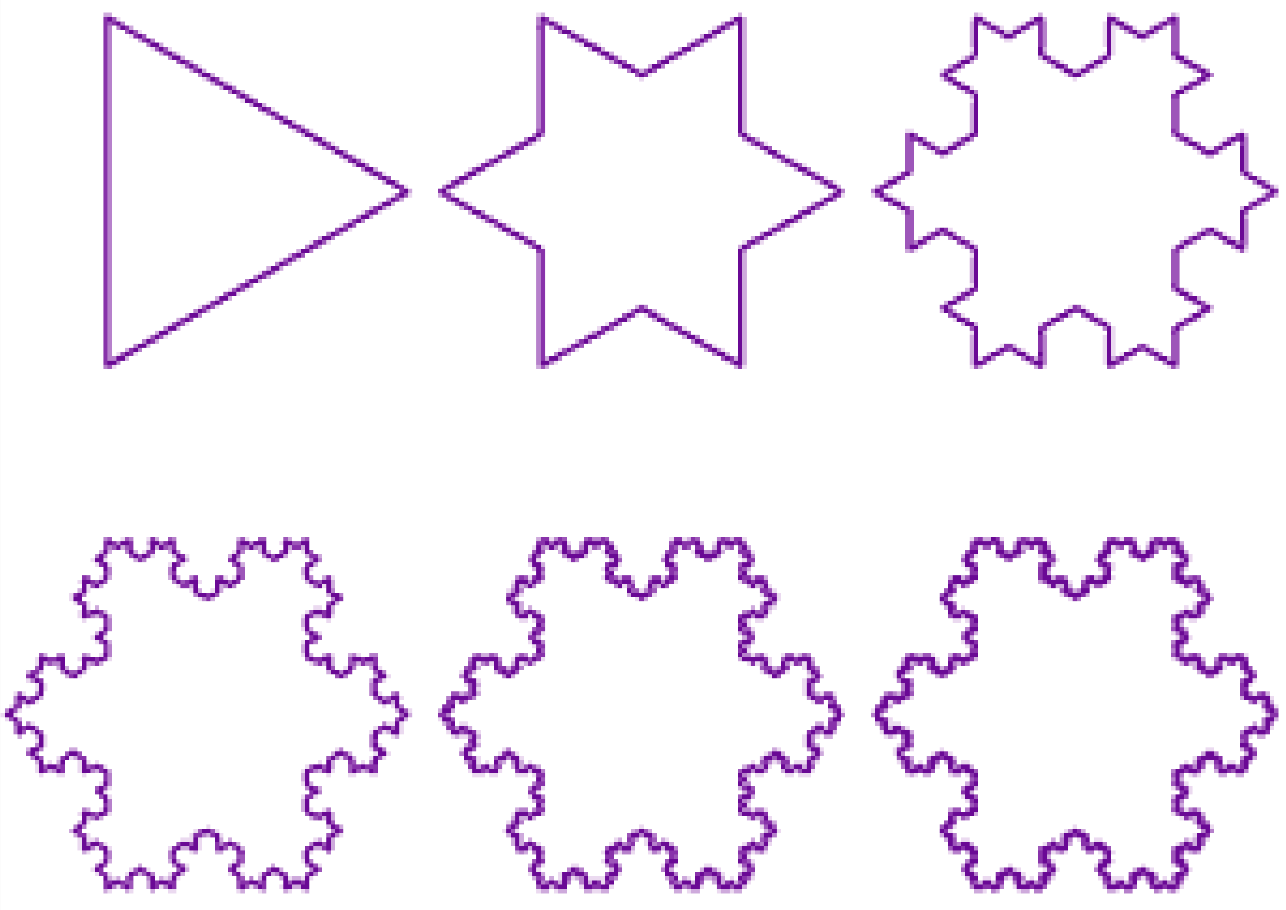


Fractal Spiral



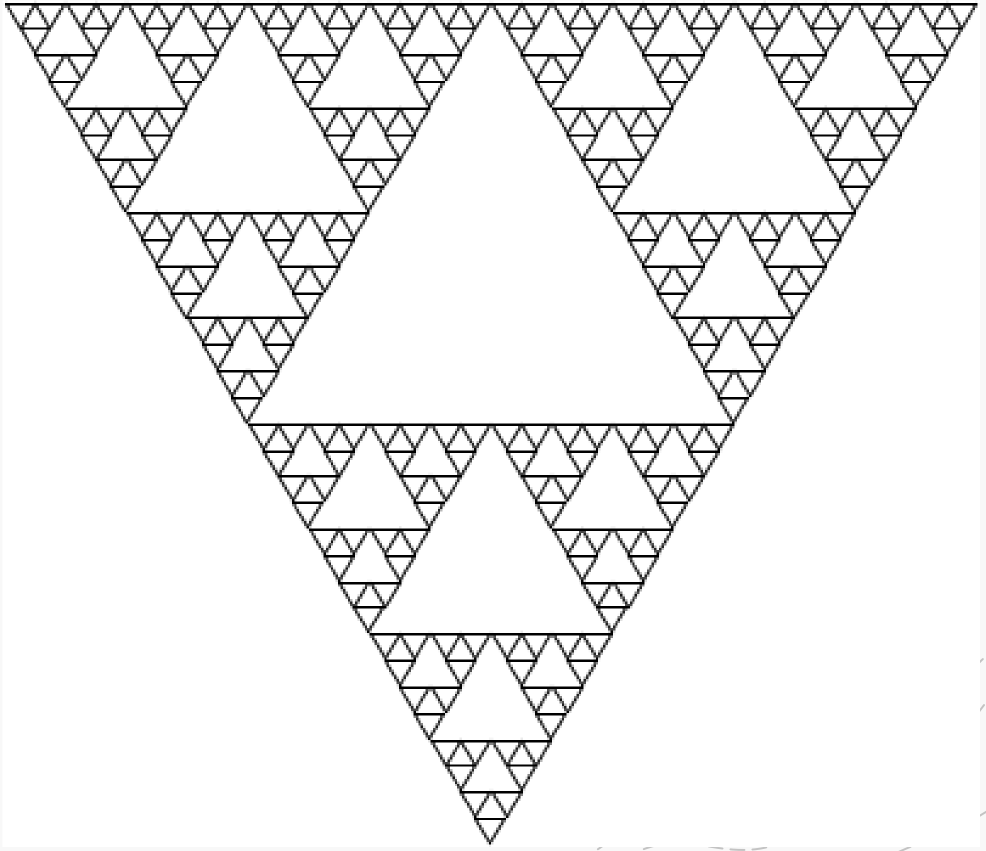
More Examples – Von Knoch snowflakes

- Divide an interval side into three even parts
- Move one-third of side in the direction specified by angle



More Examples – Sierpinski Triangle

- A fractal attractive fixed set
- The overall shape of an equilateral triangle
- Subdivided recursively into smaller equilateral triangles



Recursion vs. Iteration

- Some recursive algorithms can also be easily implemented with loops
- When possible, it is usually better to use iteration, since we don't have the overhead of the run-time stack (as in the previous slide)
- Other recursive algorithms are very difficult to do any other way

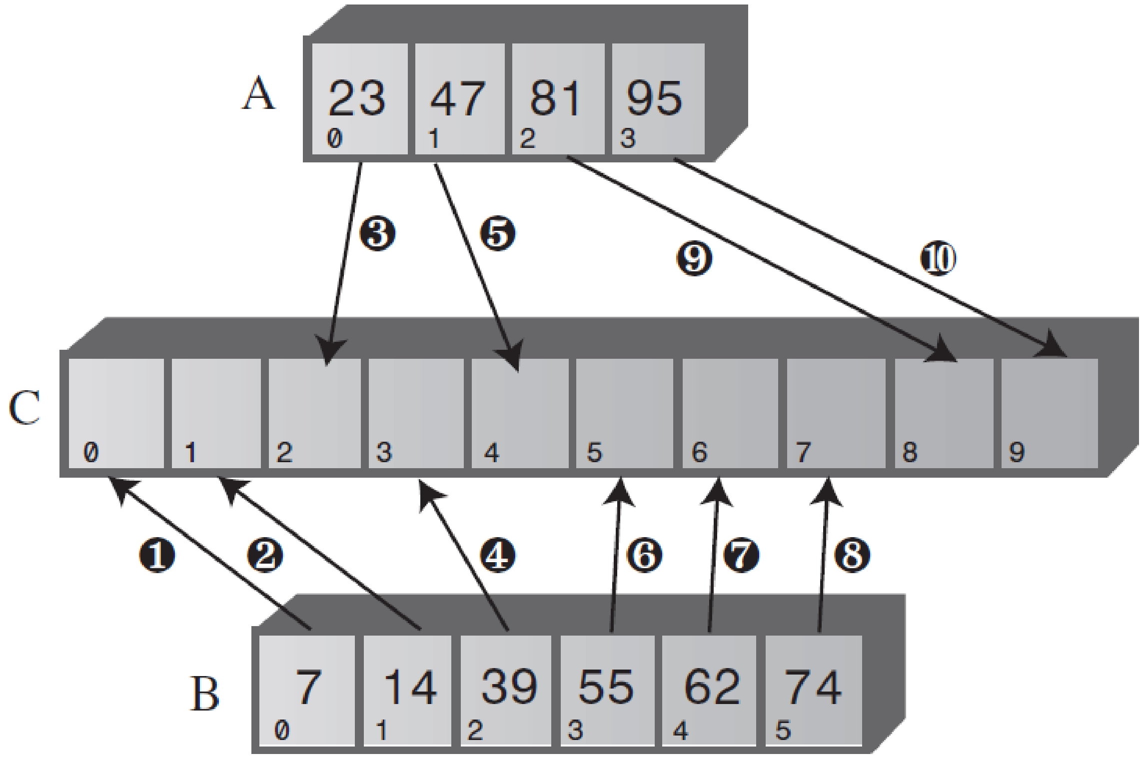
Merge Sort

Merge Sort

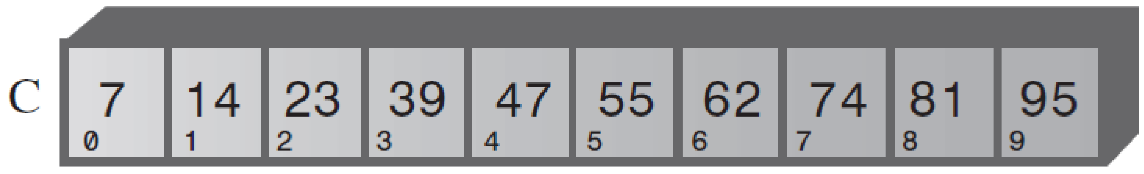
- Simple Sorting Algorithms: $O(N^2)$
  - Bubble Sort, Selection Sort, Insertion Sort
  - Using Sorted Linked List
- MergeSort: **O(NlogN)**
- Approach to MergeSort
  - Merging Two Sorted Arrays
  - Sorting by Merging
  - Efficiency of MergeSort

Merging two sorted arrays

- Given two sorted arrays (A, B)
- Creating sorted array C containing all elements of A, B



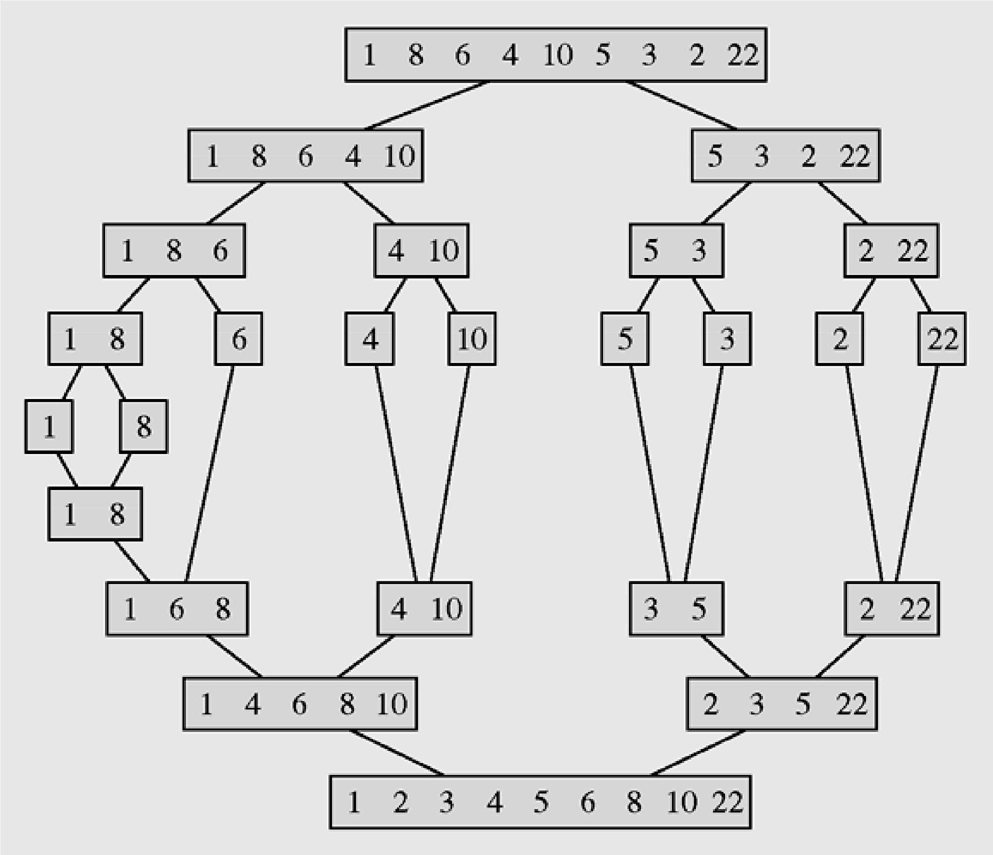
a) Before Merge



b) After Merge

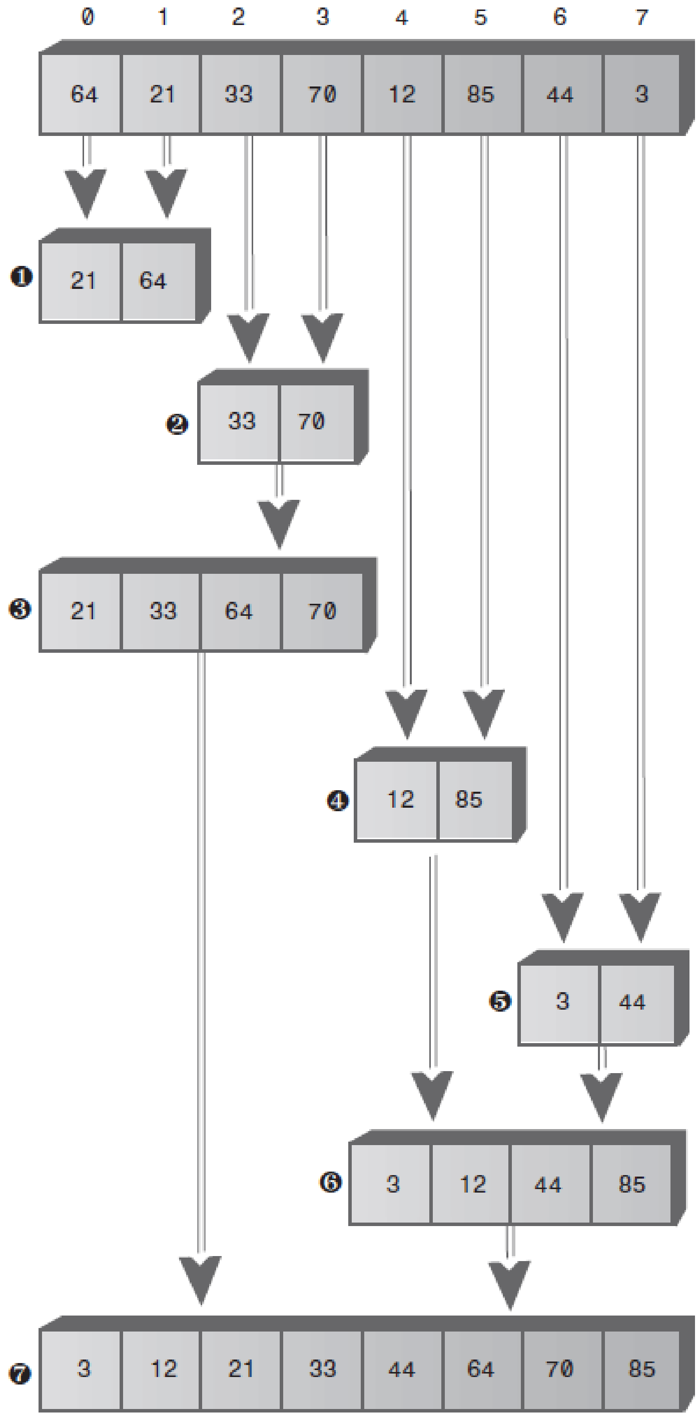
Merge Sort

- Divide an array in halves
- Sort each half: Using recursion
  - Divide half into quarters
  - Sort each of the quarters
  - Merge them to make a sorted half
- Call **merge()** to merge two halves into a single sorted array



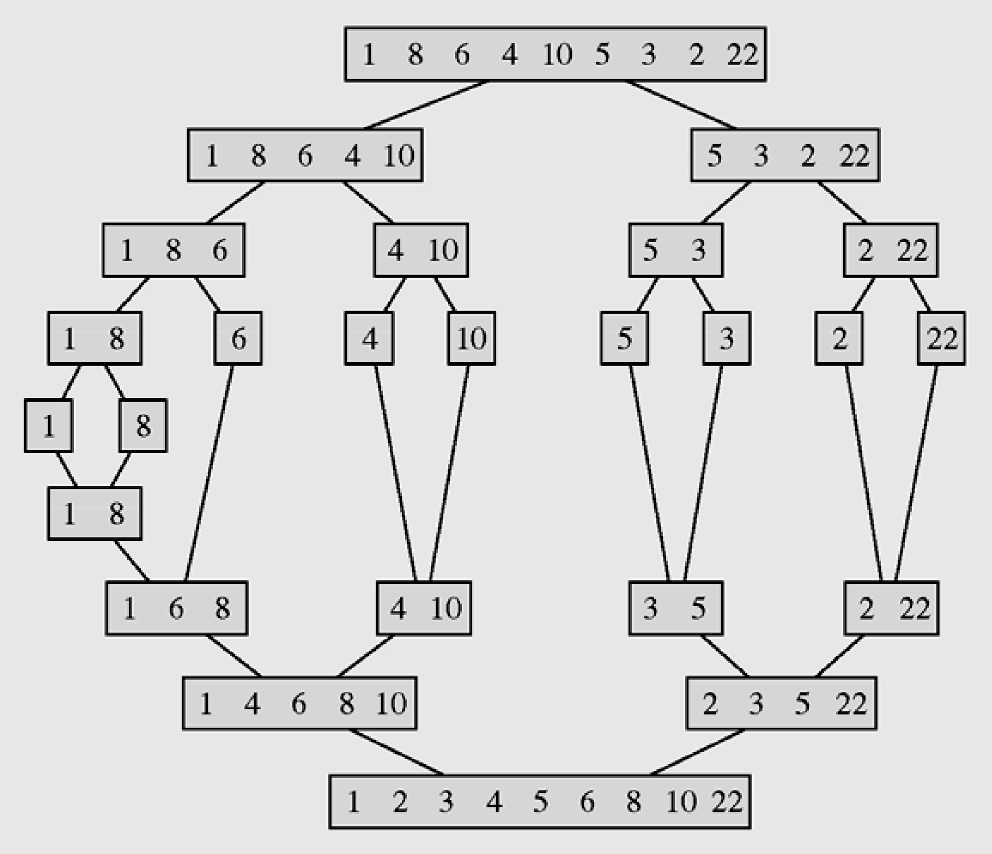
Merge Sort

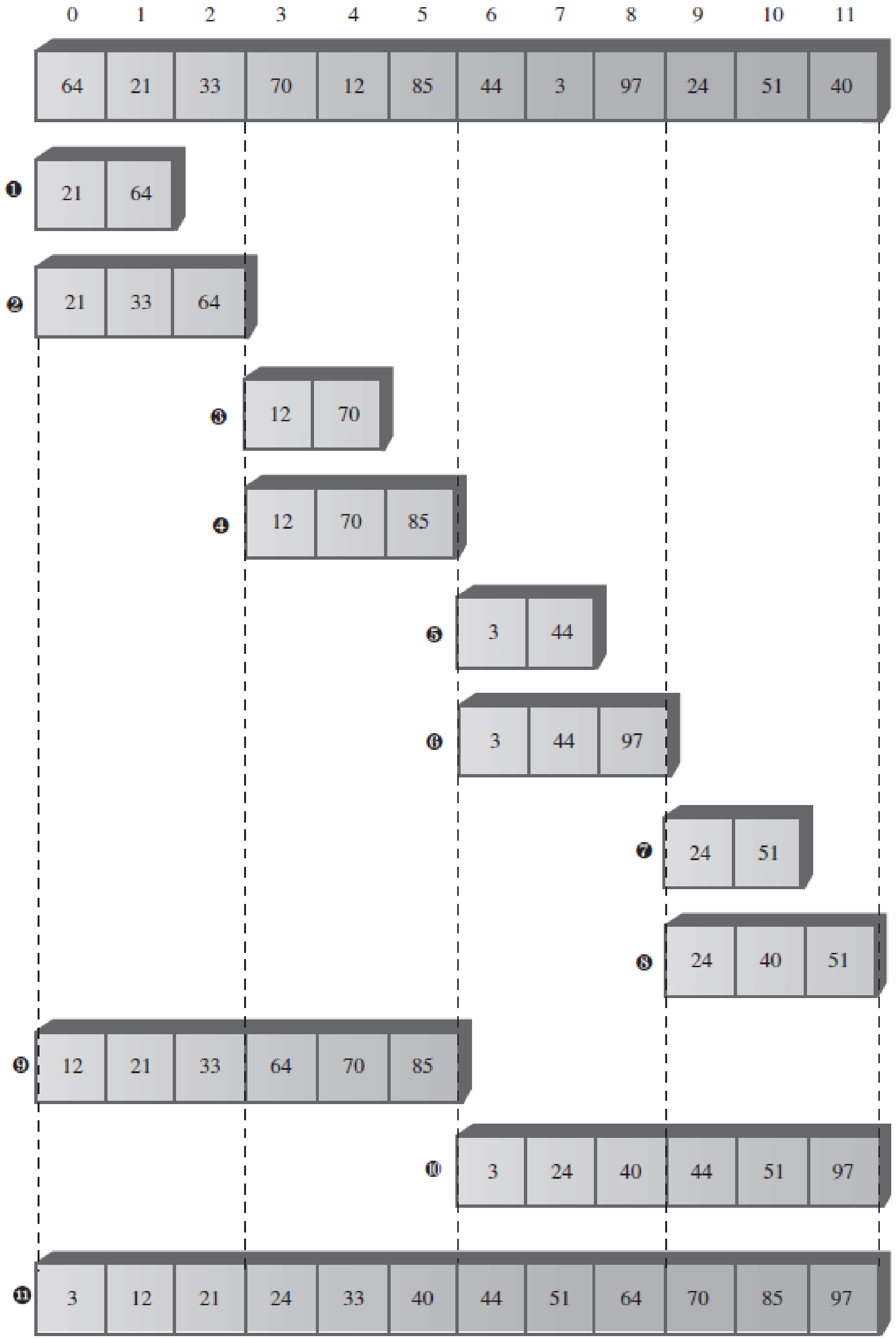
- Divide an array in halves
- Sort each half: Using recursion
  - Divide half into quarters
  - Sort each of the quarters
  - Merge them to make a sorted half
- Call **merge()** to merge two halves into a single sorted array



Merge Sort

- Array size not a power of 2





Merge sort - algorithm

- The merge sort algorithm is defined recursively:
  - If the list is of size 1, it is sorted—we are done;
  - Otherwise:
    - Divide an unsorted list into two sub-lists,
    - Sort each sub-list recursively using merge sort, and
    - Merge the two sorted sub-lists into a single sorted list

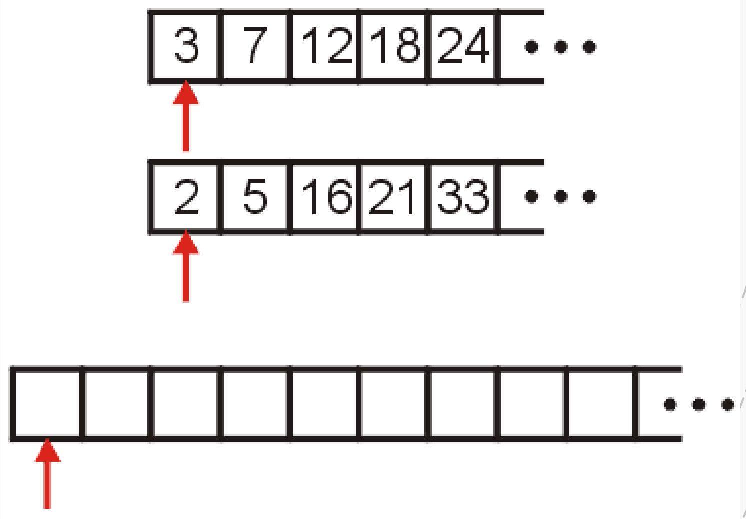
➔ This is the first divide-and-conquer algorithm

Implementation

```
private void recMergeSort(long[] workSpace, int lowerBound,
                                                int upperBound)
   {
   if(lowerBound == upperBound)            // if range is 1,
      return;                              // no use sorting
   else
      {                                    // find midpoint
      int mid = (lowerBound+upperBound) / 2;
                                           // sort low half
      recMergeSort(workSpace, lowerBound, mid);
                                           // sort high half
      recMergeSort(workSpace, mid+1, upperBound);
                                           // merge them
      merge(workSpace, lowerBound, mid+1, upperBound);
      }  // end else
   }  // end recMergeSort
```

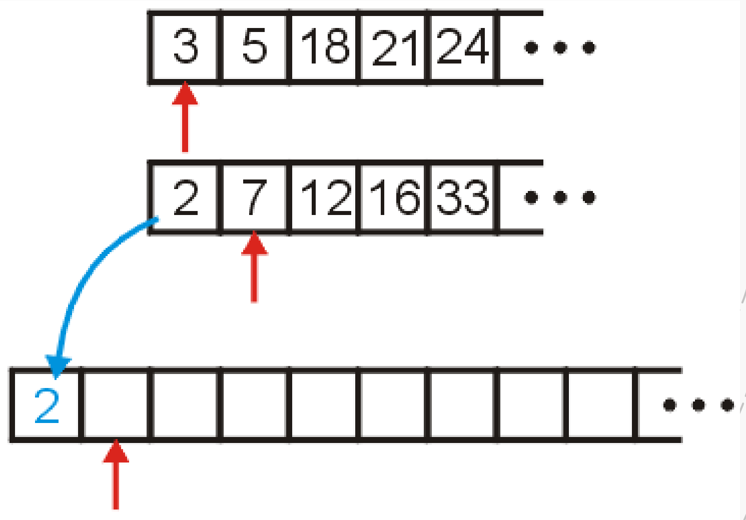
Merge sort – Merging Example

- Consider the two sorted arrays and an empty array
- Define three indices at the start of each array



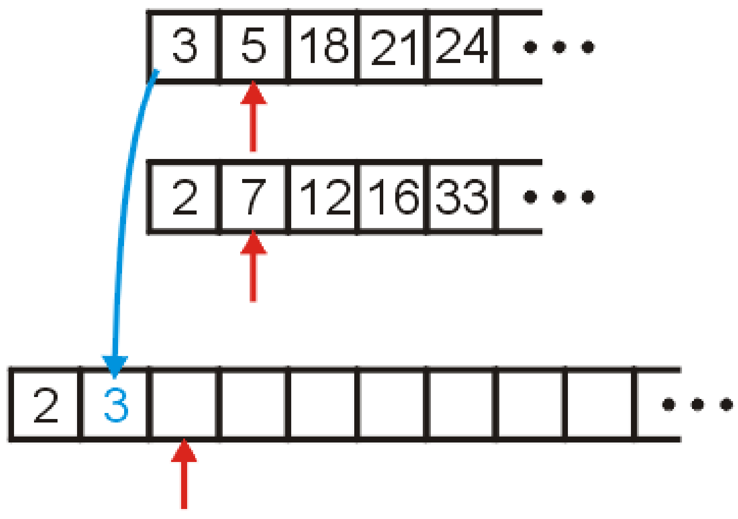
Merging Example

- We compare 2 and 3: 2 < 3
  - Copy 2 down
  - Increment the corresponding indices



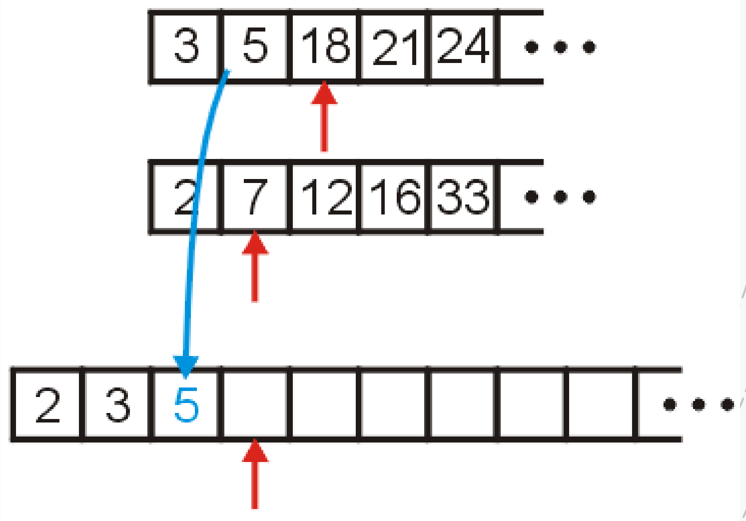
Merging Example

- We compare 3 and 7
  - Copy 3 down
  - Increment the corresponding indices



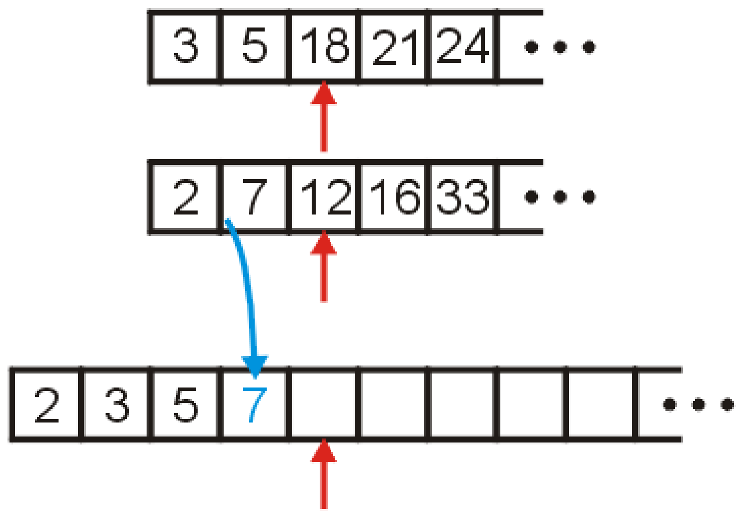
Merging Example

- We compare 5 and 7
  - Copy 5 down
  - Increment the appropriate indices



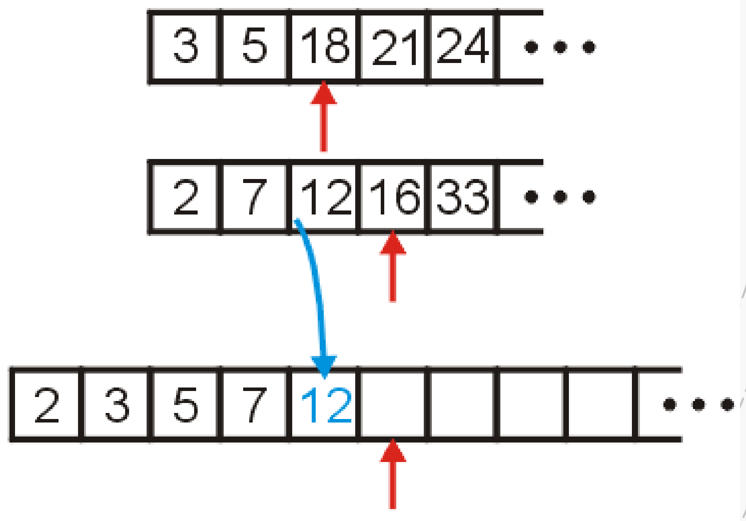
Merging Example

- We compare 18 and 7
  - Copy 7 down
  - Increment…



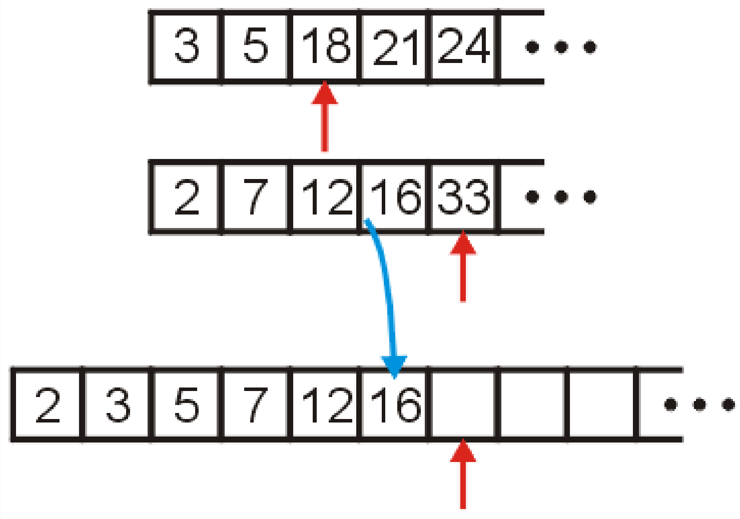
Merging Example

- We compare 18 and 12
  - Copy 12 down
  - Increment…



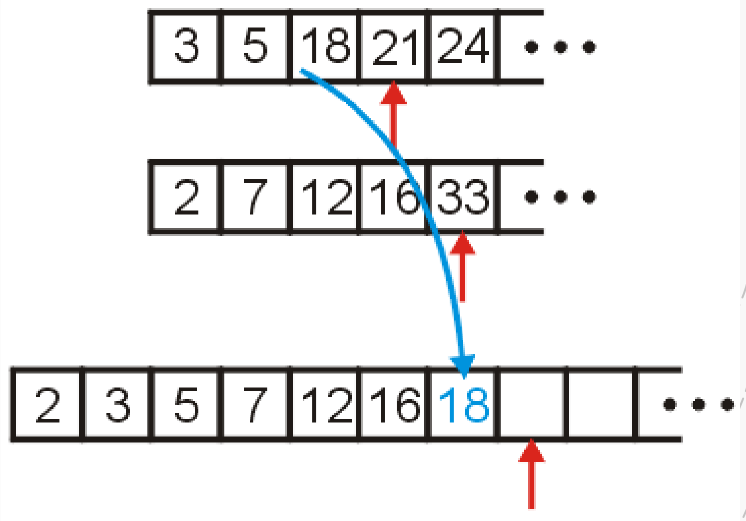
Merging Example

- We compare 18 and 16
  - Copy 16 down
  - Increment…



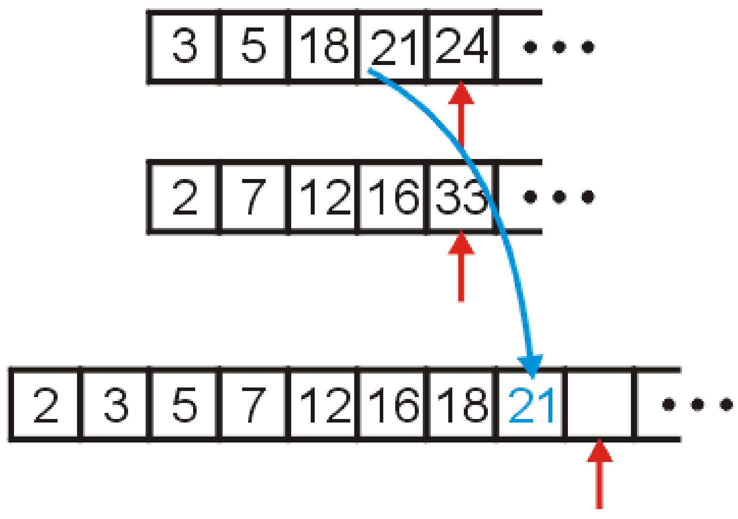
Merging Example

- We compare 18 and 33
  - Copy 18 down
  - Increment...



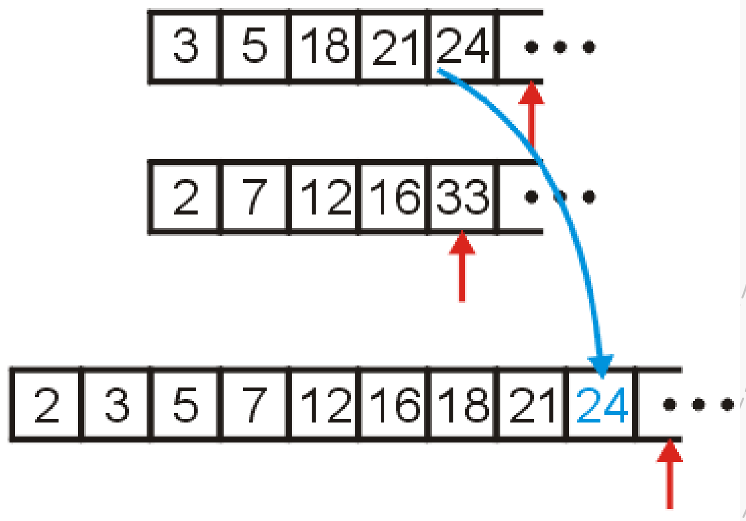
Merging Example

- We compare 21 and 33
  - Copy 21 down
  - Increment…



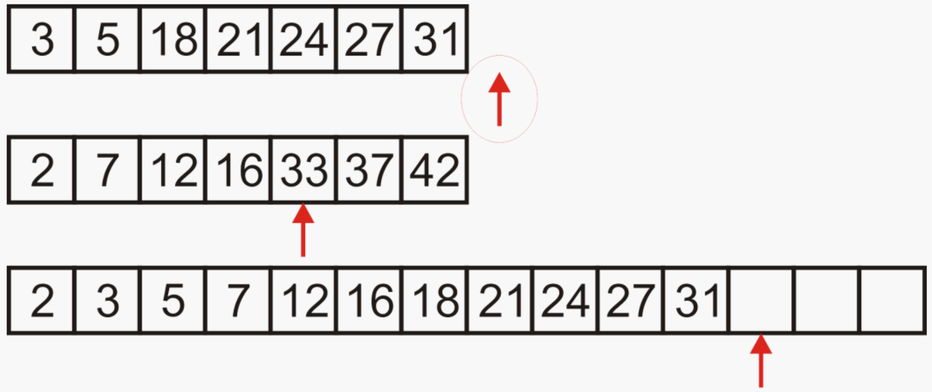
Merging Example

- We compare 24 and 33
  - Copy 24 down
  - Increment…

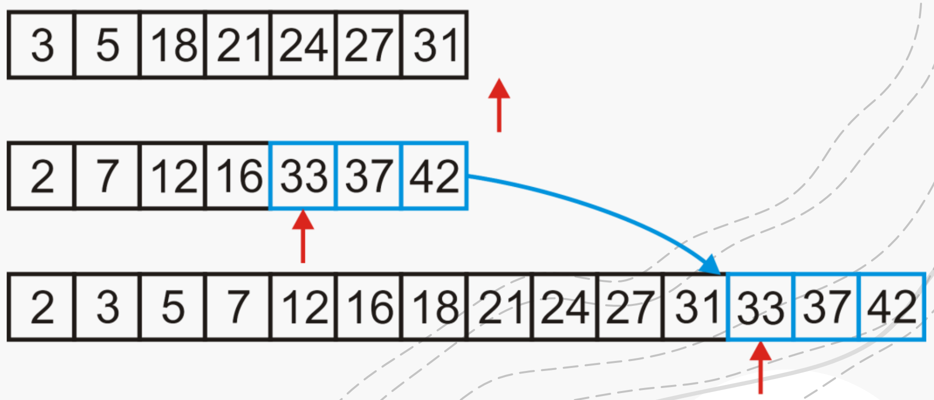


Merging Example

- We would continue until we have passed beyond the limit of one of the two arrays



- After this, we simply copy over all remaining entries in the non-empty array



Code Implementation

```
merge(workSpace, lowerBound, mid+1, upperBound);
```

```
//-------------------------------------------------------------
private void merge(long[] workSpace, int lowPtr,
                        int highPtr, int upperBound)
   {
   int j = 0;                             // workspace index
   int lowerBound = lowPtr;
   int mid = highPtr-1;
   int n = upperBound-lowerBound+1;       // # of items

   while(lowPtr <= mid && highPtr <= upperBound)
      if( theArray[lowPtr] < theArray[highPtr] )
         workSpace[j++] = theArray[lowPtr++];
      else
         workSpace[j++] = theArray[highPtr++];

   while(lowPtr <= mid)
      workSpace[j++] = theArray[lowPtr++];

   while(highPtr <= upperBound)
      workSpace[j++] = theArray[highPtr++];

   for(j=0; j<n; j++)
      theArray[lowerBound+j] = workSpace[j];
   }  // end merge()
//-------------------------------------------------------------
```

Efficiency of Merge Sort: O(NlogN)

TABLE 6.4 Number of Operations When N Is a Power of 2

N	$\log_2 N$	Number of Copies into Workspace ($N*\log_2 N$)	Total Copies	Comparisons Max (Min)
2	1	2	4	1 (1)
4	2	8	16	5 (4)
8	3	24	48	17 (12)
16	4	64	128	49 (32)
32	5	160	320	129 (80)
64	6	384	768	321 (192)
128	7	896	1792	769 (448)

Summary

- Recursive definitions are programming concepts that define themselves
- Some value of its arguments causes a recursive method to return without calling itself. This is called the ***base case***.
- Recursive definitions serve two purposes:
  - Generating new elements
  - Testing whether an element belongs to a set
- Recursive definitions are frequently used to define functions and sequences of numbers
- Tail recursion is characterized using only one recursive call at the very end of a method implementation.

Vietnam National University of HCMC
International University
School of Computer Science and Engineering

THANK YOU

Dr Vi Chi Thanh - vcthanh@hcmiu.edu.vn
https://vichithanh.github.io

SCAN ME