# ZAMAN UNIVERSITY



Data Structures and Algorithms

## Chapter 3

# Recursion and Quicksort

# Outline



- Recursion
- Applied Recursion
- Quicksort
- Improving Quicksort

# Outline



- Recursion
- Applied Recursion
- Quicksort
- Improving Quicksort

# Recursion

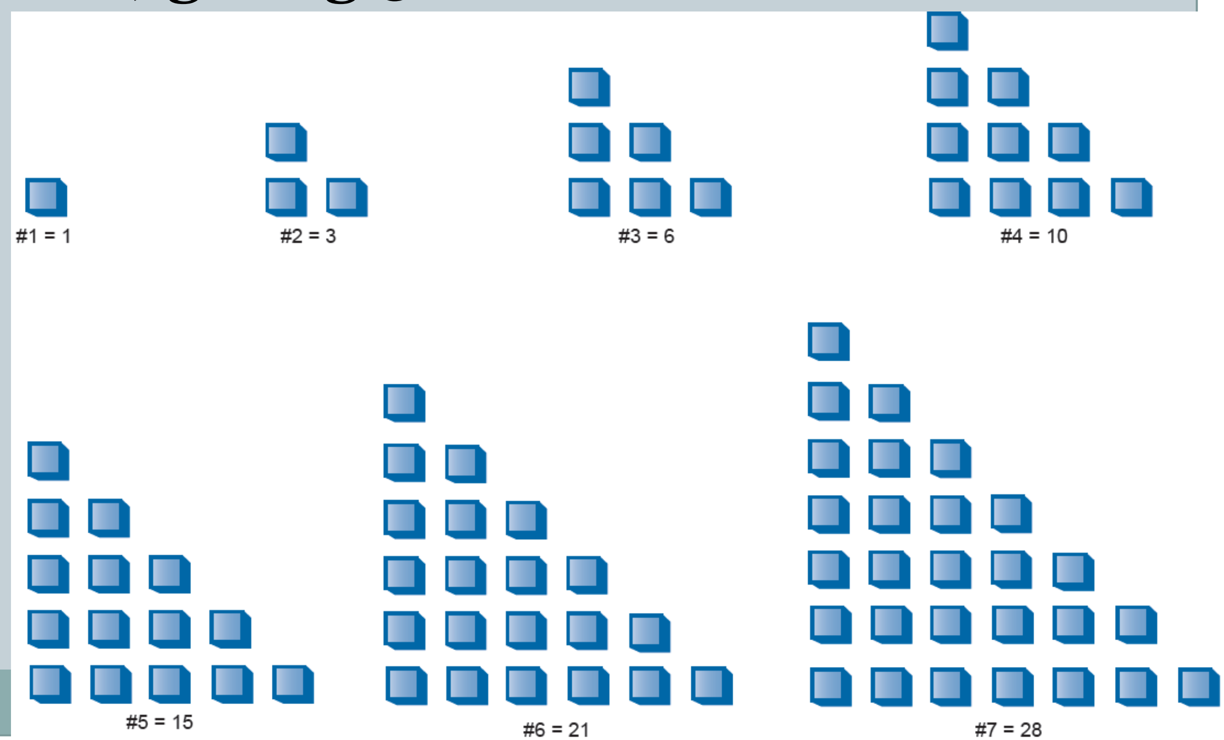


- Recursion is one of the most interesting, and surprisingly effective, techniques in programming
- Recursion is a programming technique in which a function calls *itself*

# Recursion: Demonstrating Recursion with Triangular Numbers



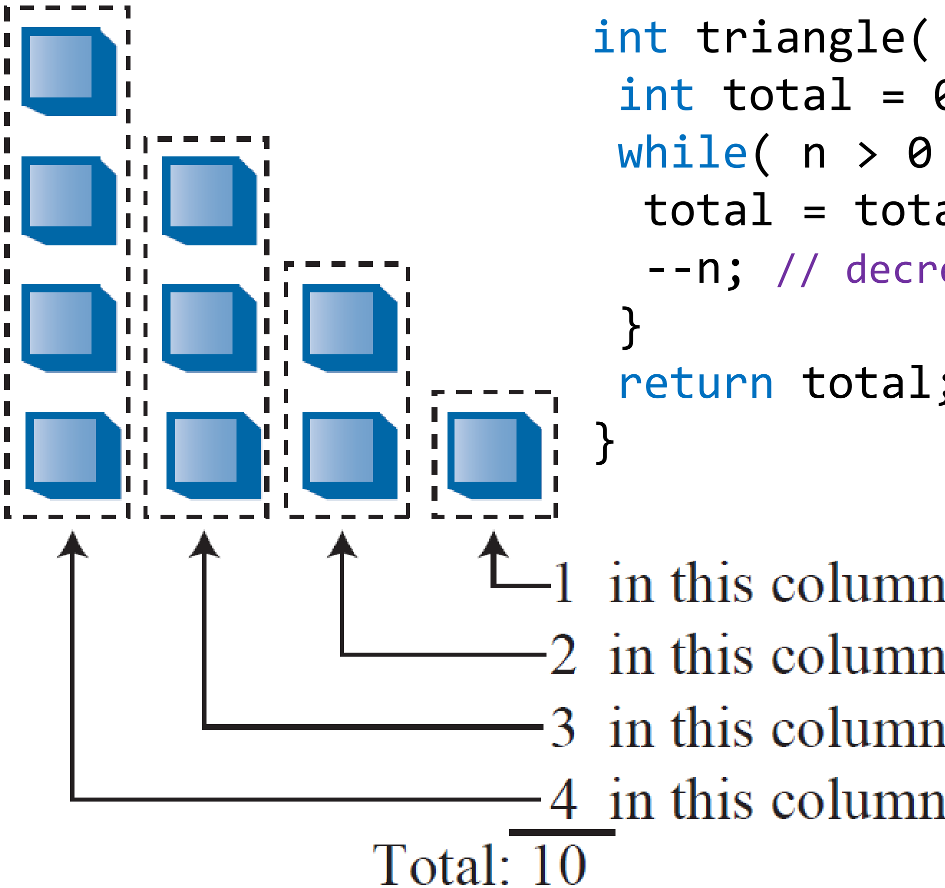
- The $n^{th}$ term in the series is obtained by adding $n$ to the previous term
- Thus the second term is found by adding 2 to the first term (which is 1), giving 3.
- The third term is 3 added to the second term giving 6, and so on.



# Recursion: Finding the $n^{th}$ Term Using a Loop



- In the $4^{th}$ term, the first column has four little squares, the second column has three, and so on  Adding `4+3+2+1`  gives `10`.

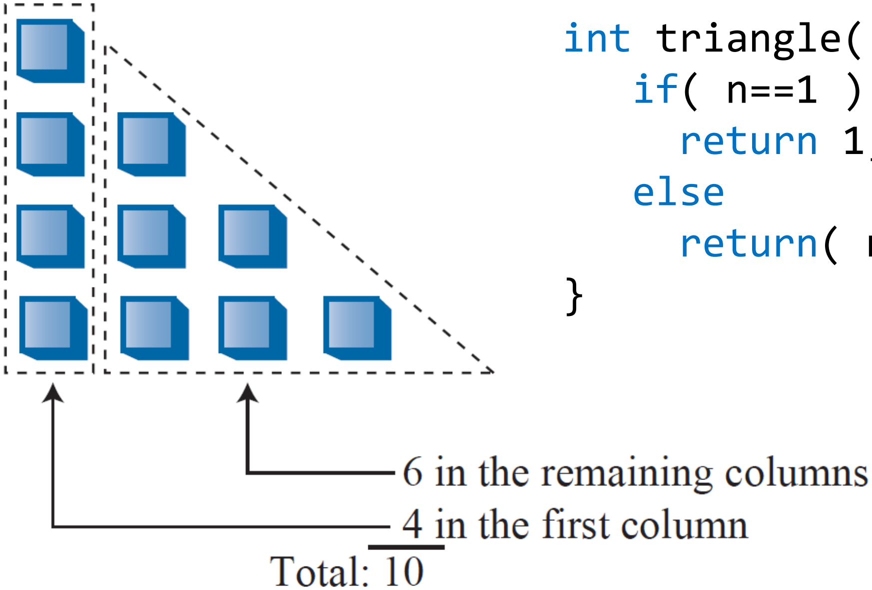



```
int triangle( int n ) {
 int total = 0;
 while( n > 0 ) { //until n is 1
  total = total + n; //add n (column height) to total
  --n; // decrement column height
 }
 return total;
}
```

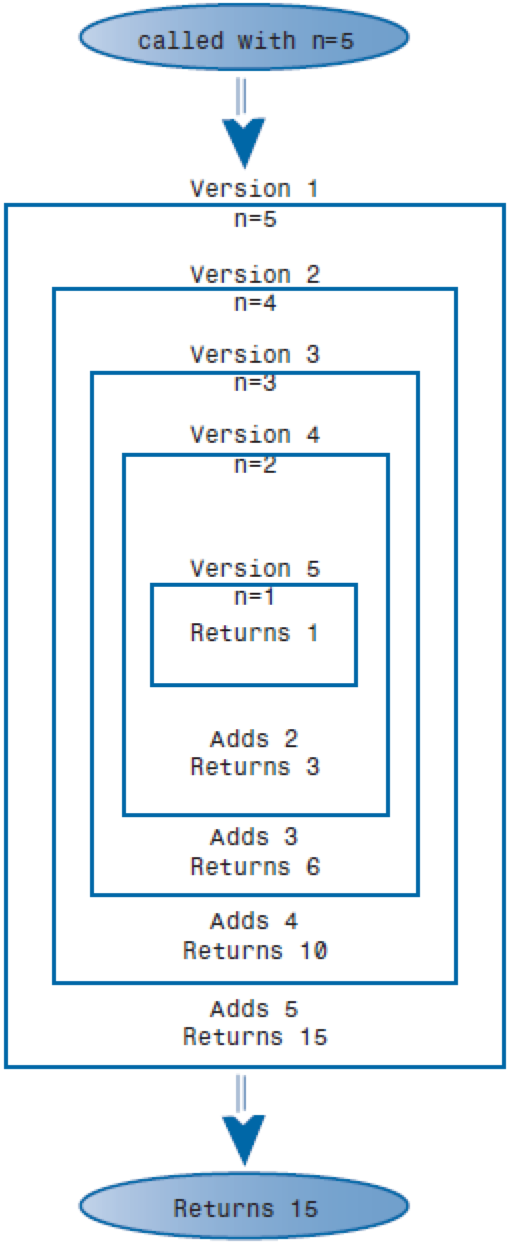
# Recursion: Finding the $n^{th}$ Term Using Recursion



- The **loop** approach might seem straightforward, but there's another way to look at this problem
- The value of the $n^{th}$ term can be thought of as the sum of only two things, instead of a whole series. These are:
  - The first (tallest) column, which has the value *n*
  - The *sum* of all the remaining columns




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

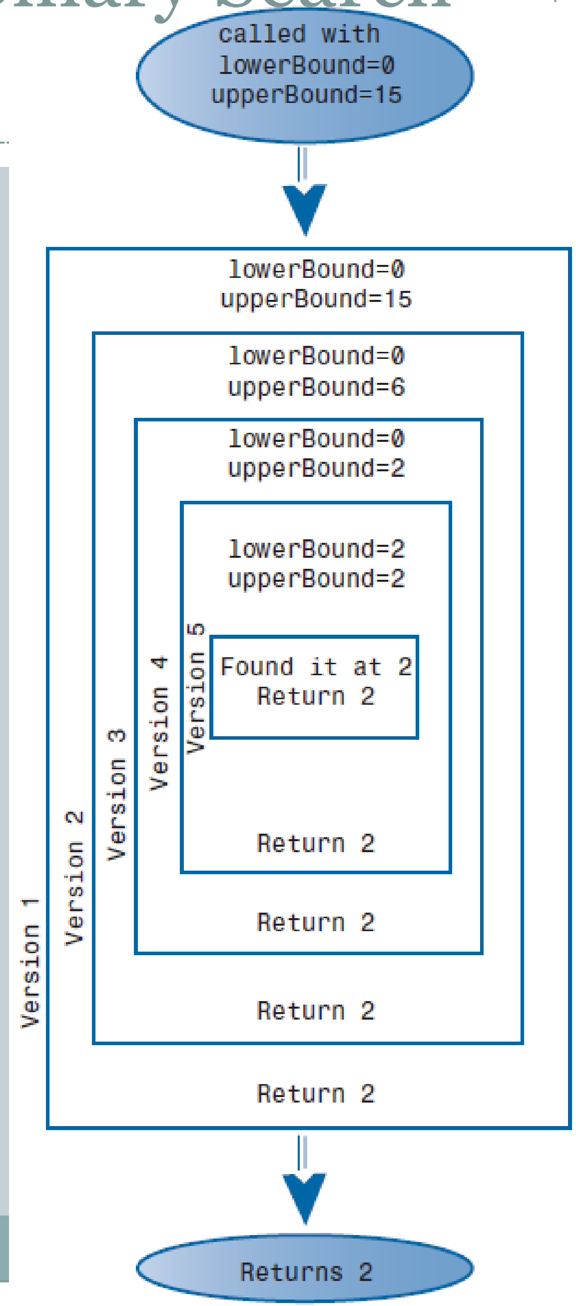


# Recursion: Using Recursion in a Binary Search



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

10, 23, 27, 34,…,99
Key = 27

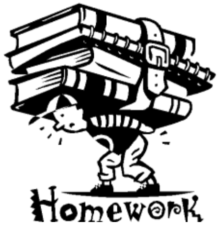


## Recursion: Understanding Divide-and-Conquer Algorithms



- The recursive binary search is an example of the ***divide-and-conquer*** approach
- The big problem divided into two smaller problems and solve each one separately
- The process continues until you get to the base case, which can be solved easily, with no further division into halves

- A ***divide-and-conquer*** approach usually involves a function that contains two recursive calls to itself, one for each half of the problem
- In the binary search, there are two such calls, but only one of them is actually executed.







Write a program that uses recursion:

1. for Binary Search;
2. to calculate the factorial of a number.

# Outline



- Recursion
- Applied Recursion
- Quicksort
- Improving Quicksort

# Applied Recursion: Mergesort

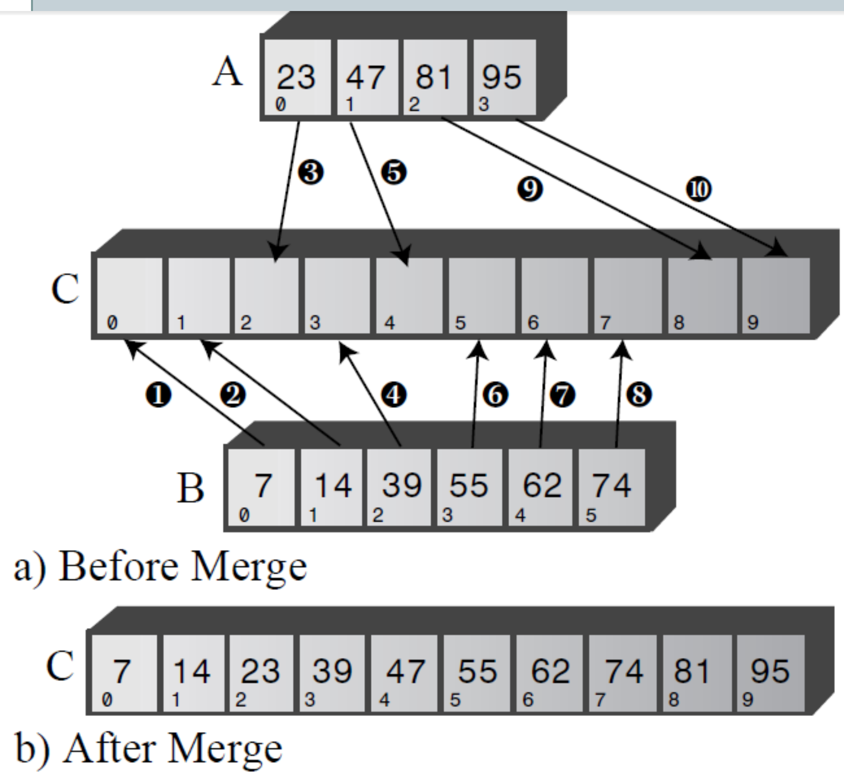


- Mergesort is more efficient sorting technique than Bubble and Insertion sort, in term of time
- Bubble and Insertion take `O(N²)` time, the mergesort is `O(N*logN)`
- For example: if `N` is `10,000`, then $N^2$ is `100,000,000`, where as `N*logN` is `40,000`. Thus, if sorting this many items required `40` Seconds with the mergesort, it would take almost `28` hours for the insertion sort
- The mergesort is also fairly easy to implement, it's conceptually easier than *quicksort*

# Merge Two Sorted Arrays



- Merging of two sorted arrays A and B to C, that contains all the elements of A and B, also arranged in sorted order




| Step | Comparison | 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 |

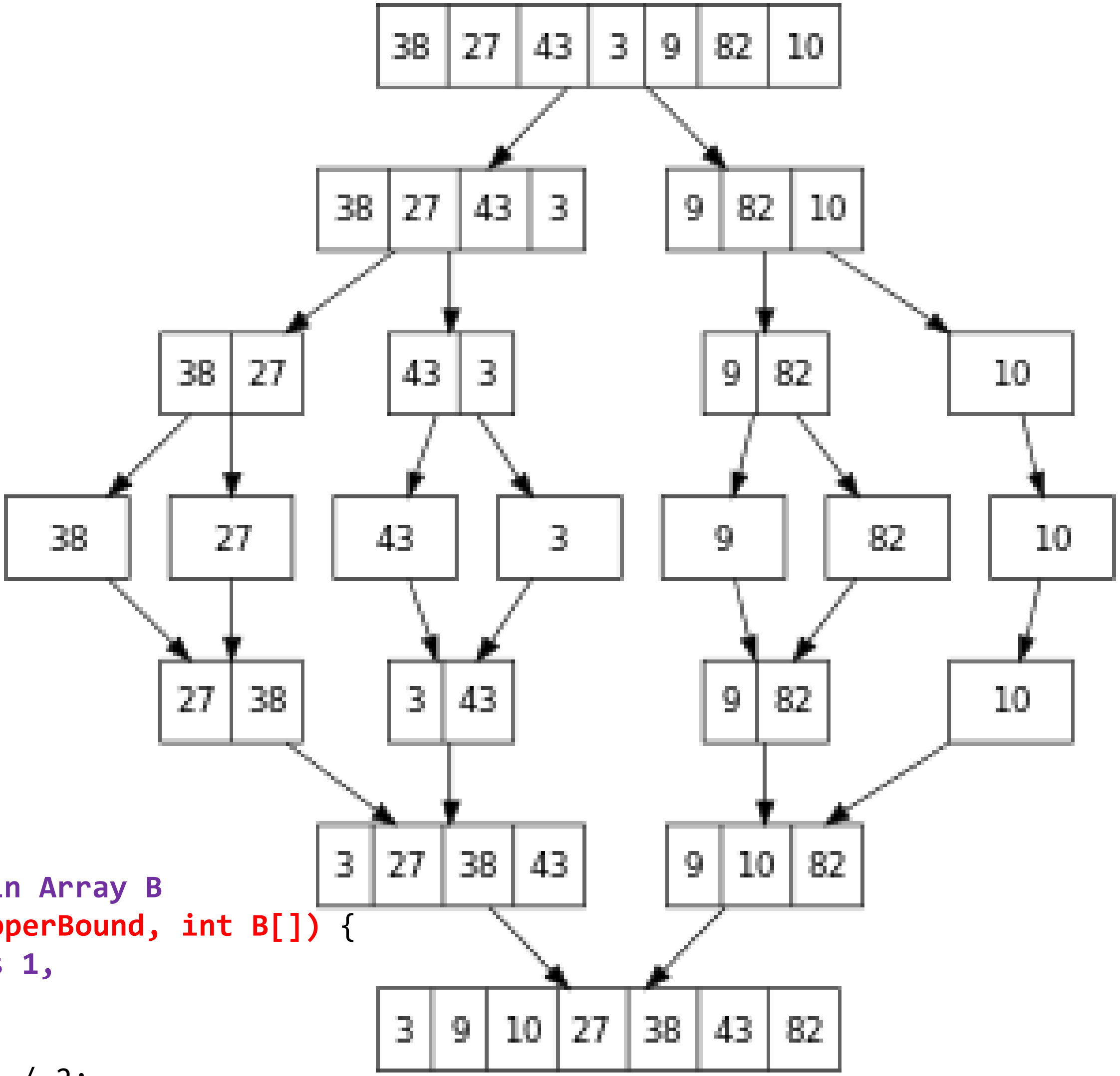
# Recursion: Sorting by Merging



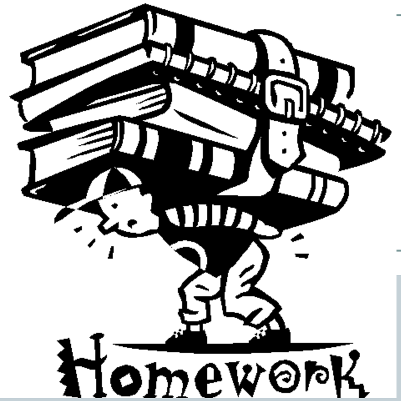
- Mergesort is to divide an array in half, sort each half, and then merge the two halves into a single sorted array

- How do you sort each half?

- Half array divide into two quarters, sort each of the quarters, and merge them to make a sorted half
- You divide the array again and again until you reach a subarray with only one element. The one element is already sorted.




```
//Result of Sorted Array A will be written in Array B
RecMergeSort(int A[], int lowerBound, int upperBound, int B[]) {
  if(lowerBound == upperBound) //if range is 1,
           return; //no use sorting
  else { //find midpoint
           int mid = (lowerBound+upperBound) / 2;
           //sort low half
           RecMergeSort(A, lowerBound, mid, B);
           //sort high half
           RecMergeSort(A, mid+1, upperBound, B);
           //merge lowerBound->mid and mid+1->upperBound to B
           merge(A, lowerBound, mid+1, upperBound, B);
           } //end else
} //end recMergeSort()
```




Write a program:

1. Merge of two sorted Arrays;
2. Use recursion to create mergesort function.

# Outline



- Recursion
- Applied Recursion
- Quicksort
- Improving Quicksort

# Quicksort



- The bubble and insertion sorts — are easy to implement but are rather slow
- Mergesort is applied recursion, it runs much faster than the simple sorts, but requires twice space as original array
- Quicksort runs faster than simple sorts, in O(N*logN) time, it does not require a large amount of extra memory space, as mergesort
- Quicksort is based on the idea of partitions

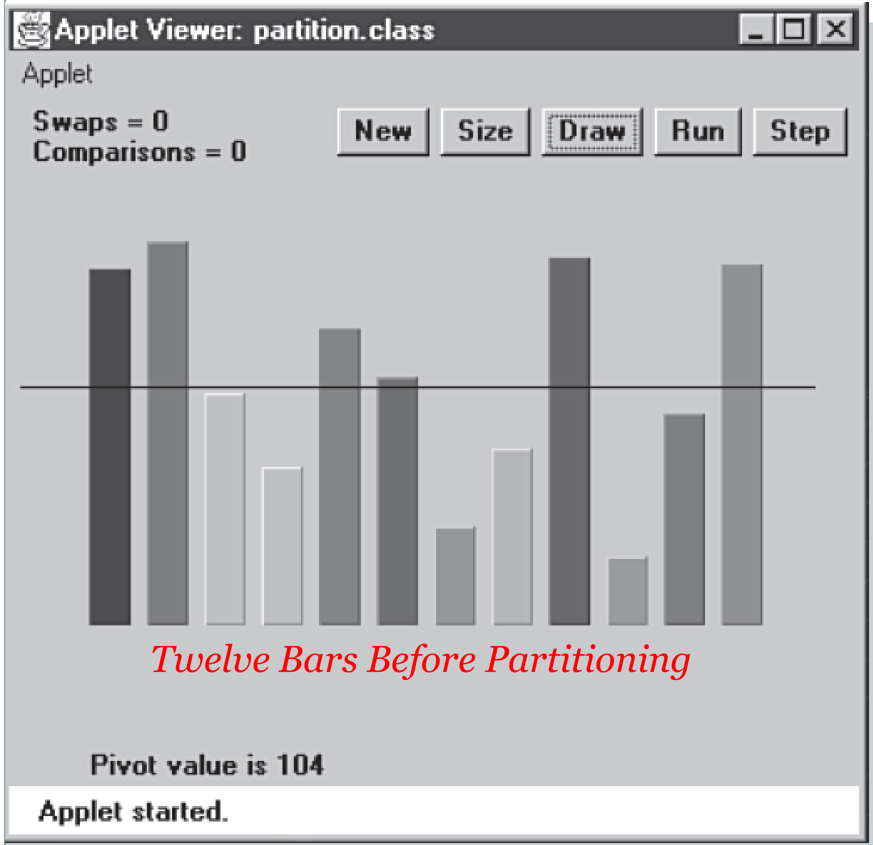
# Quicksort: Partitioning



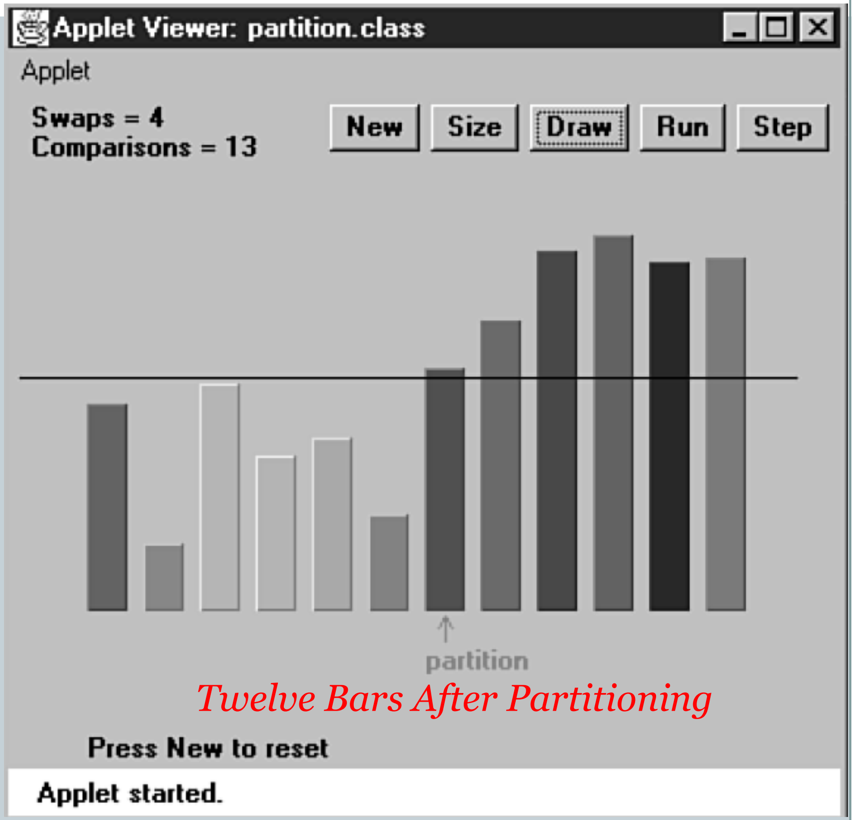
- To *partition* data is to divide it into two groups:
  - all the items with a key value higher than a specified amount;
  - All the time with a lower key value.
- Examples
  - Maybe you want to divide your personnel records into two groups: employees who live within 15 miles of the office and those who live farther away
  - A school administrator might want to divide students into those with grade point averages higher and lower than 3.5, so as to know who deserves to be on the dean's list

# Quicksort: Partitioning Example






*Twelve Bars Before Partitioning*




*Twelve Bars After Partitioning*

- Pivot – is the value used to determine into two groups (less and greater). It is the border of less than and greater.

# Quicksort: Partitioning Pseudo Code



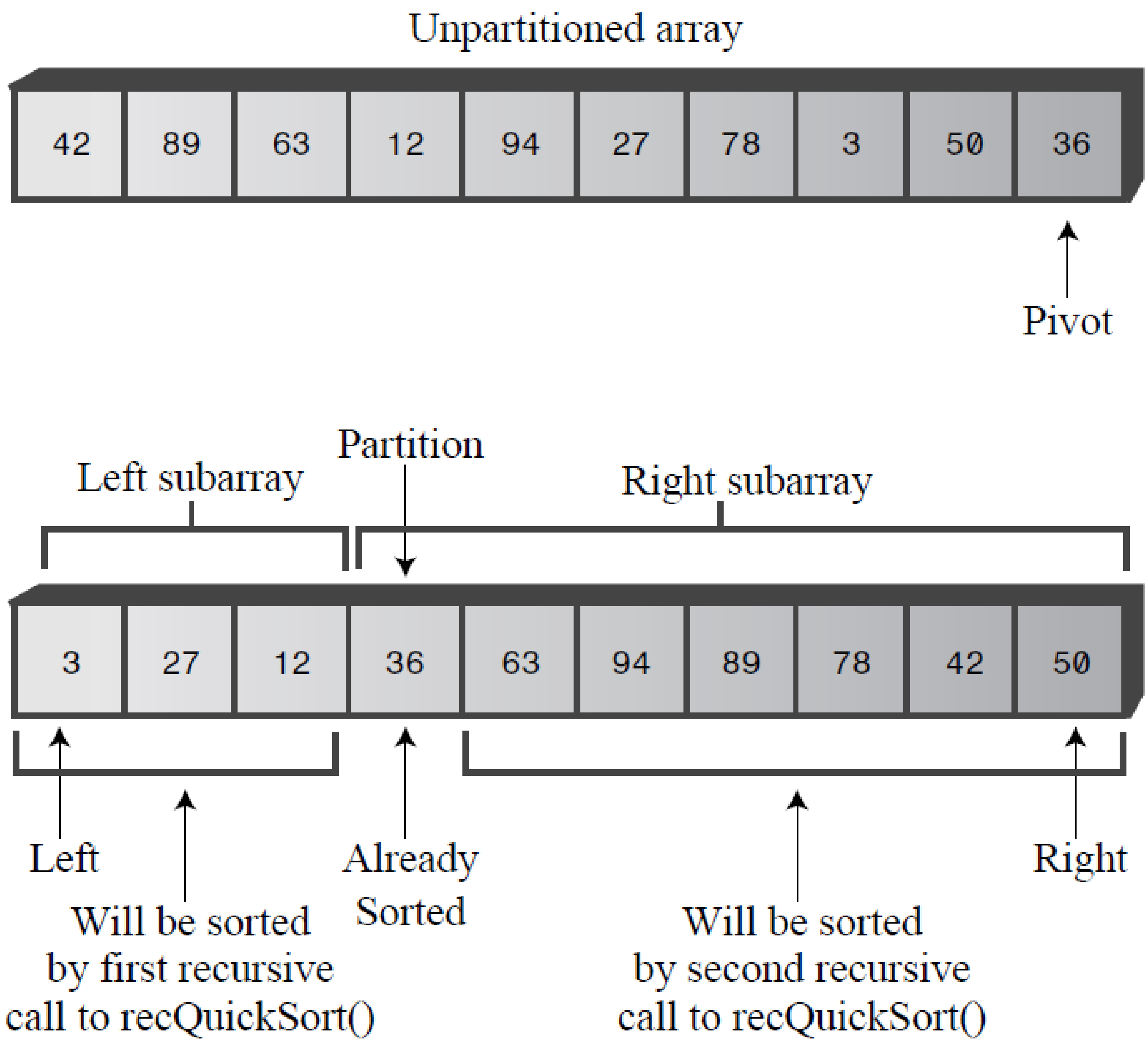
```
int PartitionIt( left, right, pivot) {
  while( true ){
    find the element greater than pivot; //find to the right, but
                                          //possible greater than right element

    find the element smaller than pivot; //find to the left, but possible
                                          //less than the left element

    if the index of left cross to right //partition done
      then partition done (break);
    else swap( LeftMark, RightMark );
  }
  return LeftMark;
}
```

# Basic Quicksort



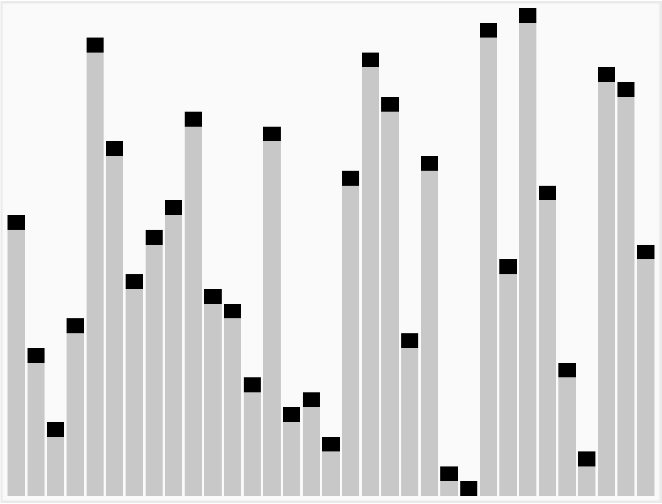
- Quicksort was discovered by British Computer Scientist C. A. R. Hoare, 1962
- Basically the quicksort algorithm operates by partitioning an array into two sub-arrays, and then calling itself recursively to quicksort each of these sub-arrays
- The pivot will be selected at right, but it will be finally placed between these two sub-arrays

# Quicksort: Recursive Calls Sort Subarrays



# Quicksort: Pseudo Code

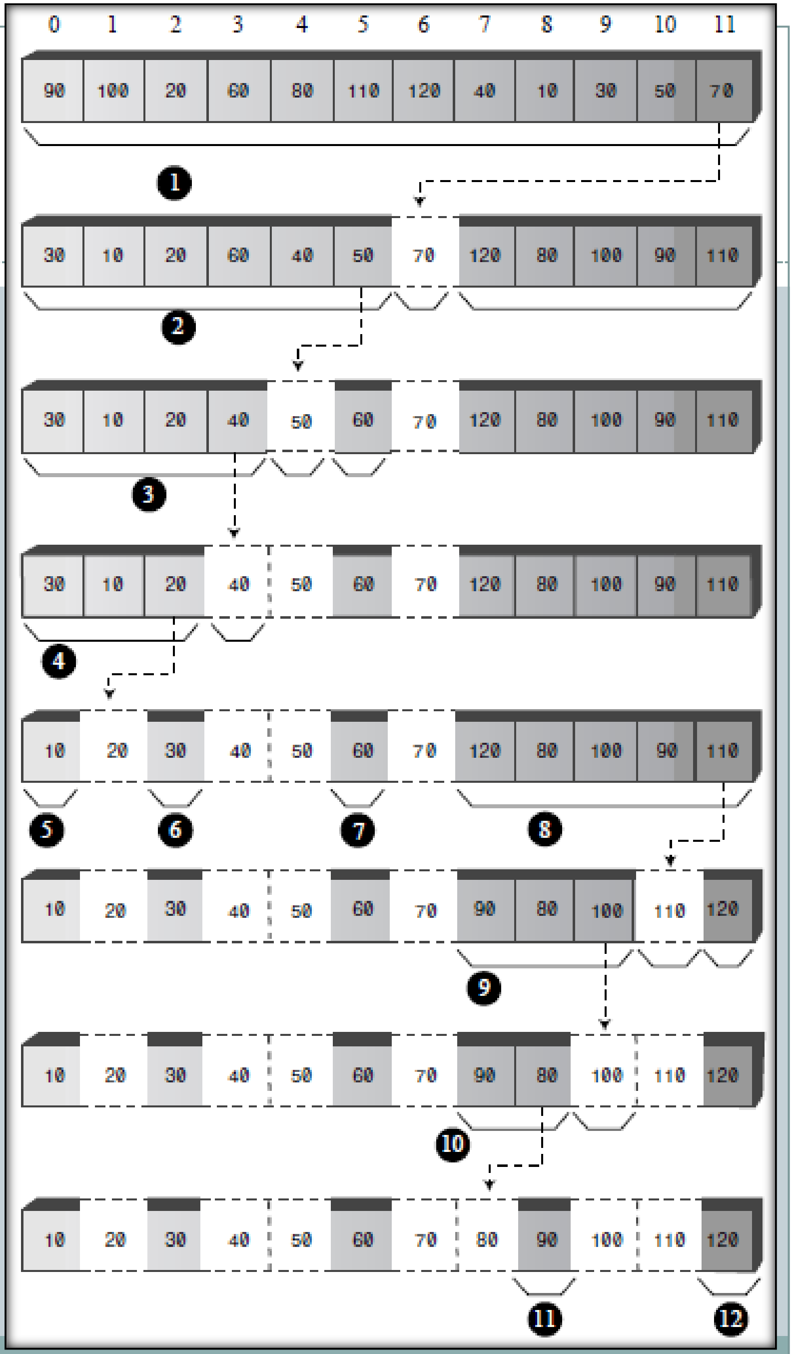


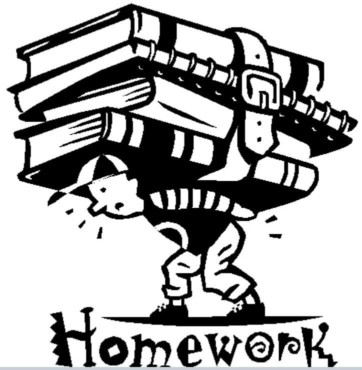


```
recQuickSort(left, right) {
  if Array has only element
      return; //already sorted
  else { //size is 2 or larger
    pivot is the right element of Array //rightmost item
   //partition range
   partition <- PartitionIt(left, right, pivot);
   recQuickSort(left, partition-1); //sort left side
   recQuickSort(partition+1, right); //sort right side
  }
} // end recQuickSort()
```

# Quicksort Process








Write a program:

1. to partition Array;
2. Use recursion to create Quicksort function.

# Outline



- Recursion
- Applied Recursion
- Quicksort
- Improving Quicksort

# Problems with Inversely Sorted Data



- If we use quicksort to sort 100 inversely sorted items, the algorithm runs much more slowly
- During partitioning, pivot will be larger then sub-arrays

- Pivot should be larger than half array, and smaller than half array
- Two equal sub-arrays is the optimum situation for the quicksort algorithm
- The worst situation results when a subarray with N elements is divided into one subarray

# How to Select Pivot?

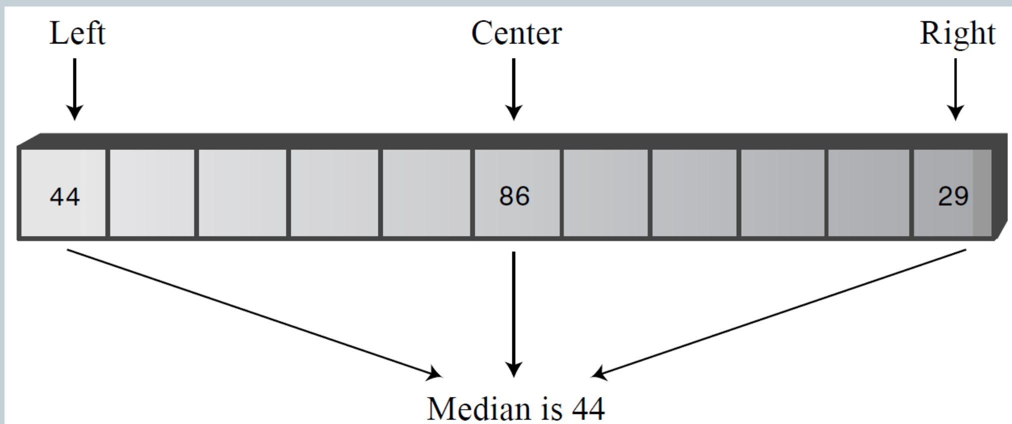


- The pivot should be the median of the items being sorted
- The *median* or *middle* item is the data item chosen, it will be divided half smaller and half larger
- Choosing pivot at random is simple but it does NOT always result in a good selection
- The method should be simple but have a good chance of avoiding the largest or smallest value
- Thus, the pivot should be selected by ***Median-of-Three***

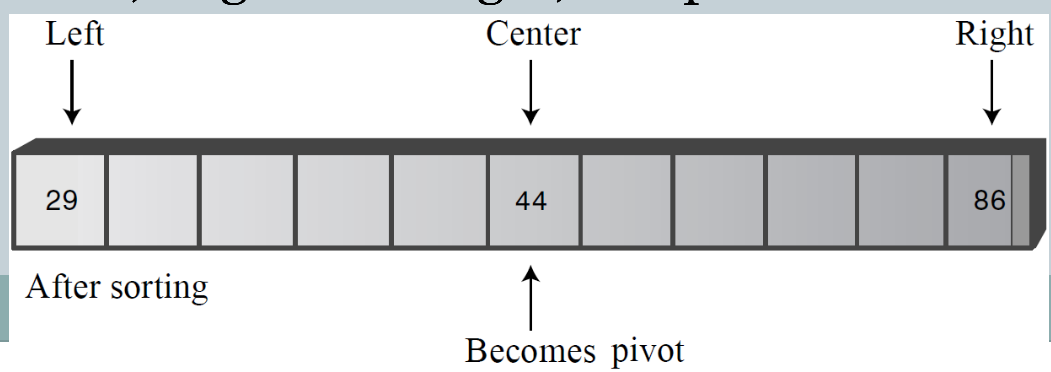
# Median-of-Three Partitioning



- A compromise solution is to find the median of the *first*, *last*, and *middle* elements of the array, and use this for the pivot
- This is called the *median-of-three*




- During finding median, the three element should be sorted (which the smallest is in left, largest is in right, and pivot is in center)



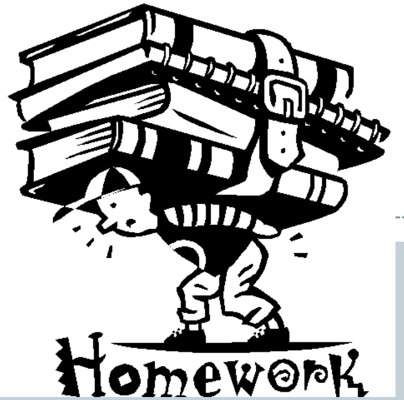
# Handling Small Partitions



- If you use the median-of-three partitioning scheme, it will not work for partitions of three or fewer items
- For example, after partitioning we have sub-arrays of 2 and 3 items
- In this case, for the sub-arrays use manual sorting

- Knuth* recommends using insertion sort, in case if number item of array is 9

** - The Art of Computer Programming by Donald E. Knuth, of Stanford University (Addison Wesley, 1997)*




Write a QuickSort program which is:

1. pivot is selected by median-off-three;
2. and using **insertion sort** for sub-arrays, which is number items 2 or 3.

End of Chapter 3