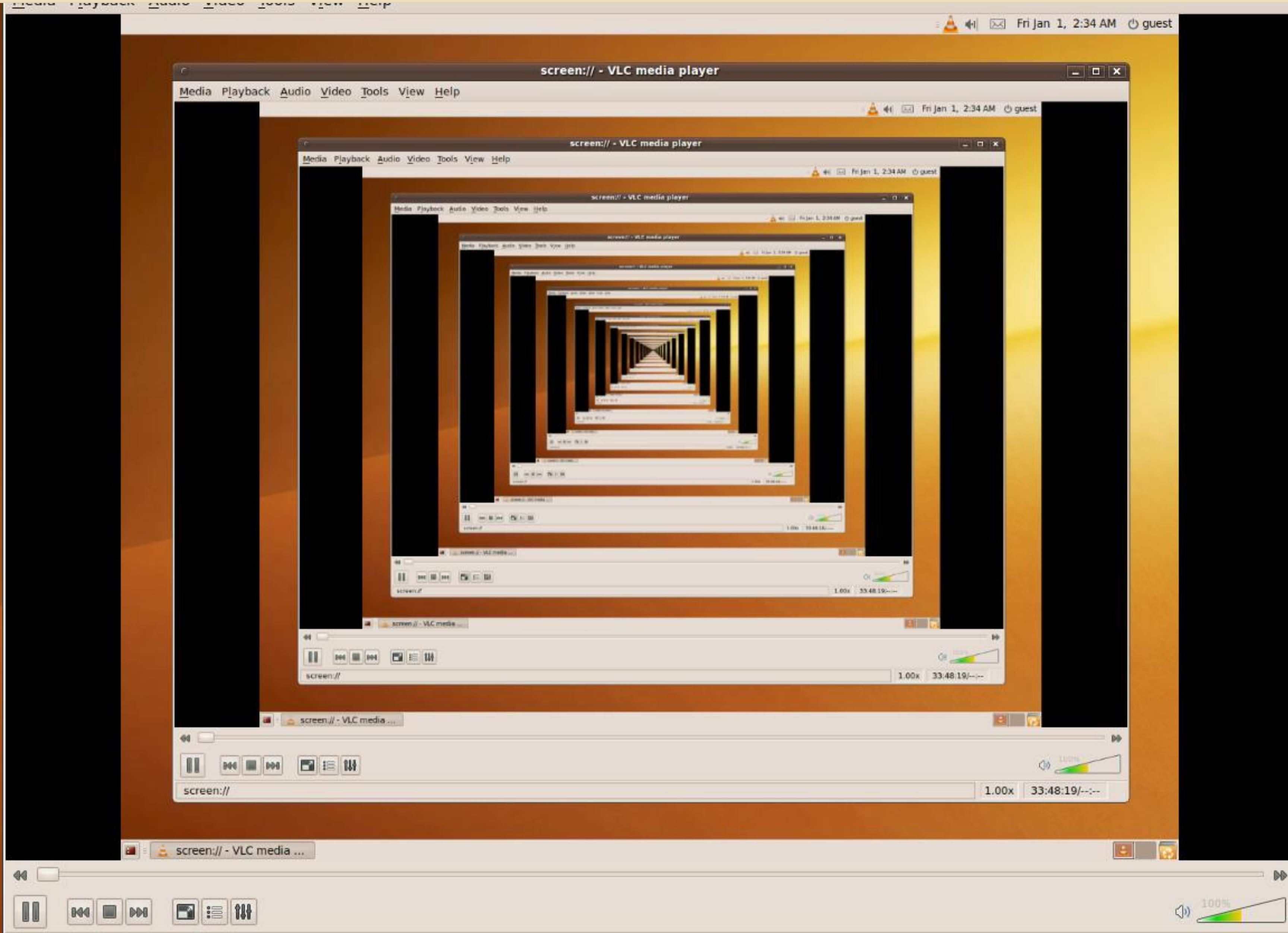
RECURSION

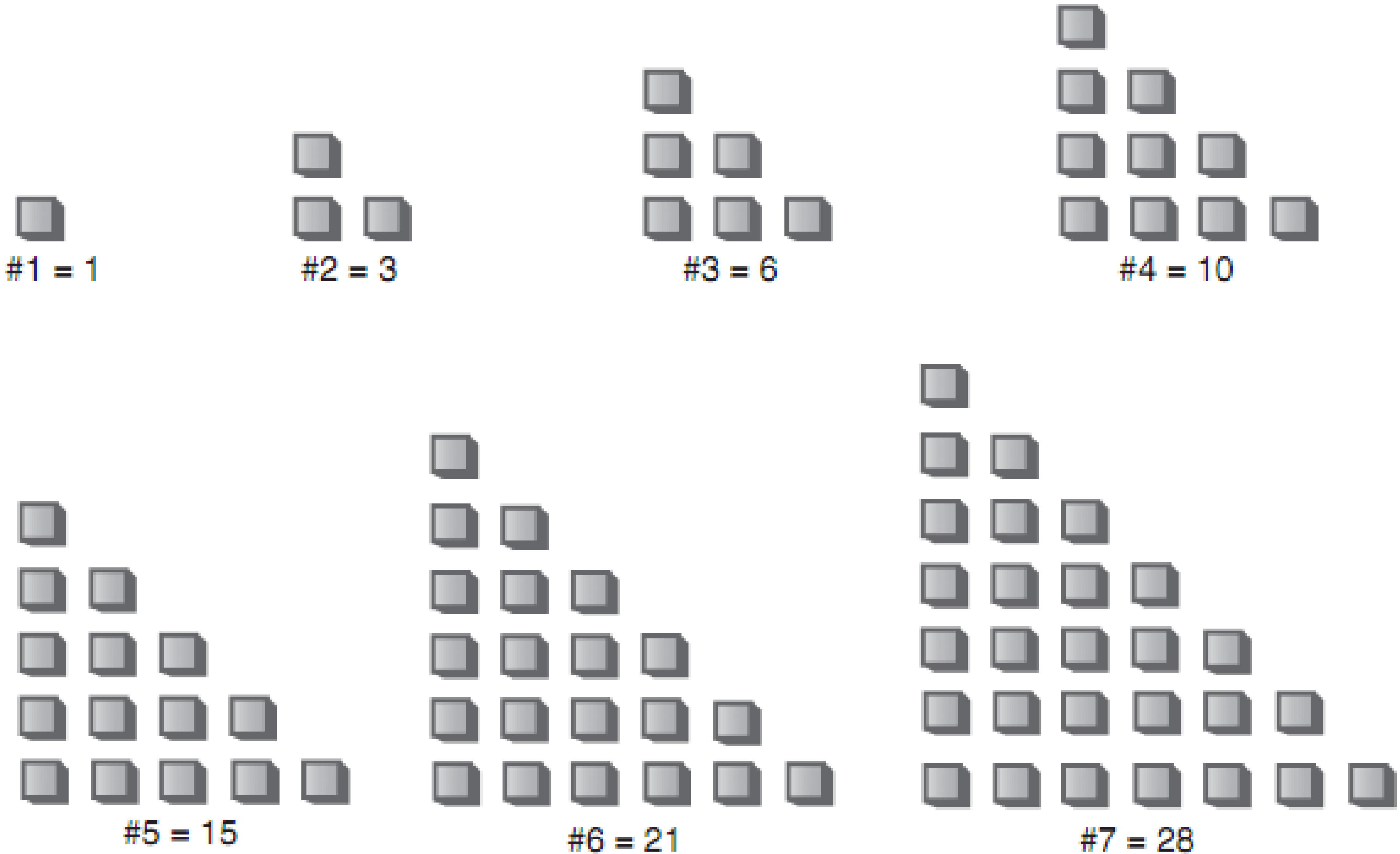
Tran Thanh Tung

Contents

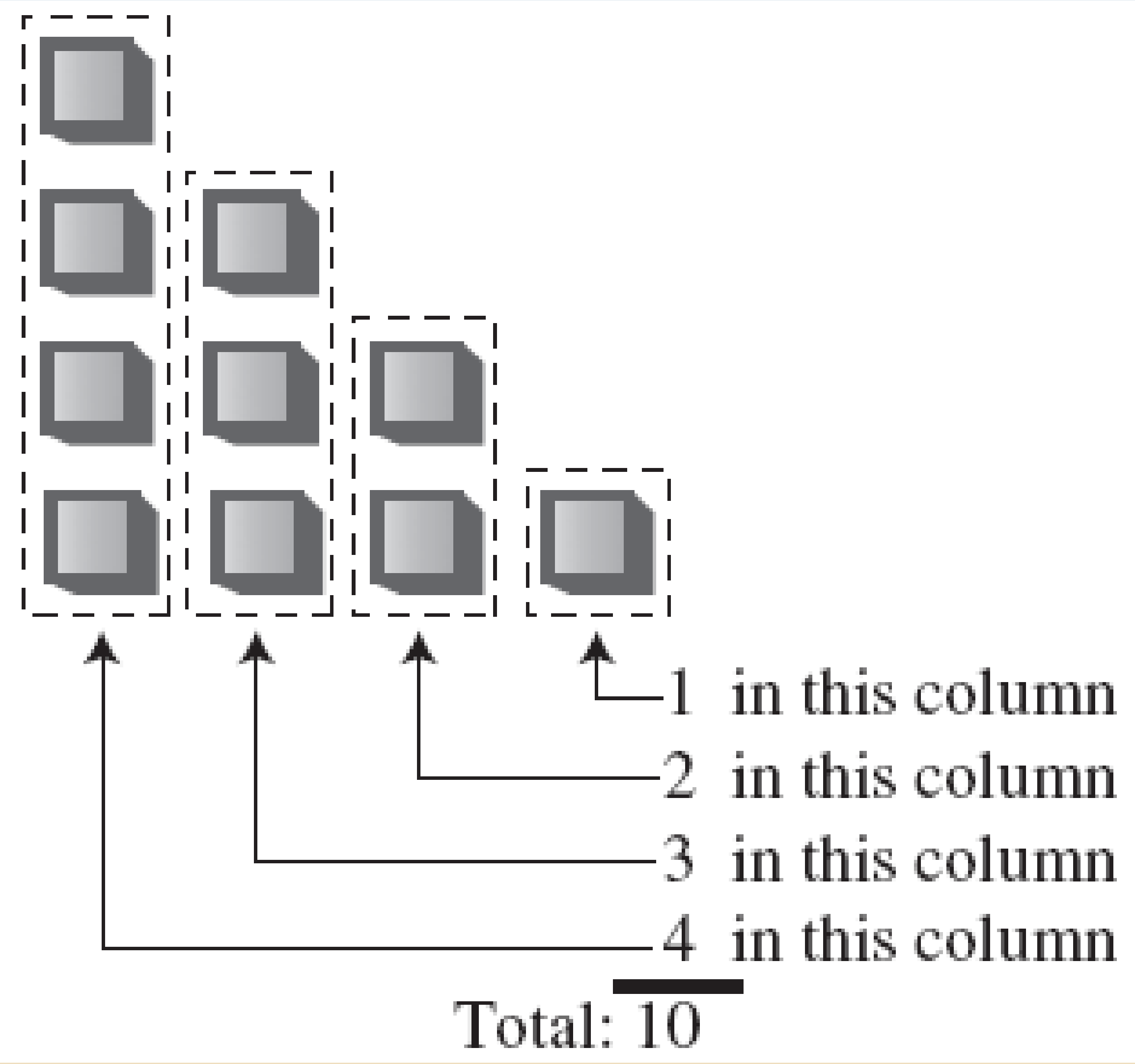
Introduction



Triangular Numbers: Examples



5 Finding nth Term using a Loop

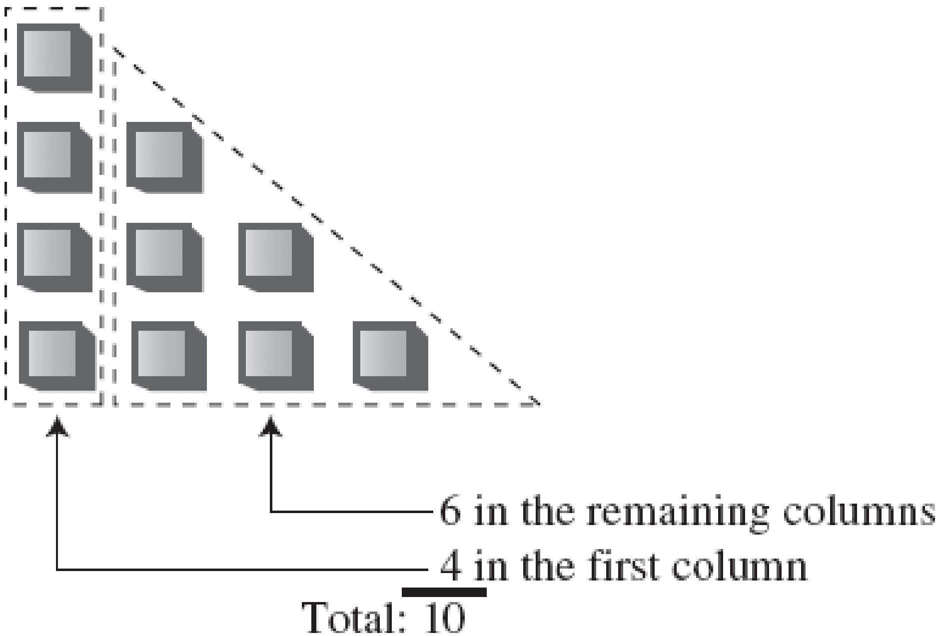


```
int triangle(int n)
   {
   int total = 0;

   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

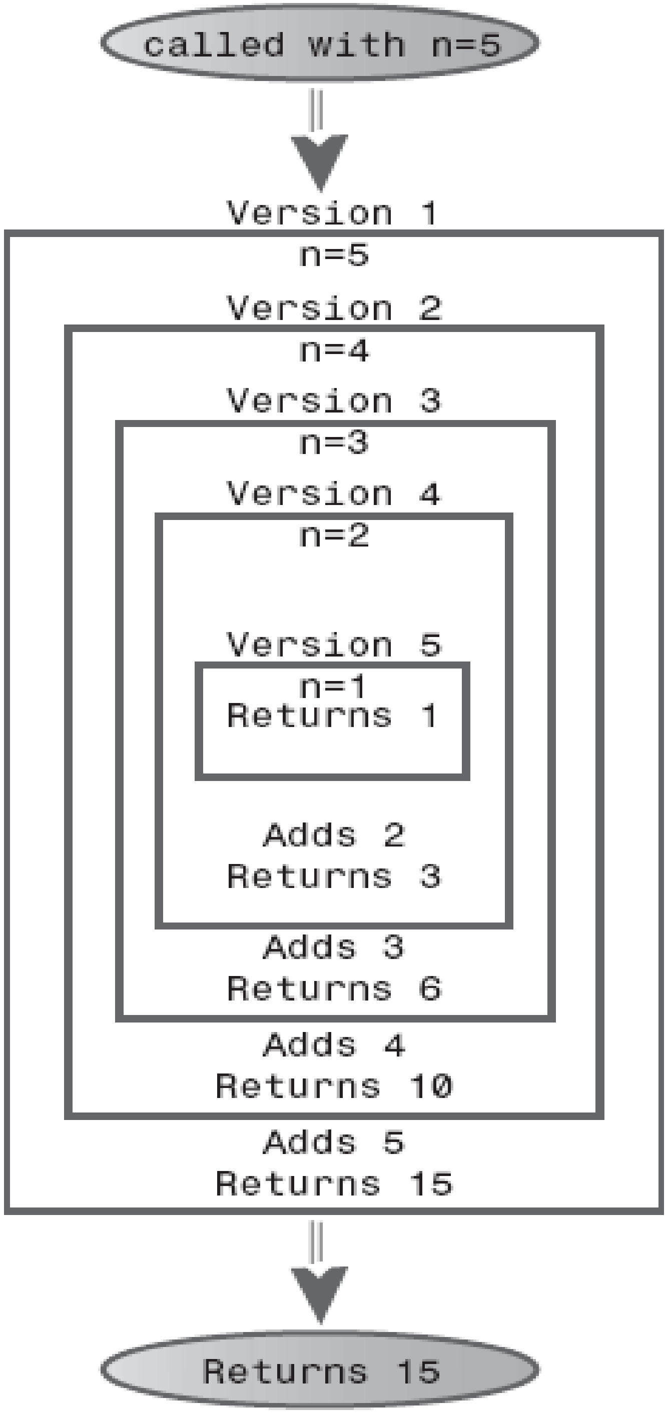

```
int triangle(int n)
    {
    return( n + triangle(n-1) );  // (incomplete version)
    }
```

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



Recursion Characteristics

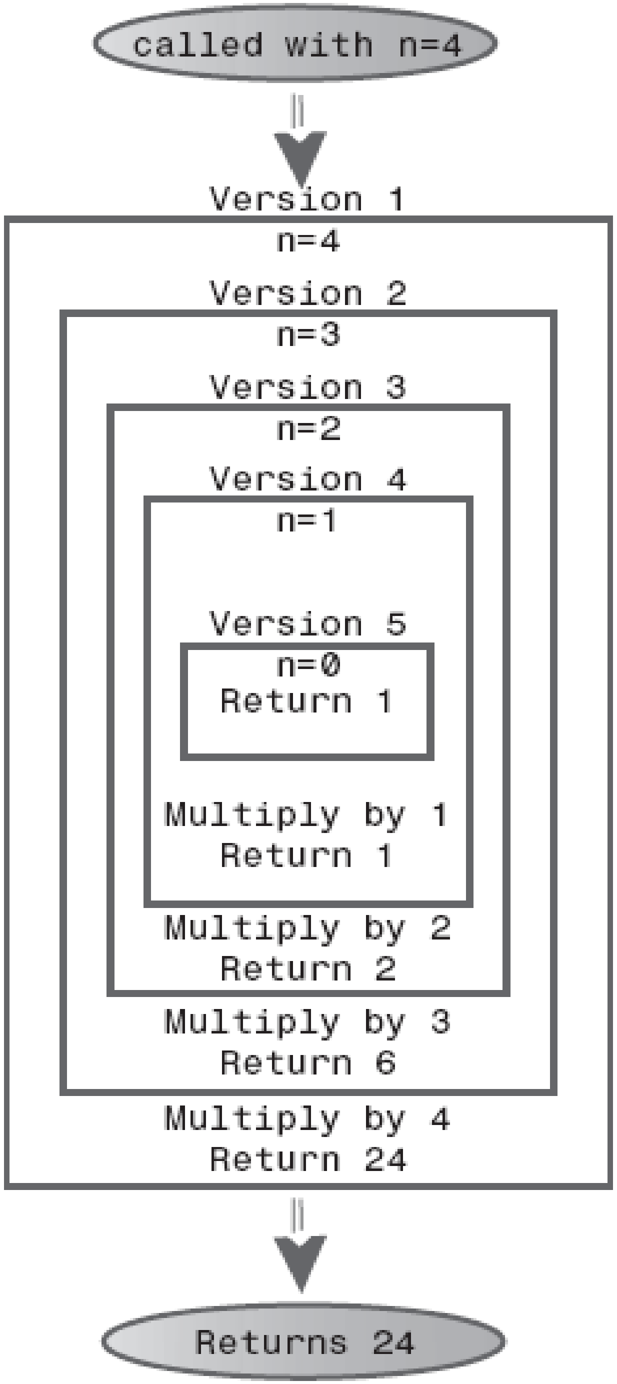
- It calls itself
- When it calls itself, it does so to solve a smaller problem
- There is some version of the problem that is simple enough that the routine can solve it, and return, without calling itself
- Is recursion efficient?
  - No
  - Address of calling methods must be remembered (in stack)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

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

Factorials


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



Binary Search: Recursion vs. Loop

Loop

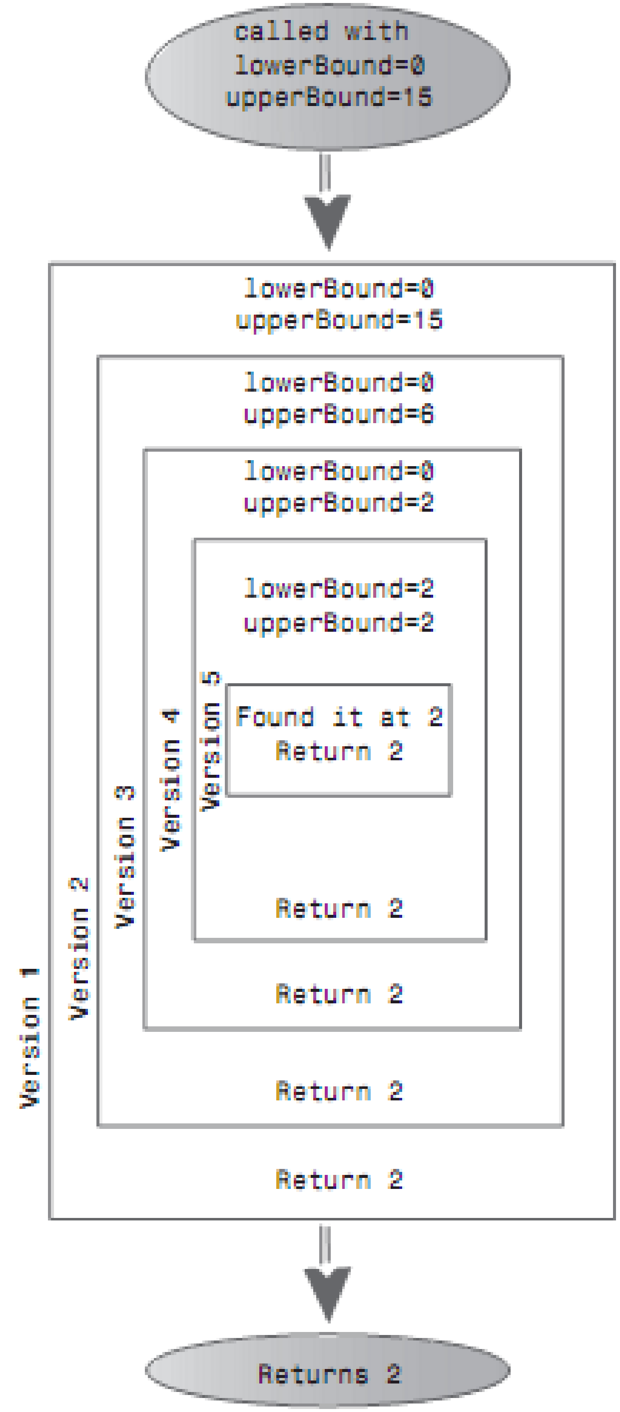
```
public int find(long searchKey)
   {
   int lowerBound = 0;
   int upperBound = nElems-1;
   int curIn;

   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()
```

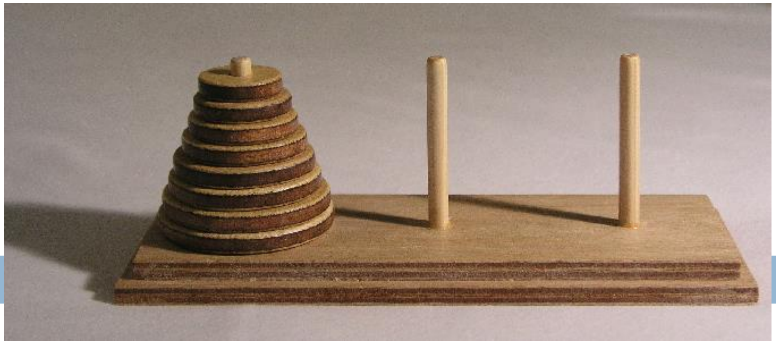


Divide-and-conquer

- 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

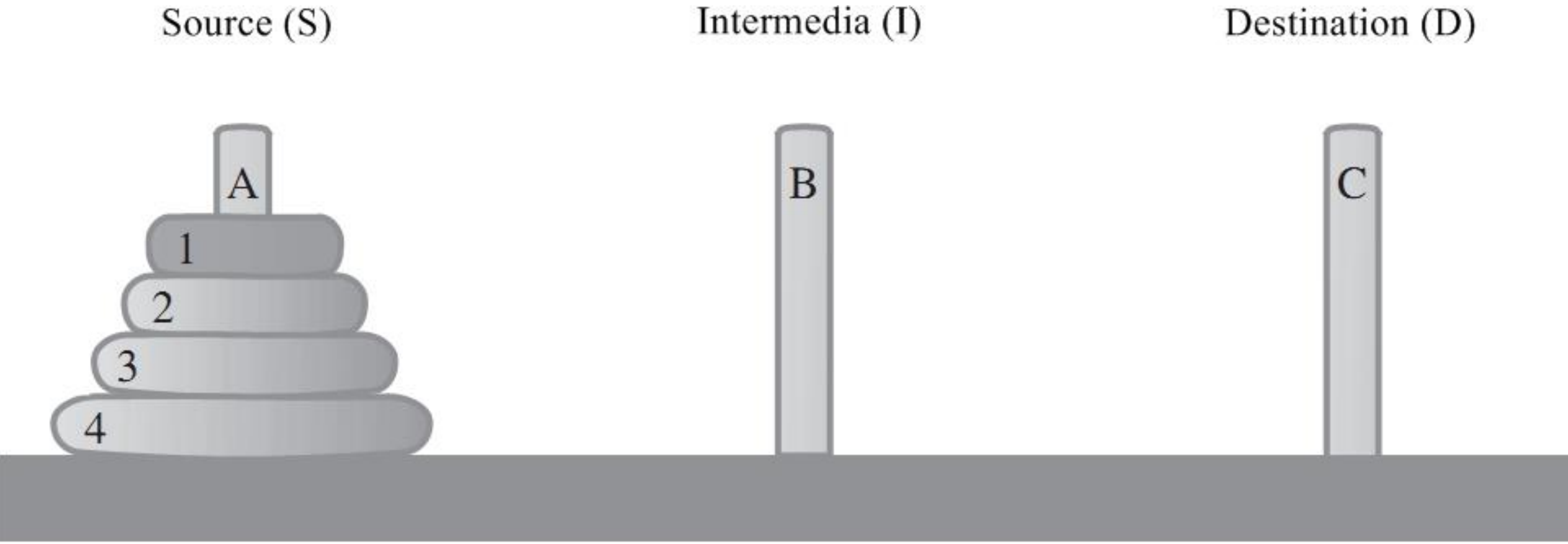
Towers of Hanoi

See Towers Workshop Applet



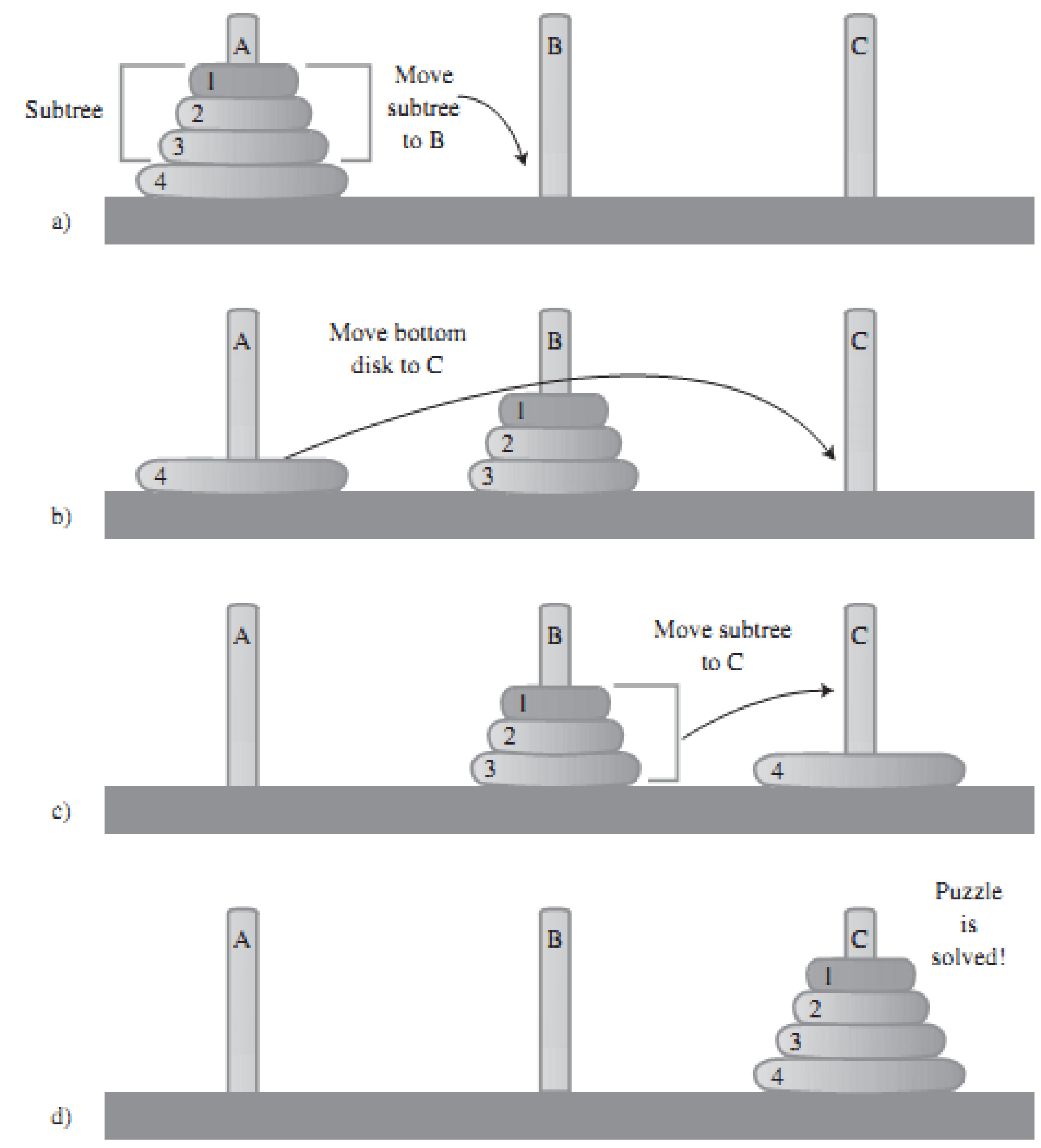
- An ancient puzzle consisting of a number of disks placed on three columns (A, B, C)
- Objectives
  - Transfer all disks from column A to column C
- Rules
  - Only one disk can be moved at a time
  - No disk can be placed on a disk that is smaller than itself

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)
A
Move bottom
disk to C
B
C
1
2
3
4
b)
A
B
Move subtree
to C
C
1
2
3
4
c)
A
B
C
Puzzle
is
solved!
1
2
3
4
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
        }
    }
```

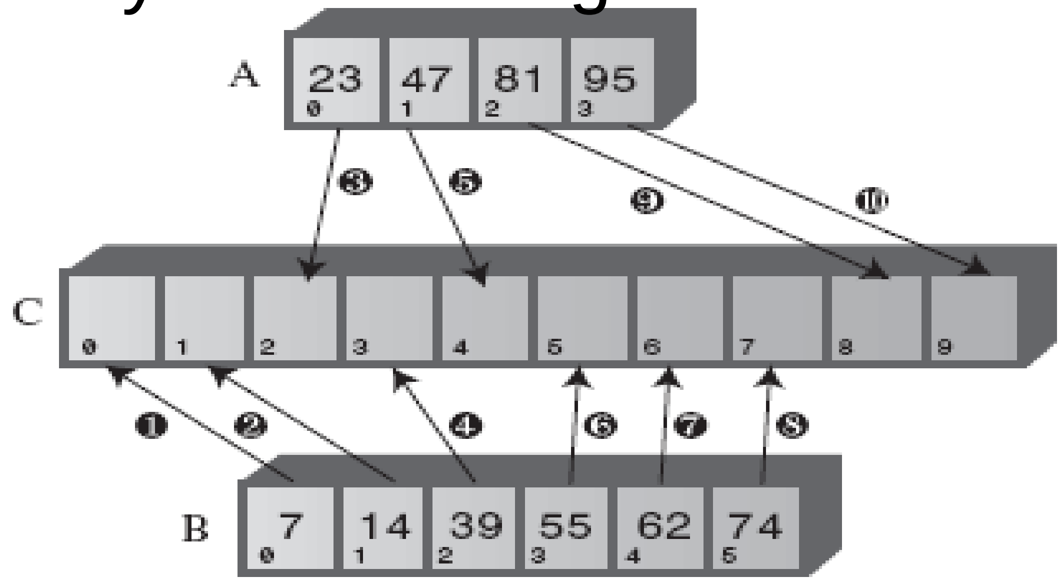
Merge Sort

Merge Sort

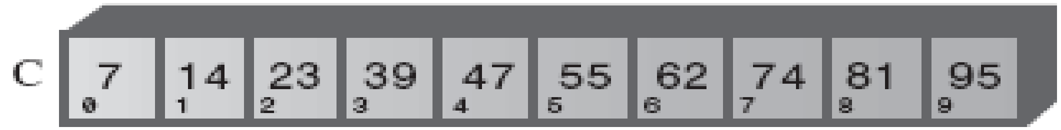
- Simple Sorting Algorithms: $O(N^2)$
  - Bubble Sort, Selection Sort, Insertion Sort
  - Using Sorted Linked List
- MergeSort: O(NlogN)
- Approach to MergeSort
  - Merging Two Sorted Arrays
  - Sorting by Merging
  - Demo via Workshop Applet
  - Efficiency of MergeSort

Merging two sorted arrays

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

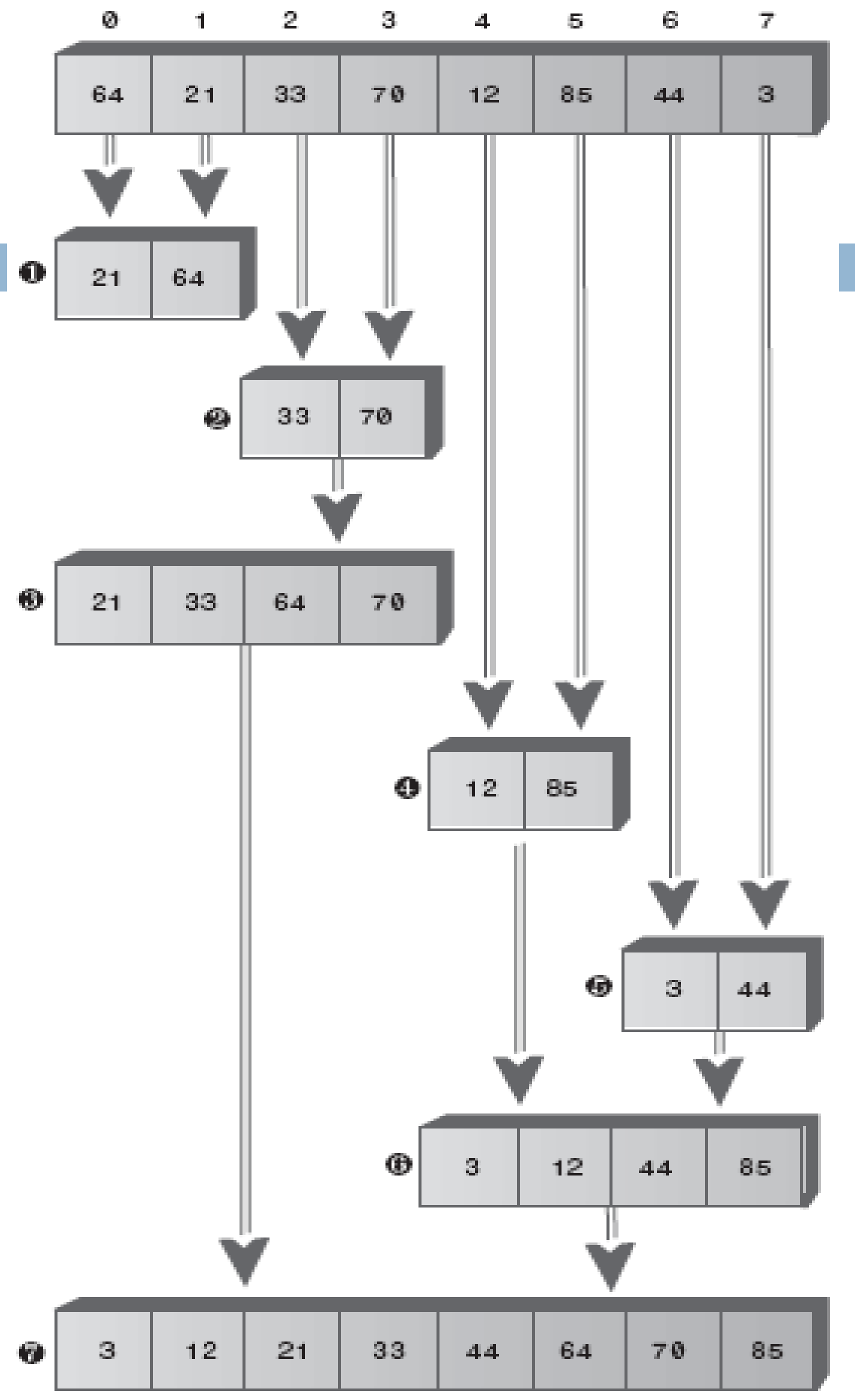


a) Before Merge



Merge Sort

- Devide 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



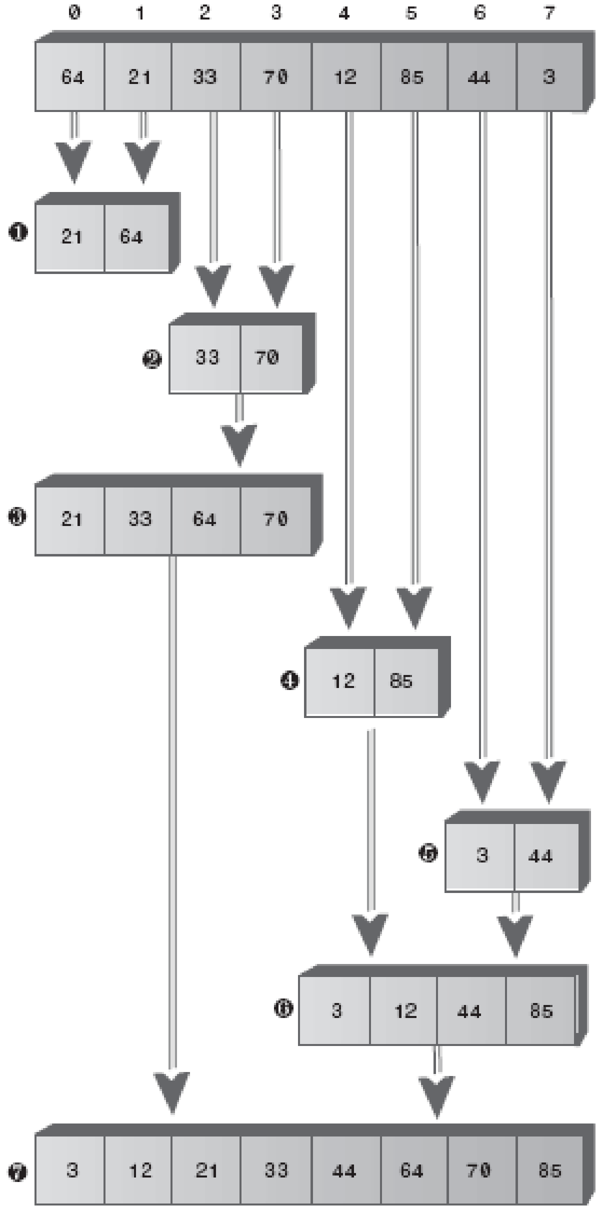
Implementation

```
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()
```

Implementation

```
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
```

Example



0 1 2 3 4 5 6 7
64 21 33 70 12 85 44 3
❶ 21 64
❷ 33 70
❸ 21 33 64 70
❹ 12 85
❺ 3 44
❻ 3 12 44 85
❼ 3 12 21 33 44 64 70 85

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)

Eliminate recursion

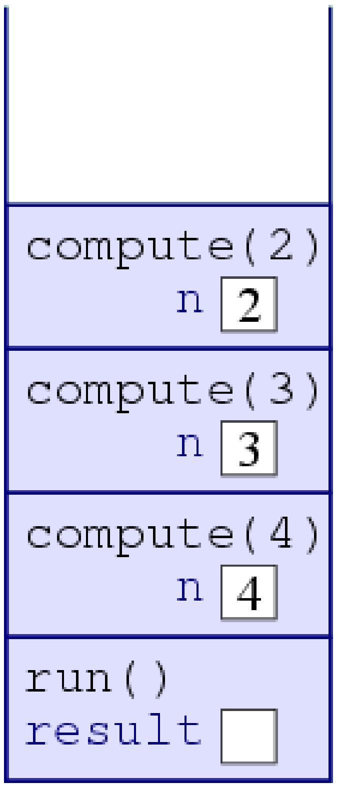
Recursion: inefficient

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

  → try to transform to non-recursive approach

Recursion & stack

- Recursion is usually implemented by stacks



Simulating a recursive method

- Read more on p.294