Introduction to Algorithm

- Algorithms are a set of rules used for solving a set of problems.

- An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces value, or set of values as output (solution to given problem).

- Algorithms are like road maps for accomplishing a given, well-defined task.

Introduction to Algorithm

- An algorithm is a finite sequence of step by step, discrete, unambiguous instructions for solving a particular problem

  -Acts on input data,
  - Expected to produce output data.

  - Each instruction can be carried out in a finite amount of time in a deterministic way

Algorithm Description

- Understand the problem before solving it

  -Identify & name each Input (**Given/Provided)**
  -Identify & name each Output (**Results)**
  -Assign a name to our algorithm (**Name**)
  -Combine the previous 3 pieces of information into a formal statement (**Definition**)

Method

- Once we have prepared the Algorithm Description, we need to solve the problem

- We develop a series of instructions (limited to those described previously) that, when executed, will compute the desired **Results** from the **Givens** (**Method**)

Analysis of Algorithm

How to calculate Running time of an algorithm?

- We can calculate the running time of an algorithm reliably by running the implementation of the algorithm on a computer.

- Alternatively we can calculate the running time by using a technique called algorithm analysis. We can estimate an algorithm's performance by counting the number of basic operations required by the algorithm to process an input of a certain size.

- The analysis of algorithms is the determination of the number of resources (such as time and storage) necessary to execute them.

Analysis of Algorithm – Contd.

Basic Operation: The time to complete a basic operation does not depend on the particular values of its operands. So it takes a constant amount of time.

Examples: Arithmetic operation (addition, subtraction, multiplication, division), Boolean operation (AND, OR, NOT), Comparison operation, Module operation, Branch operation etc.

Input Size: It is the number of input processed by the algorithm.

Example: For sorting algorithm the input size is measured by the number of records/elements to be sorted.

Growth Rate: The growth rate of an algorithm is the rate at which the running time (cost) of the algorithm grows as the size of the input grows.
The growth rate has a tremendous effect on the resources consumed by the algorithm.

Analysis of Algorithm – Contd.

algorithm to solve the problem of finding the 1st element in an array of n integers:

```
public int findFirstElement(int[] a){
int firstElement = a[0];
return firstElement;
}
```

It is clear that no matter how large the array is, the time to copy the value from the first position of the array is always constant (say k).

So the time T to run the algorithm as a function of n, T(n) = k . Here T(n) does not depend on the array size n. We always assume T(n) is a non-negative value.

Analysis of Algorithm – Contd.

Algorithm to find the smallest element in an array of n integers:

```
public int findSmallElement(int[] a){
int smElement = a[0];
for(int i=0; i<n ; i++)
if(a[i] < smElement)
smElement=a[i];
return smElement;
}
```

Here the basic operation is to compare between two integers and each comparison operation takes a fixed amount of time (say k) regardless of the value of the two integers or their position in the array.

In this algorithm the comparison operation is repeated n times due to for loop. So the running time of the above algorithm, T(n) = kn. So, this algorithm is said to have linear growth rate.

Analysis of Algorithm – Contd.

We want a reasonable approximation so we do ignore the time required to increment the variable i , the time for actual assignment when a smaller value is found or time taken to initialize the variable smElement.

Analysis of Algorithm – Contd.

Algorithm to find the smallest element from a two dimensional array n rows and n columns:

```
public int findSmallElement(int[][] a){
int smElement = a[0][0];
for(int i=0; i<n ; i++)
for(int j=0; j<n ; j++)
if(a[i][j] < smElement)
smElement=a[i][j];
return smElement;
}
```

The total number of comparison operation occurs n*n=n^2 times. So the running time of the algorithm, T(n) = kn^2. The above algorithm is said to have **quadratic** growth rate.

Why study algorithms?

- As the speed of processors increase, performance is often said to be less important than the other software quality characteristics (e.g. security, extensibility, re-usability, etc.). However, large problem sizes are common place in the area of computational science; which makes performance a very important factor.
- This is because longer computation time, mean slower results and higher cost of computation (if buying CPU hours from an external party).
- The study of algorithms, therefore, gives us a language to express performance as a function of problem size.
- A similar approach can also be taken to express resource requirements as a function of problem size.

Space Complexity

- The space complexity of an algorithm is the amount of memory it requires to run to its completion.

- The way in which the amount of storage space required by an algorithm varies with the size of the problem it is solving. Space complexity is normally expressed as an order of magnitude, e.g. O(N^2) means that if the size of the problem (N) doubles then four times as much working storage will be needed.

Components of Space needed by program

- **Instruction space:** stores the executable version of programs and is generally fixed.
- **Data space:**

  -Space required by constants and simple variables. Its space is fixed and/or is negligible.

  -Space needed by fixed/dynamic size stucture/class variables.

  - Dynamically allocated space. This space is usually variable.
- E**nviorntal stack:** to stores information required to reinvoke suspended processes or functions**.** It stores:

  - return address.

  - value of all local variables

  - value of all formal parameters in the function.

Time Complexity

- The time complexity of an algorithm is the amount of time it needs to run to its completion.

- The way in which the number of steps required by an algorithm varies with the size of the problem it is solving. Time complexity is normally expressed as an order of magnitude, e.g. O(N^3) means that if the size of the problem (N) doubles then the algorithm will take eight times as many steps to complete.

Components of time needed by program

- To measure the time complexity we can count all operations performed in an algorithm and if we know the time taken for each operation then we can easily compute the total time taken by the algorithm.

  This time varies from system to system.

- We estimate execution time of an algorithm irrespective of the computer on which it will be used. Hence, we identify the key operation and count such operation performed till the program completes its execution.

- The time complexity can be expressd as a function of a key operation performed.

Big-O Notation

- Big O notation is used to classify algorithms by how they respond (e.g., in their processing time or working space requirements) to changes in their input size.

- The Big-O notation or O(n) is a way to measure code efficiency based on the input size "n". This notation DOES NOT measure the exact number of lines of code executed, it measures the amount of code it will need to execute if we were to increase the input size "n" (growth function) in the worst case scenario.

.

Question

```
for(int i = 1; i<= n; ++i)  {
        if(array[i]==0) return true;
}
```

What is the Order for the above code?

Question

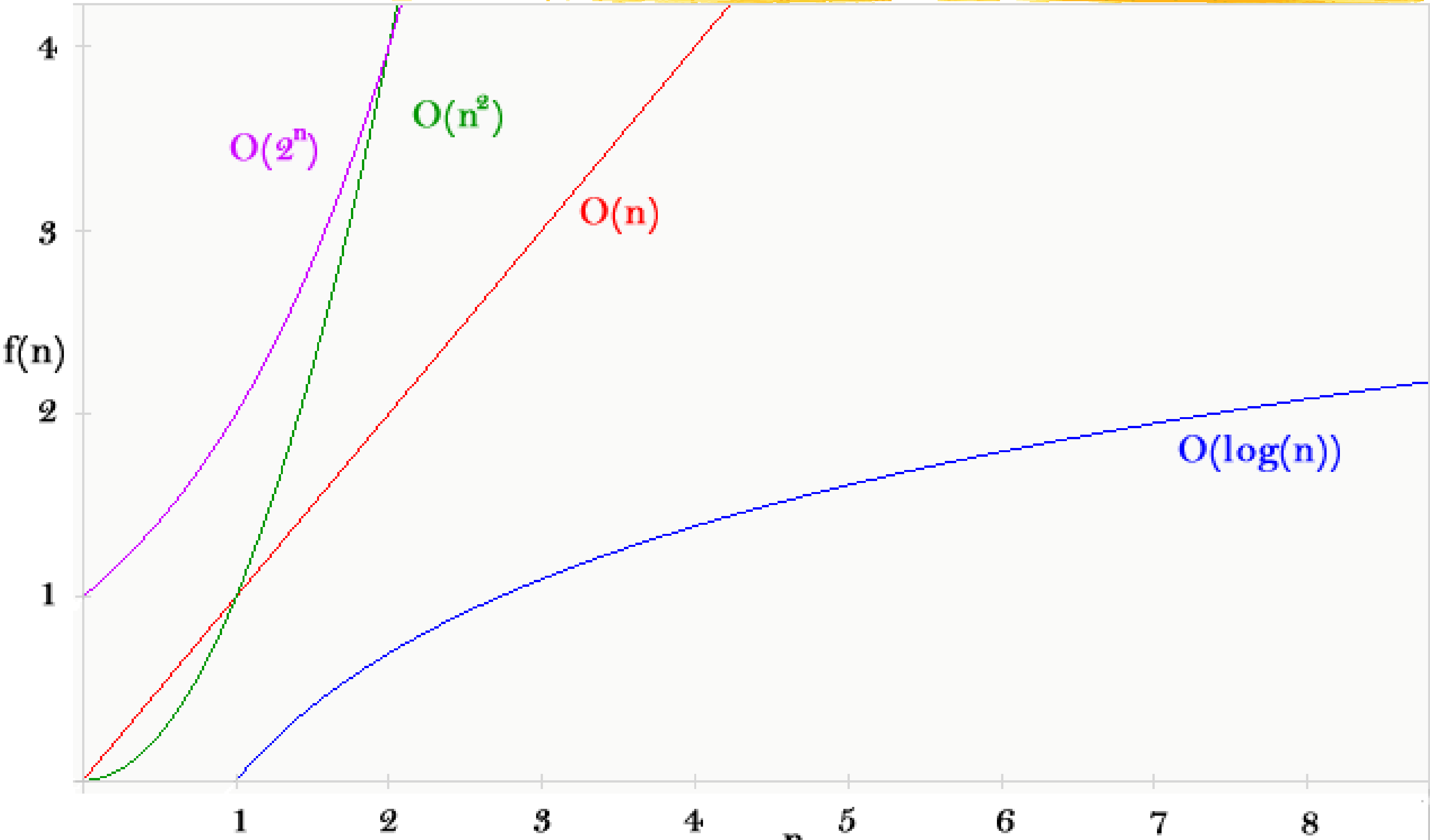
```
for(int i = 1; i<= n; ++i)  {
        for(int j = 1; j<= n; ++j)  {
          printf(“array[i][j] = %d”, array[i][j]);
          if(array[i][j] == 0) return true;
        }
        printf(“array[i][j] = %d”, array[i][j]);
}
```

What’s the Order of the above code?

Big-O notations from fastest to slowest:

- O(1) - Constant: input size n has no effect on execution time
- O(log(n)) - Log: When only part of the total input size is visited
- O(n) - Linear
- O(n*log(n)) - Some items are visited more then once
- O(n^2) - Power
- O(2^n) - Exponential: For every input size increased, the total work multiplies!
- O(n^n) - Hyper-Exponential

growth function f(n) relative to inpute size "n":



Sorting

- Given a set (container) of n elements
  - E.g. array, set of words, etc.
- Suppose there is an order relation that can be set across the elements
- Goal Arrange the elements in ascending order
  - Start → 1 23 2 56 9 8 10 100
  - End → 1 2 8 9 10 23 56 100

Bubble Sort

- Simplest sorting algorithm
- Idea:
  - 1. Set flag = false
  - 2. Traverse the array and compare pairs of two elements
    - 1.1 If E1 ≤ E2 - OK
    - 1.2 If E1 > E2 then Switch(E1, E2) and set flag = true
  - 3. If flag = true goto 1.
- What happens?

Bubble Sort

```
1   1   23   2    56   9    8    10   100
2   1   2    23   56   9    8    10   100
3   1   2    23   9    56   8    10   100
4   1   2    23   9    8    56   10   100
5   1   2    23   9    8    10   56   100
---- finish the first traversal  ----
----          start again            ----
1   1   2    23   9    8    10   56   100
2   1   2    9    23   8    10   56   100
3   1   2    9    8    23   10   56   100
4   1   2    9    8    10   23   56   100
---- finish the second traversal ----
----          start again            ----
......................
```

Why Bubble Sort ?

Implement Bubble Sort with an Array

```
void bubbleSort (Array S,  length n) {
    boolean isSorted = false;
     int j=1;
    while(!isSorted) {
    isSorted = true;

    for(i = 0; i<n-j; i++) {
         if(S[i] > S[i+1]) {
    int aux = S[i];
    S[i] = S[i+1];        S[i+1] = aux; isSorted = false;
    }
                 }
  j++
}
```

Running Time for Bubble Sort

- One traversal = move the maximum element at the end
- Traversal #i : $n – i + 1$ operations
- Running time:

  $(n – 1) + (n – 2) + … + 1 = (n – 1) n / 2 = O(n^2)$

- When does the worst case occur ?
- Best case ?

Divide-and-Conquer

- ***Divide and Conquer*** is a method of algorithm design (strategy) applied to solve sorting problem.
- Has three distinct steps:
  - **Divide**: If the input size is too large to deal with in a straightforward manner, divide the data into two or more disjoint subsets.
  - **Recur**: Use divide and conquer to solve the subproblems associated with the data subsets.
  - **Conquer**: Take the solutions to the subproblems and “merge” these solutions into a solution for the original problem.

Merge-Sort

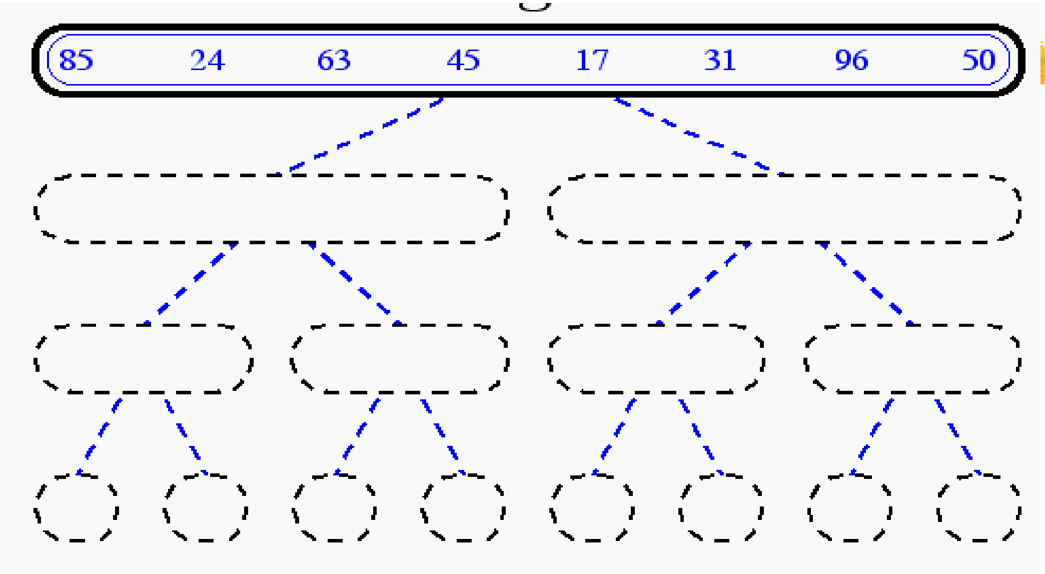
- Algorithm

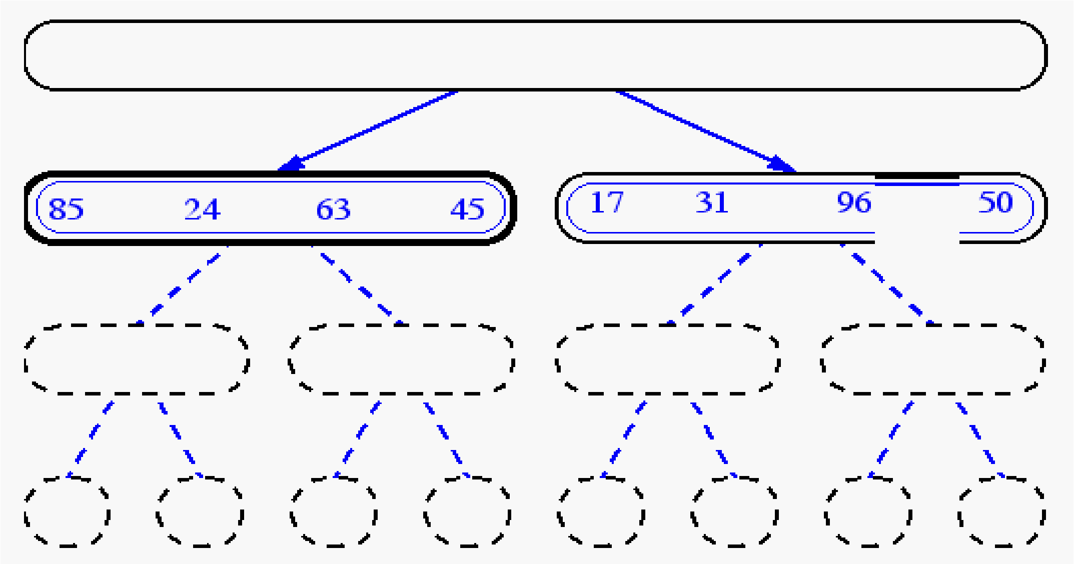
  - **Divide**: If S has at leas two elements (nothing needs to be done if S has zero or one elements), remove all the elements from S and put them into two sequences, S_1 and S_2, each containing about half of the elements of S. (i.e. S_1 contains the first $\lceil n/2 \rceil$ elements and S_2 contains the remaining $\lfloor n/2 \rfloor$ elements.

  - **Recur**: Recursive sort sequences S_1 and S_2.

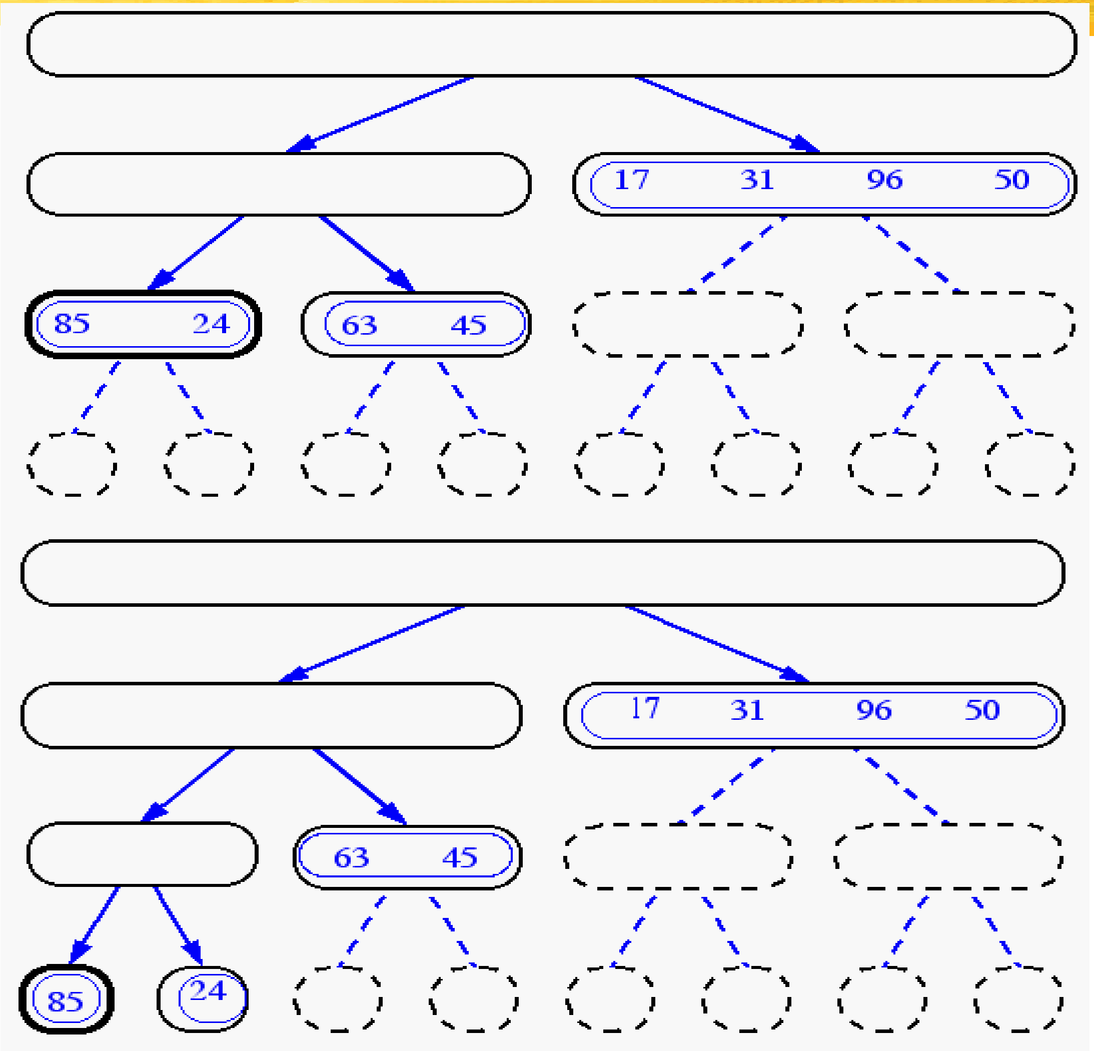
  - **Conquer**: Put back the elements into S by merging the sorted sequences S_1 and S_2 into a unique sorted sequence.

Merge-Sort

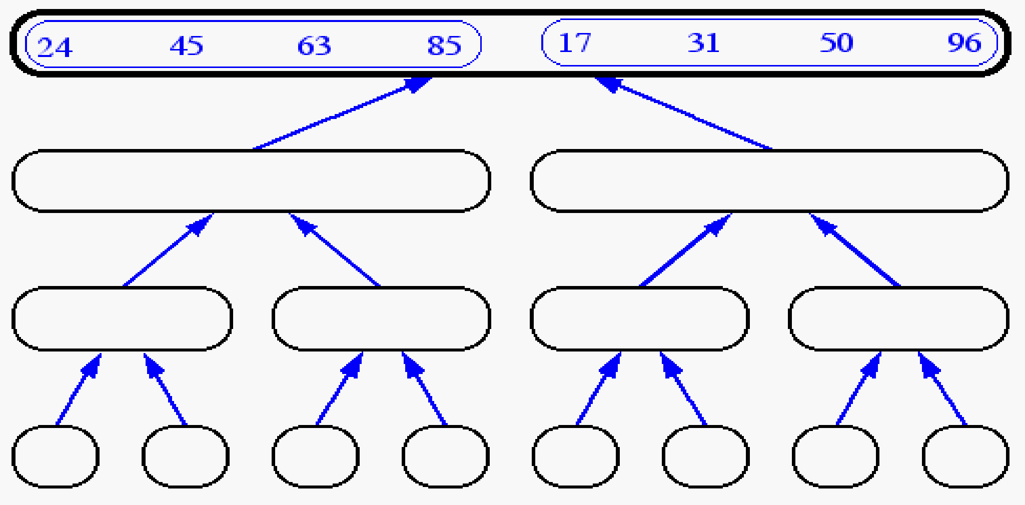


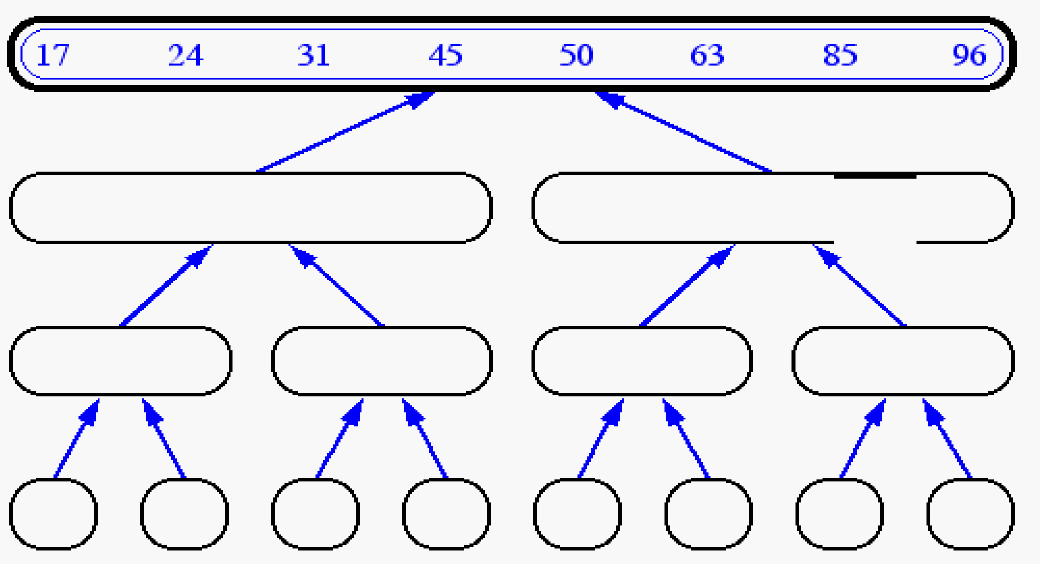


Merge-Sort(cont.)



Merge-Sort (cont'd)





Merge-Sort

- Algorithm

  Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.
-
- Step 1 − if it is only one element in the list it is already sorted, return.
- Step 2 − divide the list recursively into two halves until it can no more be divided.
- Step 3 − merge the smaller lists into new list in sorted order.

Merge-Sort

- Algorithm

```
procedure mergesort( var a as array )
   if ( n == 1 ) return a

   var l1 as array = a[0] ... a[n/2]
   var l2 as array = a[n/2+1] ... a[n]

   l1 = mergesort( l1 )
   l2 = mergesort( l2 )

   return merge( l1, l2 )
end procedure
```

Merge-Sort -algorithm – cont.

```
procedure merge( var a as array, var b as array )

  var c as array
  while ( a and b have elements )
    if ( a[0] > b[0] )
      add b[0] to the end of c
      remove b[0] from b
    else
      add a[0] to the end of c
      remove a[0] from a
    end if
  end while

  while ( a has elements )
    add a[0] to the end of c
    remove a[0] from a
  end while

  while ( b has elements )
    add b[0] to the end of c
    remove b[0] from b
  end while

  return c
end procedure
```