6-RECURSION

Topics

- Triangular Numbers
- Characteristics of Recursive Methods
- Factorials
- Binary Search
- Towers of Hanoi
- Mergesort

Triangular Numbers

Triangular Number

- Series of numbers 1, 3, 6, 10, 15, 21, …
- The nth term in the series is obtained by adding n to the previous term.

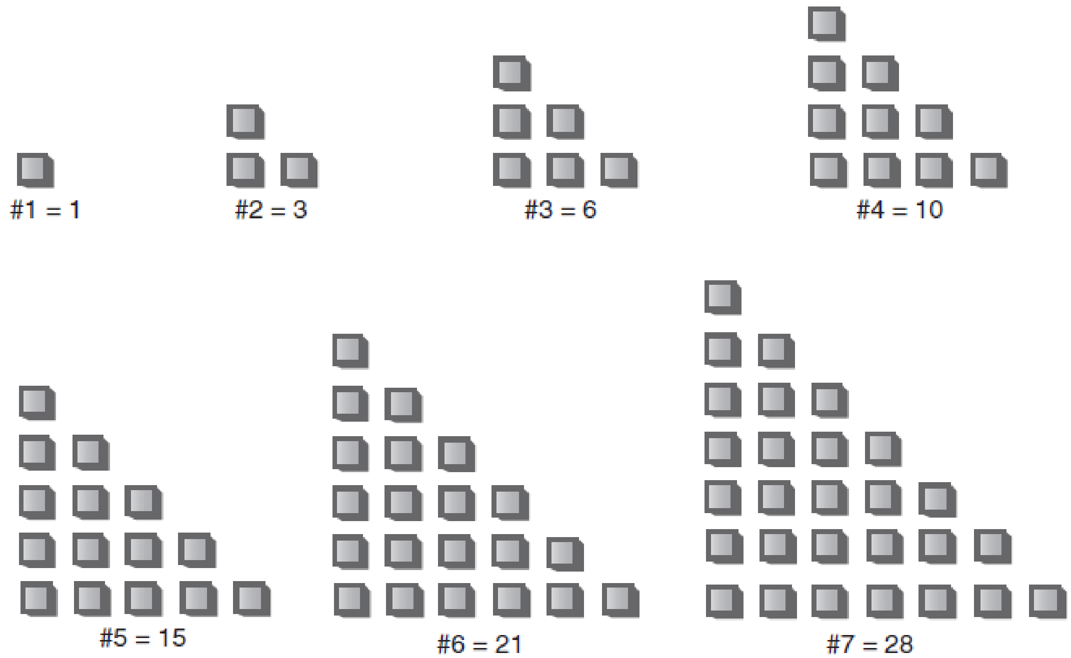


FIGURE 6.1 The triangular numbers.

Finding the 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 the nth term using recursion

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

Base case: does some work without making a recursive call

Listing 6.1, page 255

Extra work to convert the result of the recursive call into the result of *this* call

Recursive case: recurs with a simpler parameter

What's really happening?

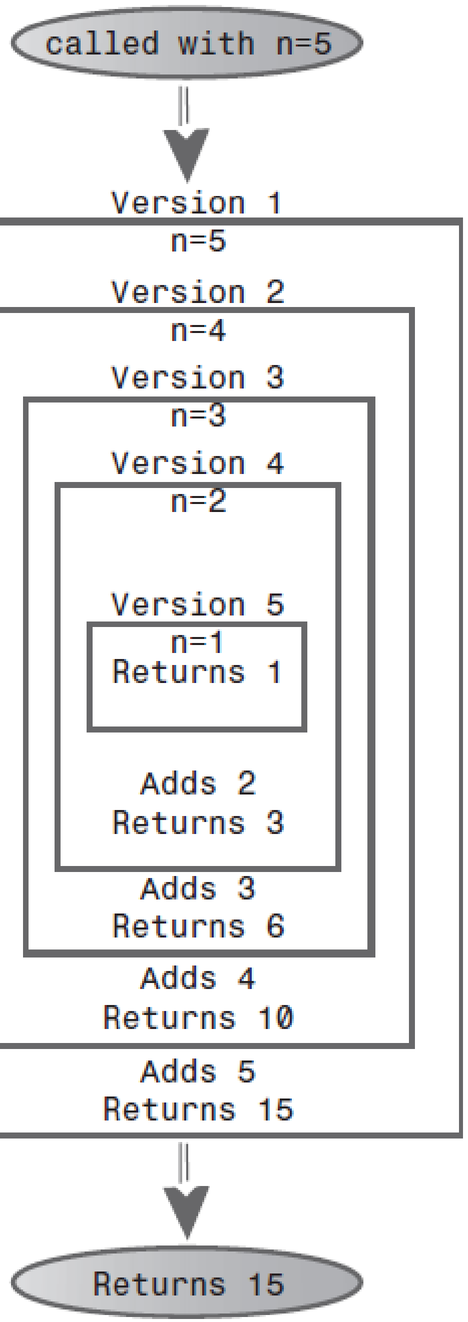
```
public static int triangle(int n)
{
  System.out.println("Entering: n=" + n);
  if(n==1)
  {
    System.out.println("Returning 1");
    return 1;
  }
  else
  {
    int temp = n + triangle(n-1);
    System.out.println("Returning " + temp);
    return temp;
  }
}
```

What’s really happening?

- Enter a number: 5 (calls: triangle(5))

- Entering: n=5
- Entering: n=4
- Entering: n=3
- Entering: n=2
- Entering: n=1
- Returning 1
- Returning 3
- Returning 6
- Returning 10
- Returning 15

- Triangle = 15


called with n=5
Version 1
n=5
Version 2
n=4
Version 3
n=3
Version 4
n=2
Version 5
n=1
Returns 1
Adds 2
Returns 3
Adds 3
Returns 6
Adds 4
Returns 10
Adds 5
Returns 15
Returns 15

Characteristics of Recursive Methods

Characteristics of Recursive Methods

- It calls itself.
- When it calls itself, it does so to solve a smaller problem. (Recursive case – may have one or more recursive case(s)).
- There's some version of the problem that is simple enough that the routine can solve it, and return, without calling itself. (Base case – may have one or more base case(s)).

Is Recursion Efficient?

- Calling a method involves certain overhead.
  - Control must be transferred from the location of the call to the beginning of the method.
  - In addition, the arguments to the method and the address to which the method should return must be pushed onto an internal stack.
- Loop approach may execute more quickly than the recursive approach.
- Recursion is usually used because it simplifies a problem conceptually, not because it's inherently more efficient.

Mathematical Induction

- Mathematical induction is a way of defining something in terms of itself. (The term is also used to describe a related approach to proving theorems.)
- Using induction, we could define the triangular numbers mathematically by saying

$$\begin{aligned} \text{tri}(n) &= 1, && \text{if } n = 1 \\ \text{tri}(n) &= n + \text{tri}(n-1), && \text{if } n > 1 \end{aligned}$$

- Defining something in terms of itself may seem circular, but in fact it's perfectly valid (provided there's a base case).

Factorials

Factorials

- Factorials are similar in concept to triangular numbers, except that multiplication is used instead of addition.
- The factorial of n is found by multiplying n by the factorial of n-1.
- That is, the factorial of 5 is 5*4*3*2*1, which equals 120.

Factorials

TABLE 6.1 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

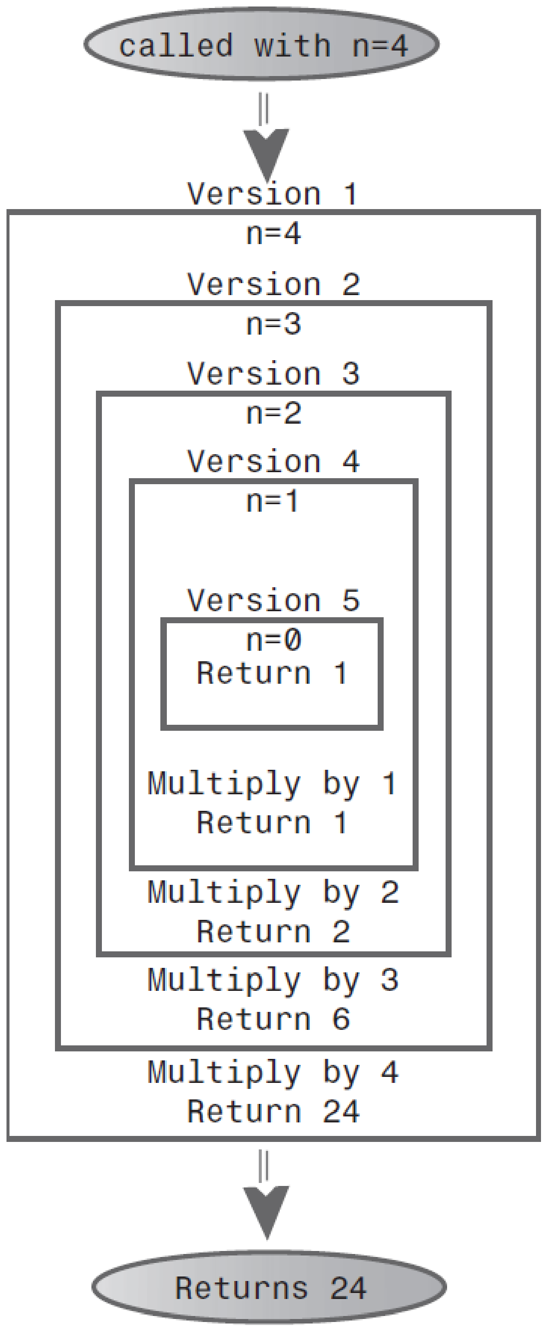
Factorials

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

Base case: does some work without making a recursive call

Recursive case: recurs with a simpler parameter

Extra work to convert the result of the recursive call into the result of *this* call


called with n=4
Version 1
n=4
Version 2
n=3
Version 3
n=2
Version 4
n=1
Version 5
n=0
Return 1
Multiply by 1
Return 1
Multiply by 2
Return 2
Multiply by 3
Return 6
Multiply by 4
Return 24
Returns 24

Binary Search

Loop-based Binary Search

```
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
      } // end else divide range
  } // end while
} // end find()
```

Recursive Binary Search

```
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()
```

Base Cases

Recursive Cases

Recursive Binary Search

- The class user, represented by main(), may not know how many items are in the array when it calls find(), and in any case shouldn't be burdened with having to know what values of upperBound and lowerBound to set initially.
- Therefore, we supply an intermediate public method, find(), which main() calls with only one argument, the value of the search key.
- The find() method supplies the proper initial values of lowerBound and upperBound (0 and nElems-1) and then calls the private, recursive method recFind(). The find() method looks like this:

```
public int find(long searchKey)
{
  return recFind(searchKey, 0, nElems-1);
}
```

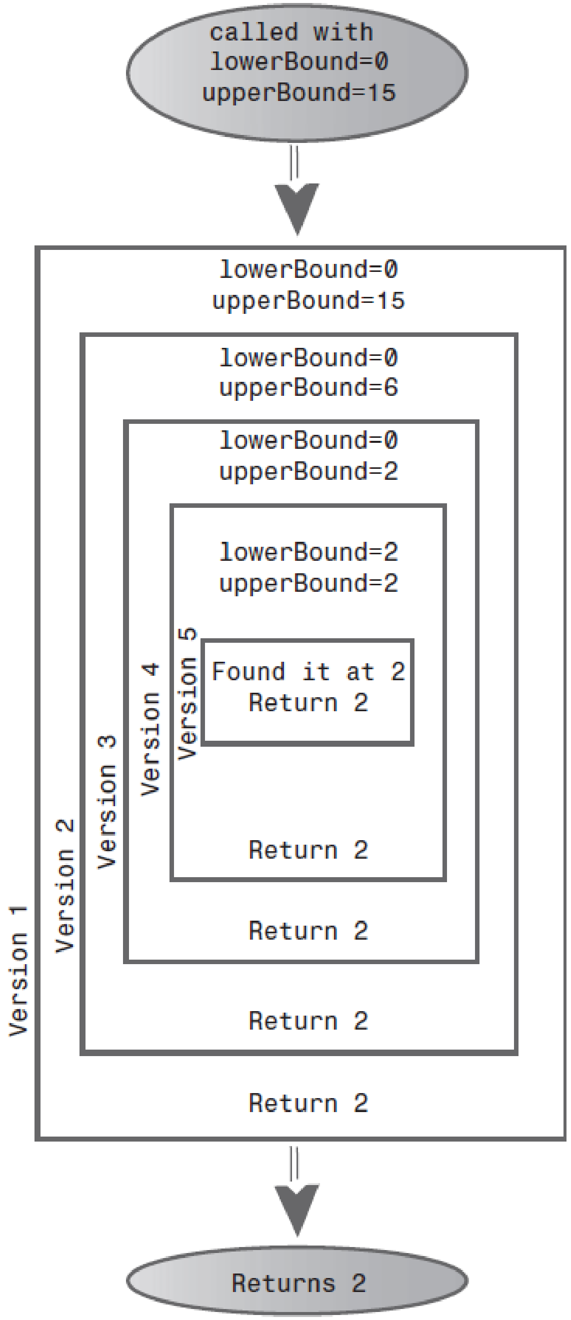


FIGURE 6.9 The recursive `binarySearch()` method.

Towers of Hanoi

The Towers of Hanoi

- The disks all have different diameters and holes in the middle so they will fit over the columns.
- All the disks start out on column A.
- The object of the puzzle is to transfer all the disks from column A to column C.
- Only one disk can be moved at a time, and no disk can be placed on a disk that's smaller than itself.
- See Workshop applet

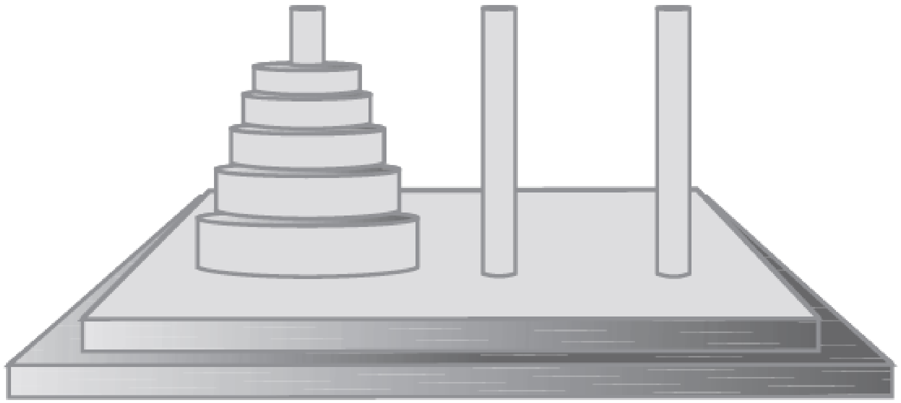


FIGURE 6.10 The Towers of Hanoi.

The Towers of Hanoi

- Moving Subtree
  - Let's call the initial tree-shaped (or pyramid-shaped) arrangement of disks on tower A a *tree*.
  - As you experiment with the applet, you'll begin to notice that smaller tree-shaped stacks of disks are generated as part of the solution process.
  - Let's call these smaller trees, containing fewer than the total number of disks, *subtrees*.

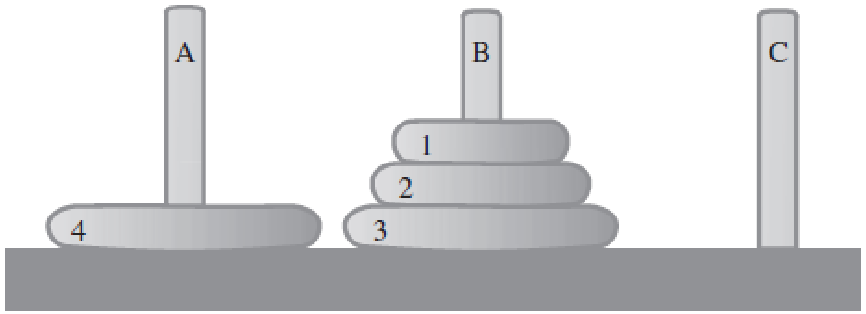


FIGURE 6.12 A subtree on tower B.

The Towers of Hanoi

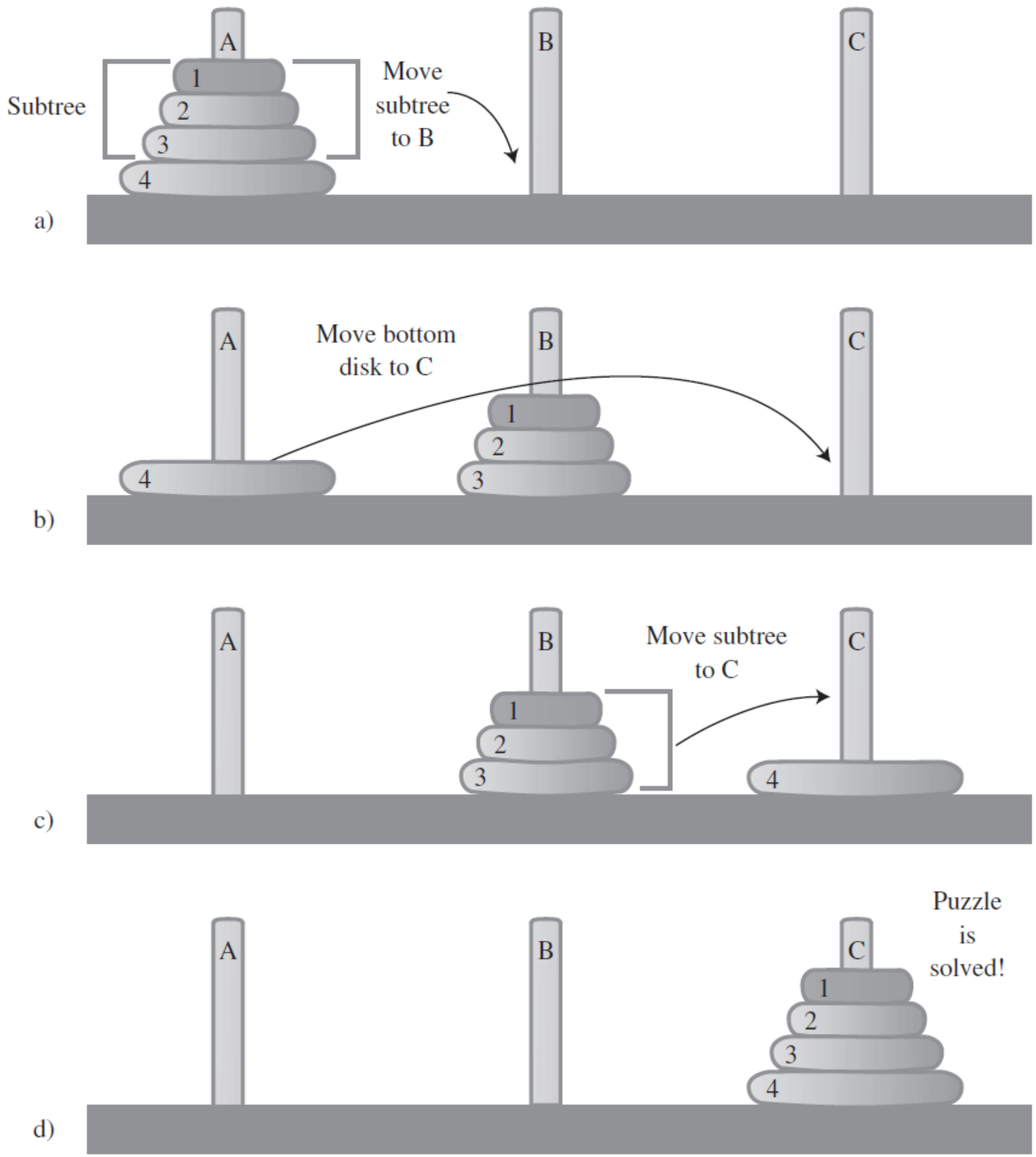
- Here's a rule of thumb:
  - If the subtree you're trying to move has an odd number of disks, start by moving the topmost disk directly to the tower where you want the subtree to go.
  - If you're trying to move a subtree with an even number of disks, start by moving the topmost disk to the intermediate tower.

The Towers of Hanoi – Recursive Algorithm

- Suppose you want to move all the disks from a source tower (call it S) to a destination tower (call it D). You have an intermediate tower available (call it I).
- Assume there are n disks on tower S. Here's the algorithm:

1. Move the subtree consisting of the top n-1 disks from S to I.
2. Move the remaining (largest) disk from S to D.
3. Move the subtree from I to D.

- When you begin, the source tower is A, the intermediate tower is B, and the destination tower is C



Subtree
Move subtree to B
a)
Move bottom disk to C
b)
Move subtree to C
c)
Puzzle is solved!
d)

Towers of Hanoi – Java code

- See Listing 6.4, Towers.java, page 278

```
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
  }
}
```

Base

Recur
sive

Towers of Hanoi – Java code

- doTowers(3, 'A', 'B', 'C');
- OutPut:
  - Disk 1 from A to C
  - Disk 2 from A to B
  - Disk 1 from C to B
  - Disk 3 from A to C
  - Disk 1 from B to A
  - Disk 2 from B to C
  - Disk 1 from A to C

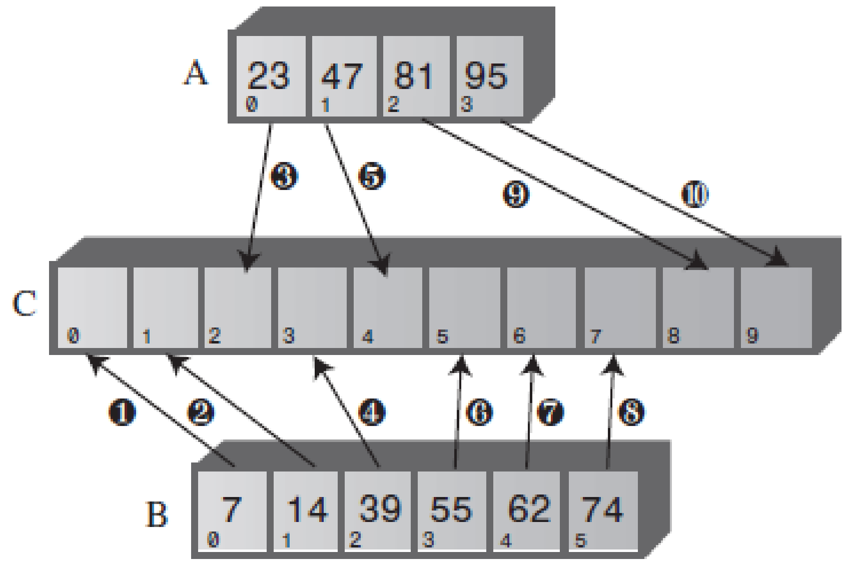
Mergesort

mergesort

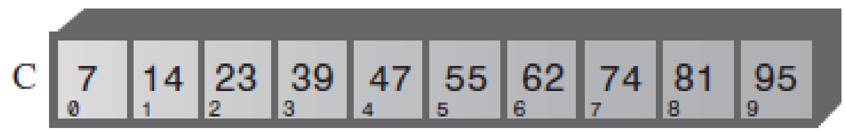
- Better complexity than Bubble, Selection and Insertion sort
- The heart of the mergesort algorithm is the merging of two already-sorted arrays.
- Merging two sorted arrays A and B creates a third array, C, that contains all the elements of A and B, also arranged in sorted order.

- We'll examine the merging process first; later we'll see how it's used in sorting.
  - Imagine two sorted arrays. They don't need to be the same size.
    - Let's say array A has 4 elements and array B has 6.
  - They will be merged into an array C that starts with 10 empty cells.

mergesort



a) Before Merge



b) After Merge

mergesort

TABLE 6.3 Merging Operations

Step	Comparison (If Any)	Copy
1	Compare 23 and 7	Copy 7 from B to C
2	Compare 23 and 14	Copy 14 from B to C
3	Compare 23 and 39	Copy 23 from A to C
4	Compare 39 and 47	Copy 39 from B to C
5	Compare 55 and 47	Copy 47 from A to C
6	Compare 55 and 81	Copy 55 from B to C
7	Compare 62 and 81	Copy 62 from B to C
8	Compare 74 and 81	Copy 74 from B to C
9		Copy 81 from A to C
10		Copy 95 from A to C

Merge – Java code

- Listing 6.5, merge.java, page 281

```
// merge A and B into C
public static void merge(
                    int[] arrayA, int sizeA,
                    int[] arrayB, int sizeB,
                    int[] arrayC )
{
    int aDex=0, bDex=0, cDex=0;
   while(aDex < sizeA && bDex < sizeB)
   // neither array empty
      if( arrayA[aDex] < arrayB[bDex] )
             arrayC[cDex++]
                    = arrayA[aDex++];
      else
             arrayC[cDex++]
                    = arrayB[bDex++];
```

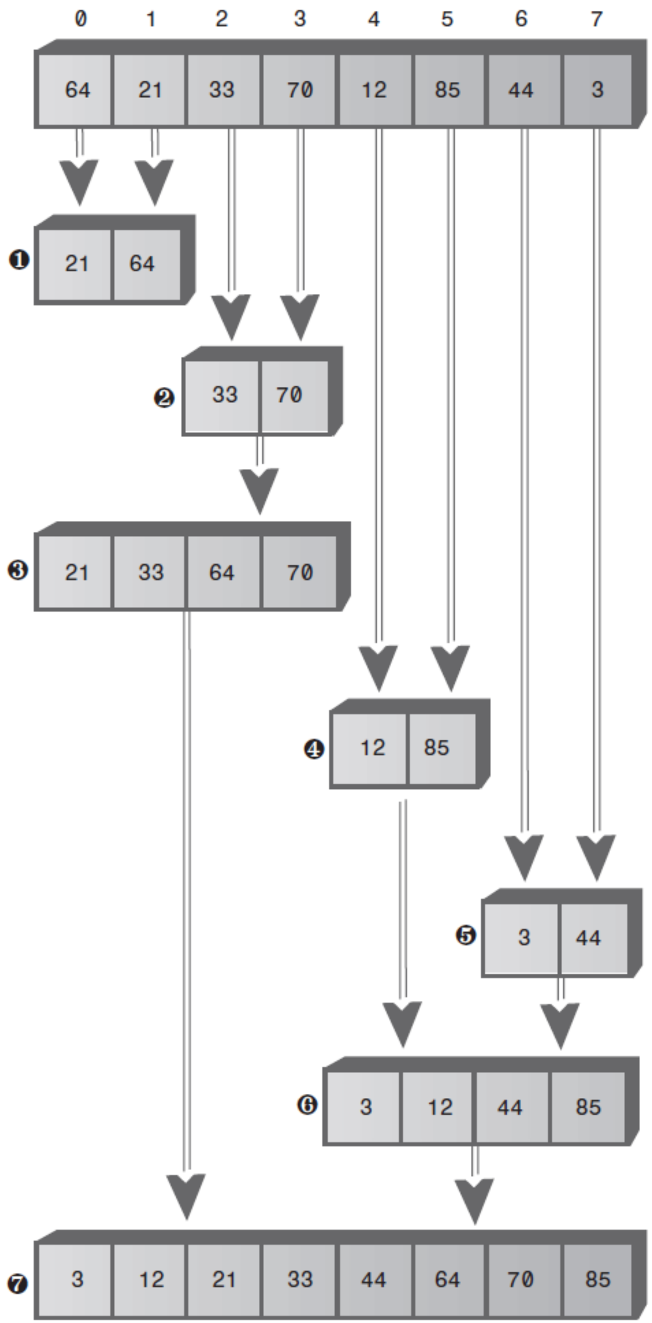
```
// arrayB is empty, but arrayA isn't
while(aDex < sizeA)
    arrayC[cDex++] = arrayA[aDex++];

// arrayA is empty, but arrayB isn't
while(bDex < sizeB)
    arrayC[cDex++] = arrayB[bDex++];

} // end merge()
```

Sorting by merging

- The idea in the mergesort is to divide an array in half, sort each half, and then use the merge() method to merge the two halves into a single sorted array.
- How do you sort each half?
  - You divide the half into two quarters, sort each of the quarters, and merge them to make a sorted half.
  - Similarly, each pair of 8ths is merged to make a sorted quarter, each pair of 16ths is merged to make a sorted 8th, and so on.
- You divide the array again and again until you reach a subarray with only one element.
  - This is the base case; it's assumed an array with one element is already sorted.

Sorting by merging

- In mergeSort() the range is divided in half each time this method calls itself, and each time it returns it merges two smaller ranges into a larger one.
  - As mergeSort() returns from finding two arrays of one element each, it merges them into a sorted array of two elements.
  - Each pair of resulting 2-element arrays is then merged into a 4-element array.
  - This process continues with larger and larger arrays until the entire array is sorted.
- This is easiest to see when the original array size is a power of 2.
- (See MergeSort workshop applet)


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

mergesort – Java code

```
// Full LISTING 6.6, mergeSort.java, page 288
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
```

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

mergesort efficeincy

- **O (N log N)**
- How do we know this?

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)

The End