# UNIT I

**Syllabus:**
**Data Structures** - Definition, Classification of Data Structures, Operations on Data Structures, Abstract Data Type (ADT)
**Preliminaries of algorithms**. Time and Space complexity.
**Searching** - Linear search, Binary search, Fibonacci search.
**Sorting**- Insertion sort, Selection sort, Exchange (Bubble sort, quick sort), distribution (radix sort), merging (Merge sort) algorithms.

---

## 1. DATA STRUCTURES

### 1.1 Basic Concepts:

### Data:

- The term *data* means a value or set of values. It specifies either the value of a variable or a constant.
- Eg: marks of students, name of an employee, address of a customer, value of *pi*, etc
- Data item that does not have subordinate data items is categorized as an **elementary item.**
- Data item that is composed of one or more subordinate data items is called a **group item.**

For example,

- Group item is a student's name which may be divided into three sub-items—first name, middle name, and last name
- Elementary item is roll number would normally be treated as a single item.

### Record:

- A *record* is a collection of data items.
- For example, the name, address, course, and marks obtained are individual data items. But all these data items can be grouped together to form a record.

### File:

- A *file* is a collection of related records.
- For example, if there are 60 students in a class, then there are 60 records of the students. All these related records are stored in a file.
- Each record in a file may consist of multiple data items but the value of a certain data item uniquely identifies the record in the file. Such a data item K is called a *primary key*.
- For example, in a student's record that contains roll number, name, address, course, and marks obtained, the field roll number is a primary key.

## 1.2 Definition:

- A *data structure* is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently.
- A data structure is a systematic way of organizing and accessing data.
- The logical or mathematical model of a particular organization of data is called a Data structure.

Data structures are used in almost every program or software system. Some common examples of data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables.

## 1.3 Applications of Data Structures:

Data structures are widely applied in the following areas:

- Compiler design
- Operating system
- Statistical analysis package
- DBMS
- Numerical analysis
- Simulation
- Artificial intelligence
- Graphics

## 1.4 Selecting a data Structure

When selecting a data structure to solve a problem, the following steps must be performed.

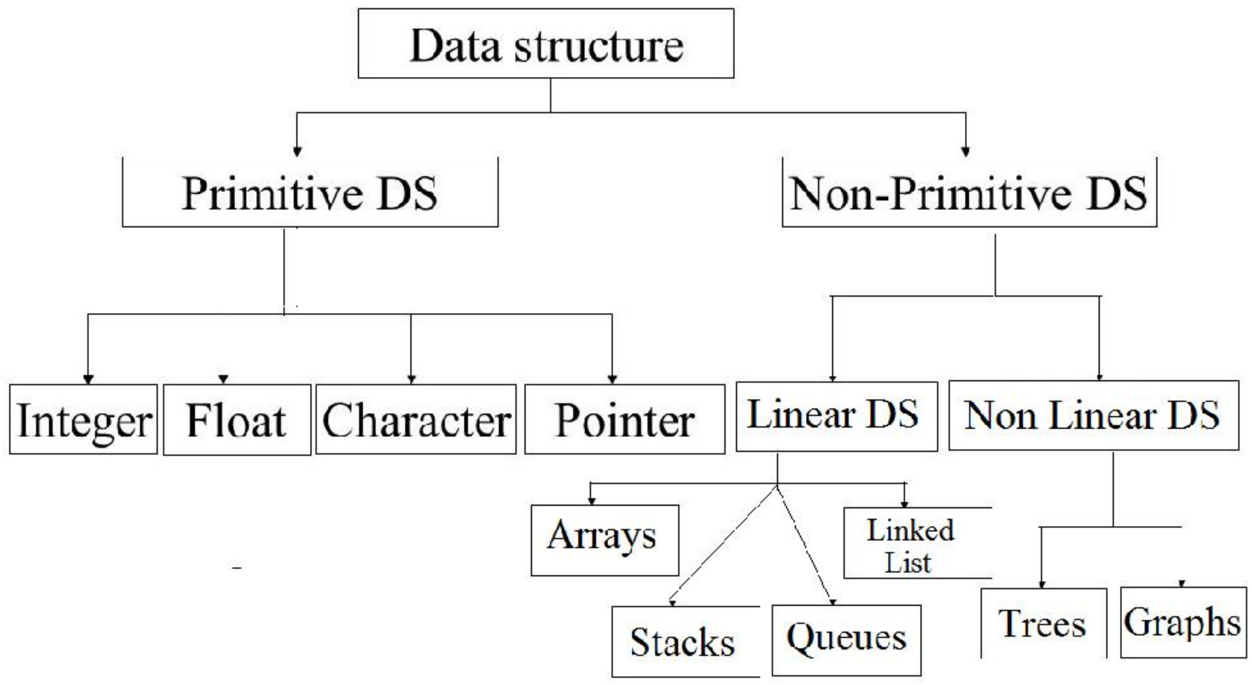
1. Analysis of the problem to determine the basic operations that must be supported. For example, basic operation may include inserting/deleting/searching a data item from the data structure.
2. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

This three-step approach to select an appropriate data structure for the problem at hand supports a data-centred view of the design process.
In the approach, the first concern is the data and the operations that are to be performed on them. The second concern is the representation of the data,
and the final concern is the implementation of that representation.

Data structures are building blocks of a program. A program built using improper data structures obtained are individual data items. But all these data items can be grouped together to form a record.

## 2. CLASSIFICATION OF DATA STRUCTURES

.



### 2.1 Classification of Data Structures

Data structures are generally categorized into two classes:
1.Primitive data strucutures
2. Non-primitive data structures.

**Primitive and Non-primitive Data Structures**

- Primitive data structures are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character, and boolean. The terms 'data type', 'basic data type', and 'primitive data type' are often used interchangeably.
- Non-primitive data structures are those data structures which are created using primitive data structures. Examples of such data structures include linked lists, stacks, trees, and graphs.

Non-primitive data structures can further be classified into two categories:

1. **Linear Data Structures**
2. **Non Linear Data Structures**

**Linear Structures data structures**

- If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure.
- Examples include arrays, linked lists, stacks, and queues.
- Linear data structures can be represented in memory in two different ways.
    i. One way is to have to a linear relationship between elements by means of sequential memory locations.
    ii. The other way is to have a linear relationship between elements by means of links.

**Non-linear Linear data structures**

- If the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure. The relationship of adjacency is not maintained between elements of a non-linear data structure.
- Examples include trees and graphs.

## 2.2 Arrays

An array is a collection of similar data elements. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an *index* .

In C, arrays are declared using the following syntax:

**type name[size];**

For example, int marks[10];
The above statement declares an array marks that contains 10 elements. In C, the array index starts from zero. This means that the array marks will contain 10 elements in all. The first element will be stored in marks[0], second element in marks[1], so on and so forth. Therefore, the last element, that is the 10th element, will be stored in marks[9]. In the memory, the array will be stored as shown in Fig.

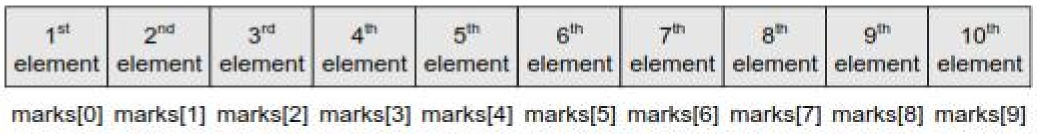


**Fig** Memory representation of an array of 10 elements

Arrays are generally used when we want to store large amount of similar type of data. But they have the following limitations:

- Arrays are of fixed size.
- Data elements are stored in contiguous memory locations which may not be always available.

- Insertion and deletion of elements can be problematic because of shifting of elements from their positions.

## 2.3 Linked Lists

- A linked list is a very flexible, dynamic data structure in which elements (called *nodes*) form a sequential list.
- In contrast to static arrays, a programmer need not worry about how many elements will be stored in the linked list. This feature enables the programmers to write robust programs which require less maintenance.
- In a linked list, each node is allocated space as it is added to the list. Every node in the list points to the next node in the list.
- In a linked list, every node contains the following two types of data:
  - The value or data of the node
  - A pointer or link to the next node in the list

The last node in the list contains a NULL pointer to indicate that it is the end or *tail* of the list.
Since the memory for a node is dynamically allocated when it is added to the list, the total number of nodes that may be added to a list is limited only by the amount of memory available.

Figure below shows a linked list of seven nodes.

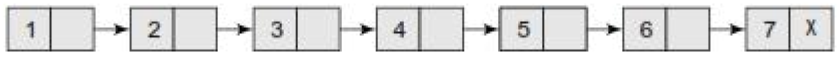


**Figure** Simple linked list

***Advantage***: Easier to insert or delete data elements
***Disadvantage***: Slow search operation and requires more memory space

## 2.4 Stacks

- A stack is a linear data structure in which insertion and deletion of elements are done at only one end, which is known as the top of the stack.
- Stack is called a last-in, first-out (LIFO) structure because the last element which is added to the stack is the first element which is deleted from the stack.

In the computer's memory, Stacks can be implemented using arrays or linked lists.

Figure below shows the array implementation of a stack.

- Every stack has a variable top associated with it.
- Top is used to store the address of the topmost element of the stack.
- It is this position from where the element will be added or deleted.
- There is another variable MAX, which is used to store the maximum number of elements that the stack can store.

If top = NULL, then it indicates that the stack is empty and if top = MAX–1, then the stack is full.

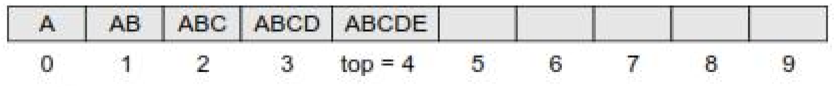


**Figure** Array representation of a stack

In the above Fig, top = 4, so insertions and deletions will be done at this position. Here, the stack can store a maximum of 10 elements where the indices range from 0–9. In the above stack, five more elements can still be stored.

A stack supports three basic operations:

1. Push,
2. Pop
3. Peep (Top most element)

- The push operation adds an element to the top of the stack.
- The pop operation removes the element from the top of the stack.
- The peep operation returns the value of the topmost element of the stack (without deleting it).

Before inserting an element in the stack, we must check for overflow conditions. An overflow occurs stack is already full.

Before deleting an element from the stack, we must check for underflow conditions. An underflow condition occurs when a stack that is already empty.

## 2.5 Queues

- A queue is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first one to be taken out.
- The elements in a queue are added at one end called the rear and removed from the other end called the front.

Queues can be implemented by using either arrays or linked lists.

Every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively. Consider the queue shown in below Fig.

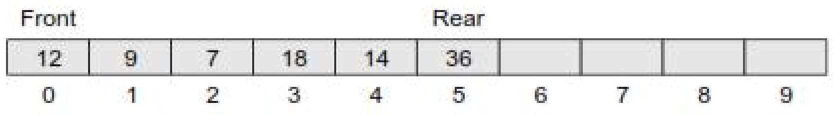


**Figure** Array representation of a queue

Here, front = 0 and rear = 5. If we want to add one more value to the list, say, if we want to add another element with the value 45, then the rear would be incremented by 1 and the value would be stored at the position pointed by the rear. The queue, after the addition, would be as shown in the below Fig. Here, front = 0 and rear = 6.

Every time a new element is to be added, we will repeat the same procedure.

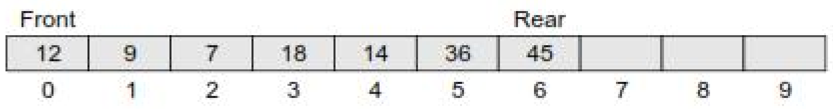


**Figure** Queue after insertion of a new element

Now, if we want to delete an element from the queue, then the value of front will be incremented.
Deletions are done only from this end of the queue. The queue after the deletion will be as shown in the below Fig.

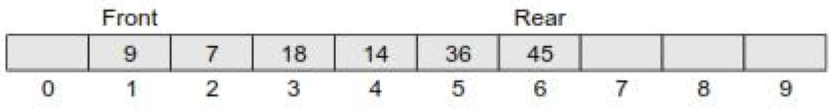


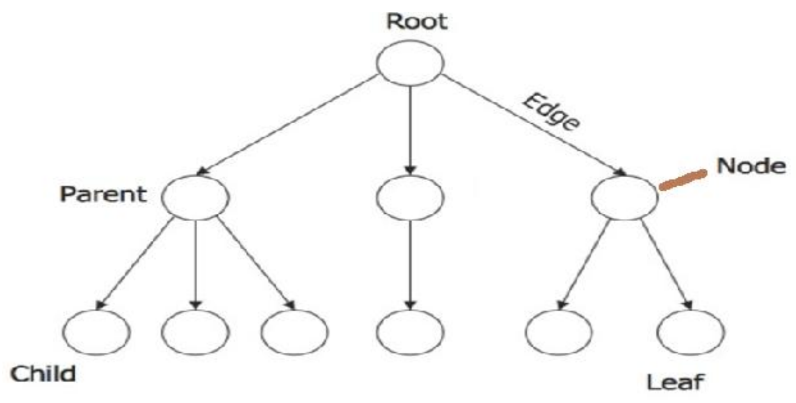
**Figure** Queue after deletion of an element

Before inserting an element in the queue, we must check for overflow conditions.
An overflow occurs when we try to insert an element into a queue that is already full.

Before deleting an element from the queue, we must check for underflow conditions.
An underflow condition occurs when we try to delete an element from a queue that is already empty.

## 2.6 Tree:

A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical order.
One of the nodes is designated as the root node, and the remaining nodes can be partitioned into disjoint sets such that each set is a sub-tree of the root.



The simplest form of a tree is a binary tree. A binary tree consists of a root node and left and right sub-trees, where both sub-trees are also binary trees. Each node contains a data element, a left pointer which points to the left sub-tree, and a right pointer which points to the right sub-tree.

The root element is the topmost node which is pointed by a ‘root’ pointer.
If root = NULL then the tree is empty.

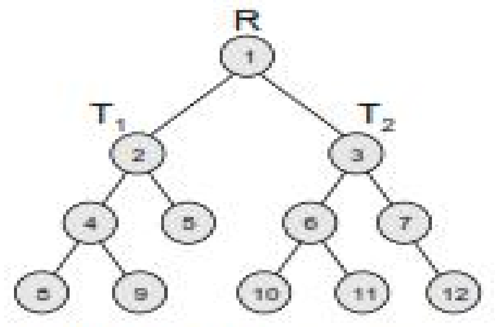


**Figure 2.7** Binary tree

The above figure shows a binary tree, where R is the root node and T1 and T2 are left and right subtrees of R. If T1 is non-empty, then T1 is said to be the left successor of R. Likewise, if T2 is non-empty,then T2 is called the right successor of R.
Here, node 2 is the left child and node 3 is the right child of the root node 1.
The left sub-tree of the root node consists of the nodes 2, 4, 5, 8, and 9.
The right sub-tree of the root node consists of the nodes 3, 6, 7, 10, 11, and 12.

***Advantage***: Provides quick search, insert, and delete operations
***Disadvantage***: Complicated deletion algorithm

## 2.7 Graphs:

A graph is a non-linear data structure which is a collection of *vertices* (also called *nodes*) and *edges* that connect these vertices.

In a tree structure, nodes can have any number of children but only one parent, a graph on the other hand relaxes all such kinds of restrictions. Figure below shows a graph with five nodes.

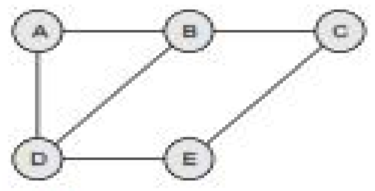


**Figure** Graph

- A node in the graph may represent a city and the edges connecting the nodes can represent roads.
- A graph can also be used to represent a computer network where the nodes are workstations and the edges are the network connections.
- Graphs have so many applications in computer science and mathematics that several algorithms have been written to perform the standard graph operations, such as searching the graph and finding the shortest path between the nodes of a graph.

Unlike trees, graphs do not have any root node.
Every node in the graph can be connected with every another node in the graph.
When twonodes are connected via an edge, the two nodes are known as *neighbours*. For example, in the above Figure, node A has two neighbours: B and D.

***Advantage***: Best models real-world situations
***Disadvantage***: Some algorithms are slow and very complex

## 3. OPERATIONS ON DATA STRUCTURES

This section discusses the different operations that can be performed on the various data structures previously mentioned.

- ***Traversing***
  It means to access each data item exactly once so that it can be processed.
  For example, to print the names of all the students in a class.

- ***Searching***
  It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items.
  For example, to find the names of all the students who secured 100 marks in mathematics.

- ***Inserting***
  It is used to add new data items to the given list of data items.
  For example, to add the details of a new student who has recently joined the course.

- ***Deleting***
  It means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course.

- ***Sorting***
  Data items can be arranged in some order like ascending order or descending order depending on the type of application.
  For example, arranging the names of students in a class in an alphabetical order.

- ***Merging***
  Lists of two sorted data items can be combined to form a single list of sorted data items.

## 4. ABSTRACT DATA TYPE

- An *abstract data type* (ADT) is the way we look at a data structure, focusing on what it does and ignoring how it does its job.
- An ADT refers to a set of data values and associated operations that are specified accurately.

.

### *Data type*

- Data type of a variable is the set of values that the variable can take. The basic data types in C include int, char, float, and double.

### *Abstract*

- The word 'abstract' in the context of data structures means *considered apart from the detailed specifications or implementation*.
- In C, an abstract data type can be a structure considered without regard to its implementation. It can be thought of as a 'description' of the data in the structure with a list of operations that can
- be performed on the data within that structure.
- For example, when we use a stack or a queue, the user is concerned only with the type of data and the operations that can be performed on it. Therefore, the fundamentals of how the data is stored should be invisible to the user. They should not be concerned with how the methods work or what structures are being used to store the data. They should just know that to work with stacks, they have push() and pop() functions available to them. Using these functions, they can manipulate the data (insertion or deletion) stored in the stack.

**Advantage of using ADTs**

In the real world, programs *evolve* as a result of new requirements or constraints, so a modification to a program commonly requires a change in one or more of its data structures.

For example, if you want to add a new field to a student's record to keep track of more information about each student, then it will be better to replace an array with a linked structure to improve the program's efficiency. In such a scenario, rewriting every procedure that uses the changed structure is not desirable. Therefore, a better alternative is to *separate* the use of a data structure from the details of its implementation. This is the principle underlying the use of abstract data types.

## 5. ALGORITHMS

- Algorithm is '**a formally defined procedure** for performing some calculation'. If a procedure is formally defined, then it can be implemented using a formal language, and such a language is known as a *programming language*.
- An algorithm provides a blueprint to write a program to solve a particular problem. It is considered to be an effective procedure for solving a problem in finite number of steps.
- Algorithms are mainly used to achieve *software reuse*. Once an idea or a blueprint of a solution is developed, it can implemented in any high-level language like C, C++, or Java.
- An algorithm is basically a set of instructions that solve a problem. It is not uncommon to have multiple algorithms to tackle the same problem, but the choice of a particular algorithm must depend on the time and space complexity of the algorithm.

## 5.1 DIFFERENT APPROACHES TO DESIGNING AN ALGORITHM

Algorithms are used to manipulate the data contained in data structures. When working with data structures, algorithms are used to perform operations on the stored data.
A complex algorithm is often divided into smaller units called modules. This process of dividing an algorithm into modules is called modularization. The key advantages of modularization are as follows:

- It makes the complex algorithm simpler to design and implement.
- Each module can be designed independently. While designing one module, the details of other modules can be ignored, thereby enhancing clarity in design which in turn simplifies implementation, debugging, testing, documenting, and maintenance of the overall algorithm.

There are two main approaches to design an algorithm—top-down approach and bottom-up approach, as shown in the below Figure.

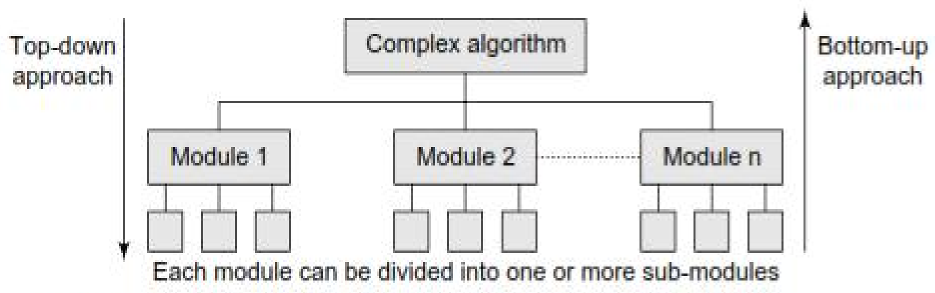


**Figure** Different approaches of designing an algorithm

## *Top-down approach*

- A top-down design approach starts by dividing the complex algorithm into one or more modules.
- These modules can further be decomposed into one or more sub-modules, and this process of decomposition is iterated until the desired level of module complexity is achieved.
- Top-down design method is a form of stepwise refinement where we begin with the top most module and incrementally add modules that it calls.
- Therefore, in a top-down approach, we start from an abstract design and then at each step, this design is refined into more concrete levels until a level is reached that requires no further refinement.

## *Bottom-up approach*

- A bottom-up approach is just the reverse of top-down approach.
- In the bottom-up design, we start with designing the most basic or concrete modules and then proceed towards designing higher level modules.
- The higher level modules are implemented by using the operations performed by lower level modules. Thus, in this approach sub-modules are grouped together to form a higher level module.
- All the higher level modules are clubbed together to form even higher level modules. This process is repeated until the design of the complete algorithm is obtained.

### *Top-down vs bottom-up approach*

Whether the top-down strategy should be followed or a bottom-up is a question that can be answered depending on the application at hand.

While top-down approach follows a stepwise refinement by decomposing the algorithm into manageable modules, the bottom-up approach on the other hand defines a module and then groups together several modules to form a new higher level module.

Top-down approach is highly appreciated for ease in documenting the modules, generation of test cases, implementation of code, and debugging. However, it is also criticized because the sub-modules are analysed in isolation without concentrating on their communication with other modules or on reusability of components and little attention is paid to data, thereby ignoring the concept of information hiding.

Although the bottom-up approach allows information hiding as it first identifies what has to be encapsulated within a module and then provides an abstract interface to define the module's boundaries as seen from the clients. But all this is difficult to be done in a strict bottom-up strategy.

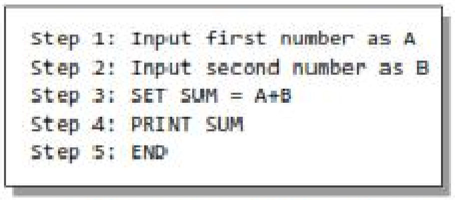
| TOP-DOWN APPROACH | BOTTOM-UP APPROACH |
|---|---|
| Breaks the complex problem into smaller subproblems. | Solves the fundamental low-level problem and integrates them into a larger one. |
| Communication not required in the top-down approach. | Needs a specific amount of communication. |
| Contain redundant information. | Redundancy can be eliminated. |
| Structure/procedural oriented programming languages (i.e. C) follows the top-down approach. | Object-oriented programming languages (like C++, Java, etc.) follows the bottom-up approach. |
| Mainly used in Module documentation, test case creation, code implementation and debugging. | Mainly used in Testing |

## 5.2 CONTROL STRUCTURES USED IN ALGORITHMS

An algorithm may employ one of the following control structures:
(a) sequence,
(b) decision, and
(c) repetition.

### Sequence

In sequence, each step of an algorithm is executed in a specified order.
Let us write an algorithm to add two numbers. This algorithm performs the steps in a purely sequential order, as shown in the below Figure.



```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: SET SUM = A+B
Step 4: PRINT SUM
Step 5: END
```

**Figure** Algorithm to add two numbers

### Decision

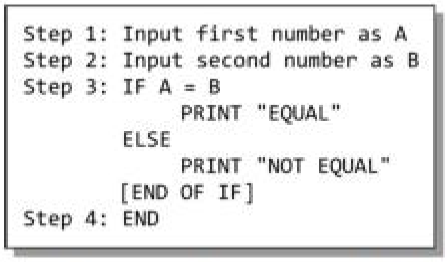
Decision statements are used when the execution of a process depends on the outcome of some condition. A condition in this context is any statement that may evaluate to either a true value or a false value.

A decision statement can also be stated in the following manner:

```
IF condition
        Then process1
ELSE process2
```

This form is popularly known as the IF–ELSE construct. Here, if the condition is true, then process1 is executed, else process2 is executed.

The below Figure shows an algorithm to check if two numbers are equal.



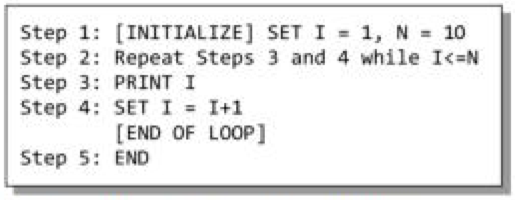
```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: IF A = B
             PRINT "EQUAL"
        ELSE
             PRINT "NOT EQUAL"
        [END OF IF]
Step 4: END
```

**Figure** Algorithm to test for equality of two numbers

## Repetition

Repetition, which involves executing one or more steps for a number of times, can be implemented using constructs such as while, do–while, and for loops. These loops execute one or more steps until some condition is true.

The below Figure shows an algorithm that prints the first 10 natural numbers.



```
Step 1: [INITIALIZE] SET I = 1, N = 10
Step 2: Repeat Steps 3 and 4 while I<=N
Step 3: PRINT I
Step 4: SET I = I+1
        [END OF LOOP]
Step 5: END
```

**Figure** Algorithm to print the first 10 natural

## 6. TIME AND SPACE COMPLEXITY

Analysing an algorithm means determining the amount of resources (such as time and memory) needed to execute it. The efficiency or complexity of an algorithm is stated in terms of time and space complexity.

- The *time complexity* of an algorithm is basically the running time of a program as a function of the input size i.e time required to run the program.

- The *space complexity* of an algorithm is the amount of computer memory that is required during the program execution as a function of the input size i.e the amount of memory space required by the algorithm.

- In other words, the number of machine instructions which a program executes is called its time complexity. This number is primarily dependent on the size of the program's input and the algorithm used.

- Generally, the space needed by a program depends on the following two parts:
    - *Fixed part*: It varies from problem to problem. It includes the space needed for storing instructions, constants, variables, and structured variables (like arrays and structures).
    - *Variable part*: It varies from program to program. It includes the space needed for recursion stack, and for structured variables that are allocated space dynamically during the runtime of a program.

However, running time requirements are more critical than memory requirements.

## 6.1 Worst-case, Average-case, Best-case, and Amortized Time Complexity

***Worst-case running time*** This denotes the behaviour of an algorithm with respect to the worst possible case of the input instance. The worst-case running time of an algorithm is an upper bound on the running time for any input. Therefore, having the knowledge of worst-case running time gives us an assurance that the algorithm will never go beyond this time limit.

***Average-case running time***
The average-case running time of an algorithm is an estimate of the running time for an average' input. It specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution. Average-case running time assumes that all inputs of a given size are equally likely.

***Best-case running time***
The term 'best-case performance' is used to analyse an algorithm under optimal conditions. For example, the best case for a simple linear search on an array occurs when the desired element is the first in the list.

While developing and choosing an algorithm to solve a problem, we hardly base our decision on the best-case performance. It is always recommended to improve the average performance and the worst-case performance of an algorithm.

***Amortized running time***
Amortized running time refers to the time required to perform a sequence of (related) operations averaged over all the operations performed. Amortized analysis guarantees the average performance of each operation in the worst case.

### 6.1.1 Time–Space Trade-off

The best algorithm to solve a particular problem at hand is that requires less memory space and takes less time to complete its execution.

But practically, designing such an ideal algorithm is not a trivial task. There can be more than one algorithm to solve a particular problem. One may require less memory space, while the other may require less CPU time to execute. Thus, it is not uncommon to sacrifice one thing for the other. Hence, there exists a time–space trade-off among algorithms.

So, if space is a big constraint, then one might choose a program that takes less space at the cost of more CPU time.

On the contrary, if time is a major constraint, then one might choose a program that takes minimum time to execute at the cost of more space.

### 6.1.2 Expressing Time and Space Complexity -

The time and space complexity can be expressed using a function f(n) where n is the input size for a given instance of the problem being solved.
Expressing the complexity is required when

- We want to predict the rate of growth of complexity as the input size of the problem increases.
- There are multiple algorithms that find a solution to a given problem and we need to find the algorithm that is most efficient.

The most widely used notation to express this function f(n) is the Big O notation.
It provides the upper bound for the complexity.

### 6.1.3 Algorithm Efficiency

- If a function is linear (without any loops or recursions), the efficiency of that algorithm or the running time of that algorithm can be given as the number of instructions it contains.

- If an algorithm contains loops, then the efficiency of that algorithm may vary depending on the number of loops and the running time of each loop in the algorithm.

Let us consider different cases in which loops determine the efficiency of an algorithm.

**Linear Loops**
To calculate the efficiency of an algorithm that has a single loop, we need to first determine the number of times the statements in the loop will be executed. This is because the number of iterations is directly proportional to the loop factor. Greater the loop factor, more is the number of iterations. For example, consider the loop given below:

```
for(i=0;i<100;i++)
          statement block;
```

Here, 100 is the loop factor.

We have already said that efficiency is directly proportional to the number of iterations. Hence, the general formula in the case of linear loops may be given as

f(n) = n

However calculating efficiency is not as simple as is shown in the above example. Consider the loop given below:

```
for(i=0;i<100;i+=2)
        statement block;
```

Here, the number of iterations is half the number of the loop factor. So, here the efficiency can be given as

f(n) = n/2

## Logarithmic Loops

In logarithmic loops, the loop-controlling variable is either multiplied or divided during each iteration of the loop. For example, look at the loops given below:

```
for(i=1;i<1000;i*=2)                          for(i=1000;i>=1;i/=2)
        statement block;                              statement block;
```

Consider the first for loop in which the loop-controlling variable i is multiplied by 2. The loop will be executed only 10 times and not 1000 times because in each iteration the value of i doubles.
Consider the second loop in which the loop-controlling variable i is divided by 2.
In this case also, the loop will be executed 10 times.
Thus, the number of iterations is a function of the number by which the loop-controlling variable is divided or multiplied.
In the examples discussed, it is 2.
That is, when n = 1000, the number of iterations can be given by log 1000 which is approximately equal to 10.

Therefore, putting this analysis in general terms, we can conclude that the efficiency of loops in which iterations divide or multiply the loop-controlling variables can be given as

f(n) = log n

## Nested Loops

Loops that contain loops are known as *nested loops*.
The total is then obtained as the product of the number of iterations in the inner loop and the number of iterations in the outer loop.

In this case, we analyse the efficiency of the algorithm based on whether it is a linear logarithmic, quadratic, or dependent quadratic nested loop.

***Linear logarithmic loop***

Consider the following code in which the loop-controlling variable of the inner loop is multiplied after each iteration. The number of iterations in the inner loop is log 10. This inner loop is controlled by an outer loop which iterates 10 times. Therefore, according the formula, the number of iterations for this code can be given as
10 log 10.

```
for(i=0;i<10;i++)
            for(j=1; j<10;j*=2)
                      statement block;
```

In more general terms, the efficiency of such loops can be given as f(n) = n log n.

***Quadratic loop***

In a quadratic loop, the number of iterations in the inner loop is equal to the number of iterations the outer loop. Consider the following code in which the outer loop executes 10 times and for each iteration of the outer loop, the inner loop also executes 10 times.

Therefore, the efficiency here is 100.

```
for(i=0;i<10;i++)
           for(j=0; j<10;j++)
                      statement block;
```

The generalized formula for quadratic loop can be given as f(n) = $n^2$
.

***Dependent quadratic loop***

In a dependent quadratic loop, the number of iterations in the inner loop is dependent on the outer loop. Consider the code given below:

```
for(i=0;i<10;i++)
           for(j=0; j<=i;j++)
                      statement block;
```

In this code, the inner loop will execute just once in the first iteration, twice in the second iteration, thrice in the third iteration, so on and so forth. In this way, the number of iterations can be calculated as
1 + 2 + 3 + ... + 9 + 10 = 55
If we calculate the average of this loop (55/10 = 5.5), we will observe that it is equal to the number of iterations in the outer loop (10) plus 1 divided by 2. In general terms, the inner loop iterates (n+ 1)/2 times. Therefore, the efficiency of such a code can be given as
f(n) = n (n + 1)/2

## 7. EXPRESSING TIME AND SPACE COMPLEXITY

The time and space complexity can be expressed using a function f(n) where n is the input size for a given instance of the problem being solved.

Expressing the complexity is required when

- We want to predict the rate of growth of complexity as the input size of the problem increases.
- There are multiple algorithms that find a solution to a given problem and we need to find the algorithm that is most efficient.

The most widely used notation to express this function f(n) is the Big O notation. It provides the upper bound for the complexity.

## 7.1 Big O notation

- If there are two different algorithms to solve the same problem where one algorithm executes in 10 iterations and the other in 20 iterations, the difference between the two algorithms is not much.
- But, if the first algorithm executes in 10 iterations and the other in 1000 iterations, then it is a matter of concern.
- The number of statements executed in the program for n elements of the data is a function of the number of elements, expressed as f(n). This factor is the Big O, and is expressed as O(n).

The Big O notation, where O stands for 'order of ', is concerned with what happens for very large values of n.

When expressing complexity using the Big O notation, constant multipliers are ignored. So, an O(4n) algorithm is equivalent to O(n), which is how it should be written.

If f(n) and g(n) are the functions defined on a positive integer number n, then f(n) = O(g(n))
That is, f of n is Big–O of g of n if and only if positive constants c and n exist, such that f(n) <=cg(n).

It means that for large amounts of data, f(n) will grow no more than a constant factor than g(n).

Hence, g provides an upper bound.
Note that here c is a constant which depends on the following factors:

- the programming language used,
- the quality of the compiler or interpreter,
- the CPU speed,
- the size of the main memory and the access time to it,
- the knowledge of the programmer, and
- the algorithm itself, which may require simple but also time-consuming machine instructions.

Big O notation provides a strict upper bound for f(n). This means that the function f(n) can do better but not worse than the specified value.
Big O notation is simply written asf(n) ∈ O(g(n)) or as f(n) = O(g(n)).

**If f(n) and g(n) are functions defined on positive integer number n, then**

$f(n) = O(g(n))$: there exist positive constants $c$ and $n_0$ such that

$$0 \le f(n) \le cg(n) \text{ for all } n \ge n_0$$

Examples of functions in o($n^3$) include: $n^{2.9}$, $n^3$, $n^3 + n$, $540n^3 + 10$.

Examples of functions not in o($n^3$) include: $n^{3.2}$, $n^2$, $n^2 + n$, $540n + 10$, $2n$

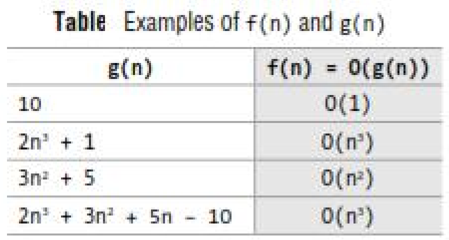
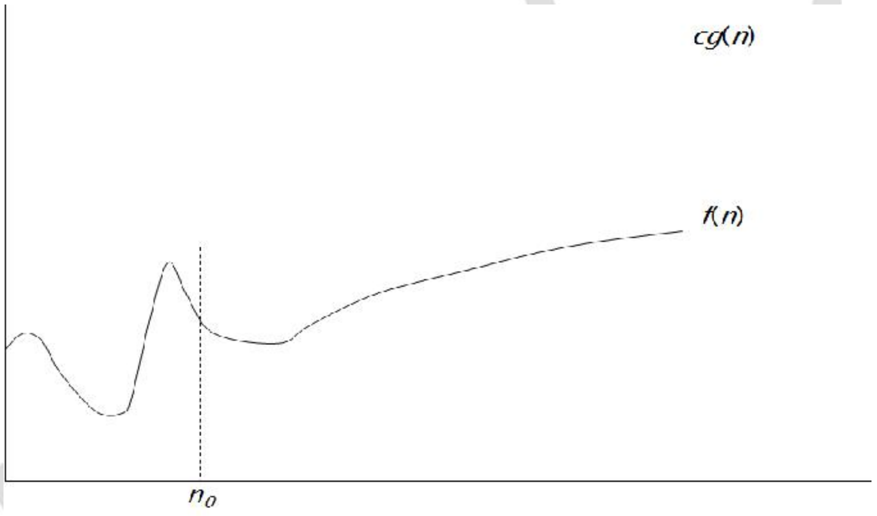


**Table** Examples of f(n) and g(n)

| g(n) | f(n) = O(g(n)) |
|---|---|
| 10 | O(1) |
| $2n^3 + 1$ | O($n^3$) |
| $3n^2 + 5$ | O($n^2$) |
| $2n^3 + 3n^2 + 5n - 10$ | O($n^3$) |



Visualization of $O(g(n))$

**Categories of Algorithms**

According to the Big O notation, we have five different categories of algorithms:

- Constant time algorithm: running time complexity given as O(1)
- Linear time algorithm: running time complexity given as O(n)
- Logarithmic time algorithm: running time complexity given as O(log n)
- Polynomial time algorithm: running time complexity given as O($n^k$) where k > 1
- Exponential time algorithm: running time complexity given as O($2^n$)

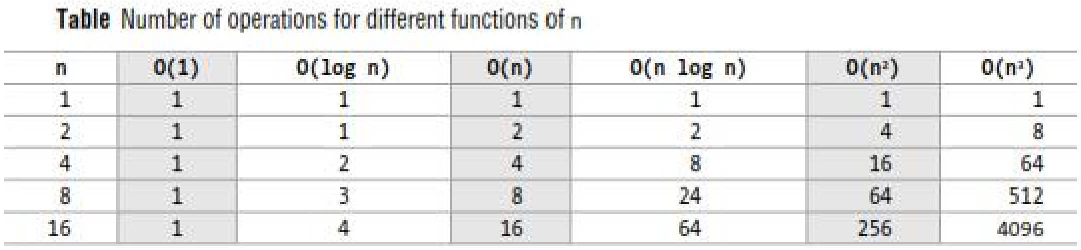
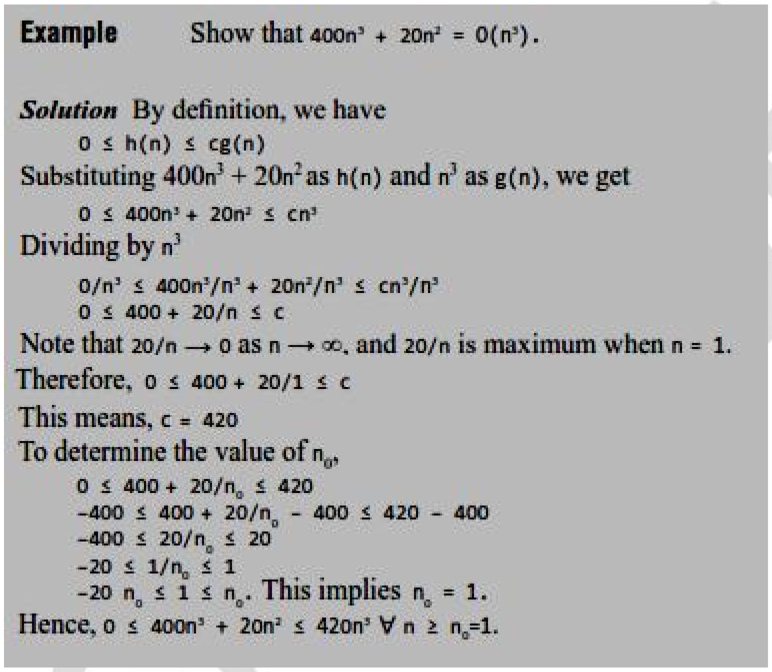


**Table** Number of operations for different functions of n

| n | O(1) | O(log n) | O(n) | O(n log n) | $O(n^2)$ | $O(n^3)$ |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 2 | 2 | 4 | 8 |
| 4 | 1 | 2 | 4 | 8 | 16 | 64 |
| 8 | 1 | 3 | 8 | 24 | 64 | 512 |
| 16 | 1 | 4 | 16 | 64 | 256 | 4096 |



**Example** Show that $400n^3 + 20n^2 = O(n^3)$.

***Solution*** By definition, we have

$$0 \le h(n) \le cg(n)$$

Substituting $400n^3 + 20n^2$ as h(n) and $n^3$ as g(n), we get

$$0 \le 400n^3 + 20n^2 \le cn^3$$

Dividing by $n^3$

$$0/n^3 \le 400n^3/n^3 + 20n^2/n^3 \le cn^3/n^3$$
$$0 \le 400 + 20/n \le c$$

Note that $20/n \to 0$ as $n \to \infty$, and $20/n$ is maximum when $n = 1$.

Therefore, $0 \le 400 + 20/1 \le c$

This means, $c = 420$

To determine the value of $n_0$,

$$0 \le 400 + 20/n_0 \le 420$$
$$-400 \le 400 + 20/n_0 - 400 \le 420 - 400$$
$$-400 \le 20/n_0 \le 20$$
$$-20 \le 1/n_0 \le 1$$

$-20\ n_0 \le 1 \le n_0$. This implies $n_0 = 1$.

Hence, $0 \le 400n^3 + 20n^2 \le 420n^3\ \forall\ n \ge n_0=1$.

**Limitations of Big O Notation**

There are certain limitations with the Big O notation of expressing the complexity of algorithms.

These limitations are as follows:

- Many algorithms are simply too hard to analyse mathematically.
- There may not be sufficient information to calculate the behaviour of the algorithm in the average case.
- Big O analysis only tells us how the algorithm grows with the size of the problem, not how efficient it is, as it does not consider the programming effort.
- It ignores important constants. For example, if one algorithm takes $O(n^2)$ time to execute and the other takes $O(100000\ n^2)$ time to execute, then as per Big O, both algorithm have equal time complexity. In real-time systems, this may be a serious consideration.

## 7.2 Omega notation (Ω)

The Omega notation provides a tight lower bound for f(n).

This means that the function can never do better than the specified value but it may do worse.

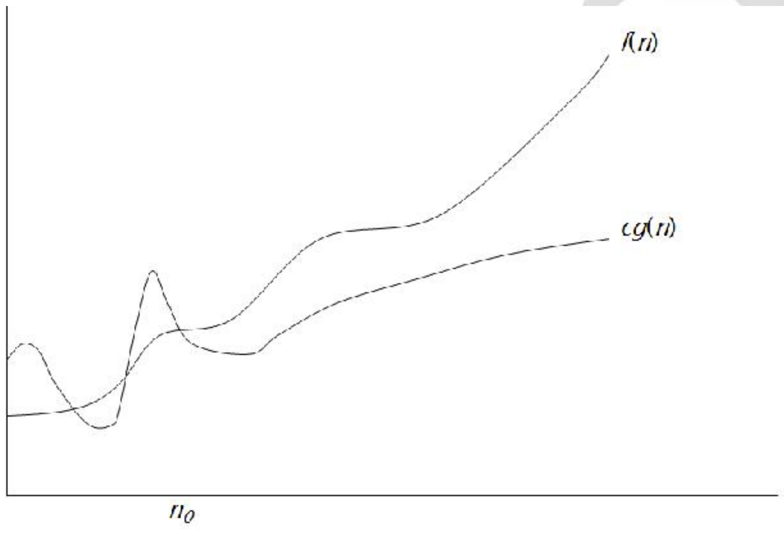
Ω notation is simply written as, $f(n) \in \Omega(g(n))$,

**If f(n) and g(n) are functions defined on positive integer number n, then**

$$f(n) = \Omega(g(n)): \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$$
$$0 \le f(n) \ge cg(n) \text{ for all } n \ge n_0$$

It means, f(n) does not go less than a constant factor g(n). Hence g is lower bond.

Examples of functions in $\Omega(n^2)$ include: $n^2$, $n^{2.9}$, $n^3 + n^2$, $n^3$
Examples of functions not in $\Omega(n^3)$ include: $n$, $n^{2.9}$, $n^2$



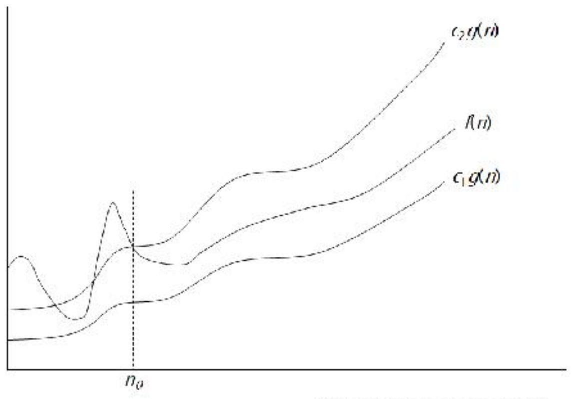
Visualization of $\Omega(g(n))$

## 7.3 Theta-notation(θ)

Theta notation provides a tight bound for f(n).

**If f(n) and g(n) are functions defined on positive integer number n, then**

$$f(n) = \Theta(g(n)): \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that}$$
$$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0$$



Visualization of $\Theta(g(n))$

## 7.4 Little o Notation

- This notation provides a non-asymptotically tight upper bound for f(n).
- To express the function using this notation, we write f(n)=o(g(n))

- Little o notation is used to describe an upper bound that cannot be tight. In other words, loose upper bound of f(n).
- Let f(n) and g(n) are the functions that map positive real numbers. We can say that the function f(n) is o(g(n)) if for any real positive constant c, there exists an integer constant n0 ≤ 1 such that f(n) > 0.

**Mathematical Relation of Little o notation**

Using mathematical relation, we can say that f(n) = o(g(n)) means,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

## 7.5 Little –Omega (ω )Notation

- It is denoted by (ω).
- Little omega (ω) notation is used to describe a loose lower bound of f(n).
- Let f(n) and g(n) are the functions that map positive real numbers. We can say that the function f(n) is o(g(n)) if for any real positive constant c, there exists an integer constant n0 ≤ 1 such that f(n) > 0.

**Mathematical Relation of Little o notation**

Using mathematical relation, we can say that f(n) = o(g(n)) means

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

## 8. SEARCHINGS

- Searching refers to finding the position of a value in a collection of values.
- Searching means to find whether a particular value is present in an array or not.
  - If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array.
  - If the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful.

There are two popular methods for searching the array elements:

1. ***Linear search***
2. ***Binary search*.**

Other methods of searching are **Interpolation search method, Jump search method and Fibonacci search method.**

### 8.1 Linear Search:

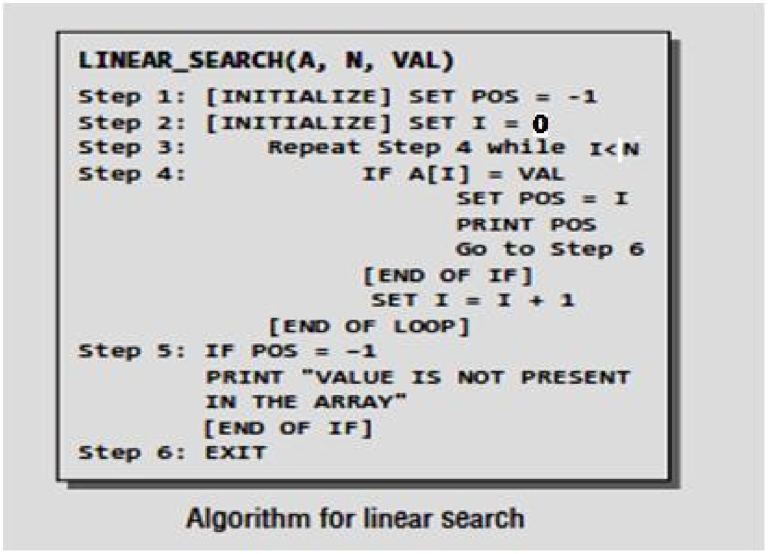
Linear search, also called as *sequential search*, is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found.

Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted).

For example, if an array A[] is declared and initialized as,
int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};
and the value to be searched is VAL = 7.

Then searching means to find whether the value '7' is present in the array or not. If yes, then it returns the position of its occurrence. Here, POS = 3 (index starting from 0).

### Linear Search Algorithm

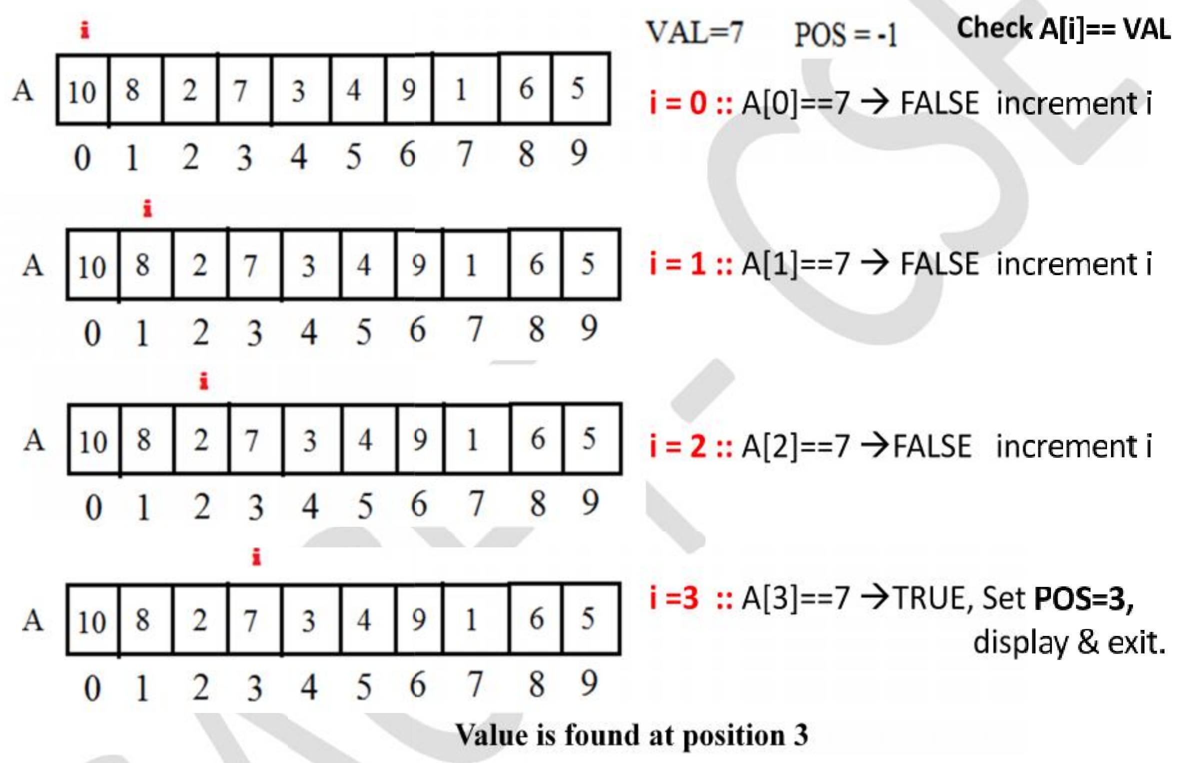


```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 0
Step 3:     Repeat Step 4 while I<N
Step 4:         IF A[I] = VAL
                    SET POS = I
                    PRINT POS
                    Go to Step 6
                [END OF IF]
                SET I = I + 1
            [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Algorithm for linear search

In Steps 1 and 2 of the algorithm, we initialize the value of POS and I.
In Step 3, a while loop is executed that would be executed till I is less than N (total number of elements in the array).
In Step 4, a check is made to see if a match is found between the current array element and VAL. If a match is found, then the position of the array element is printed, else the value of I is incremented to match the next element with VAL.
If all the array elements have been compared with VAL and no match is found, then it means that VAL is not present in the array.

## Working of Linear Search


VAL=7
POS = -1
Check A[i]== VAL
i = 0 :: A[0]==7 → FALSE increment i
i = 1 :: A[1]==7 → FALSE increment i
i = 2 :: A[2]==7 →FALSE increment i
i =3 :: A[3]==7 →TRUE, Set POS=3, display & exit.
Value is found at position 3

## Complexity of Linear Search Algorithm

Linear search executes in O(n) time where n is the number of elements in the array.

The best case of linear search is when VAL is equal to the first element of the array. In this case, only one comparison will be made.

The worst case will happen when either VAL is not present in the array or it is equal to the last element of the array.

In both the cases, n comparisons will have to be made.

## 8.2 Binary Search

- Binary search is a searching algorithm that works efficiently with a sorted list.
- Binary search is a fast search algorithm with run-time complexity of O(log n).
- This search algorithm works on the principle of divide and conquer.

Example: The mechanism of binary search can be better understood by an analogy of a telephone directory. When we are searching for a particular name in a directory, we first open the directory from the middle and then decide whether to look for the name in the first part of the directory or in the second part of the directory. Again, we open some page in the middle and the whole process is repeated until we finally find the right name.

Binary search looks for a particular item by comparing the middle most item of the collection.

If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the sub-array reduces to zero.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

In this algorithm, we see that BEG and END are the beginning and ending positions of the segment that we are looking to search for the element.

MID is calculated as (BEG + END)/2.
Initially, BEG = lower_bound and END = upper_bound.

The algorithm will terminate when A[MID] = VAL. When the algorithm ends, we will set POS = MID. POS is the position at which the value is present in the array.

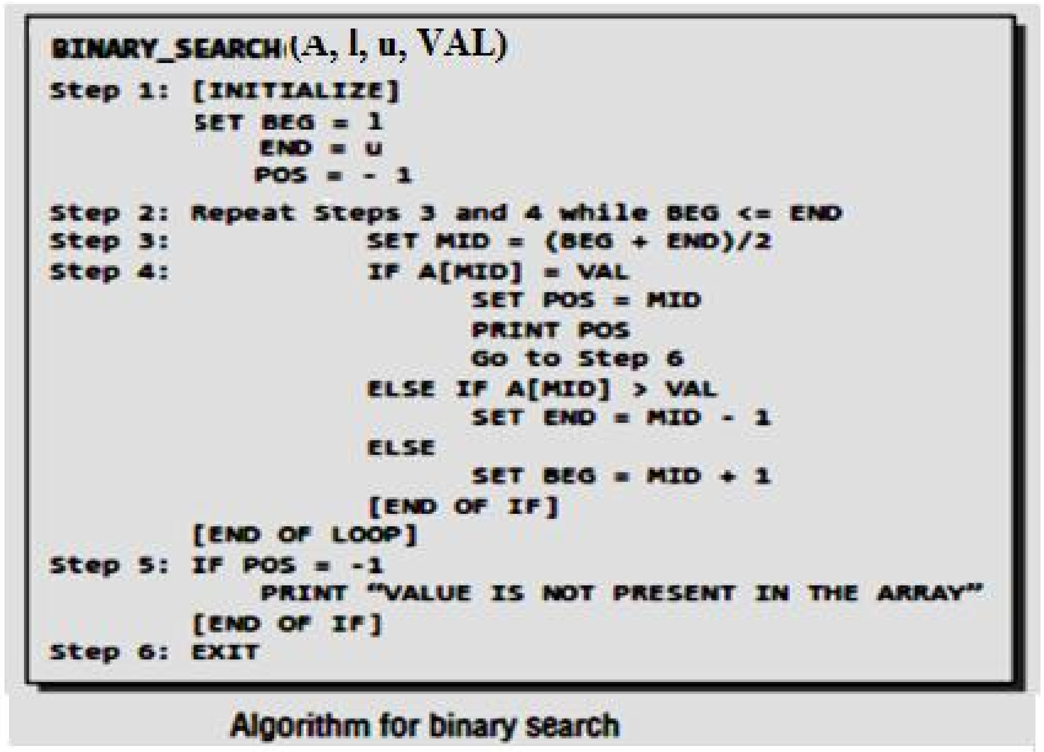
If VAL is not equal to A[MID], then the values of BEG, END, and MID will be changed depending on whether VAL is smaller or greater than A[MID].

(a) If A[MID]>VAL, then VAL will be present in the left segment of the array. So, the value of END will be changed as END = MID – 1.

(b) If A[MID]<VAL, then VAL will be present in the right segment of the array. So, the value of BEG will be changed as BEG = MID + 1.

Finally, if VAL is not present in the array, then eventually, END will be less than BEG. When this happens, the algorithm will terminate and the search will be unsuccessful.

## Algorithm for Binary Search



```
BINARY_SEARCH(A, l, u, VAL)
Step 1: [INITIALIZE]
        SET BEG = l
            END = u
            POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:            SET MID = (BEG + END)/2
Step 4:            IF A[MID] = VAL
                          SET POS = MID
                          PRINT POS
                          Go to Step 6
                   ELSE IF A[MID] > VAL
                          SET END = MID - 1
                   ELSE
                          SET BEG = MID + 1
                   [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
             PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

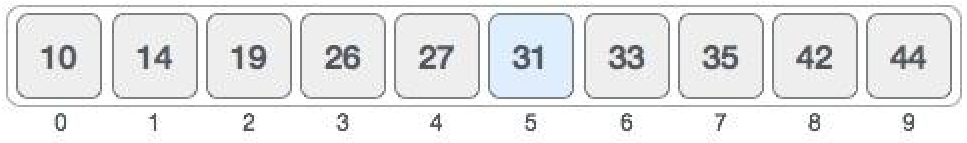
Algorithm for binary search

In Step 1, we initialize the value of variables, BEG, END, and POS.
In Step 2, a while loop is executed until BEG is less than or equal to END.
In Step 3, the value of MID is calculated.
In Step 4, we check if the array value at MID is equal to VAL (item to be searched in the array). If a match is found, then the value of POS is printed and the algorithm exits. However, if a match is not found, and if the value of A[MID] is greater than VAL, the value of END is modified, otherwise if A[MID] is greater than VAL, then the value of BEG is altered.
In Step 5, if the value of POS = –1, then VAL is not present in the array and an appropriate message is printed on the screen before the algorithm exits.

## Working of Binary search

For a binary search to work, it is mandatory for the target array to be sorted.

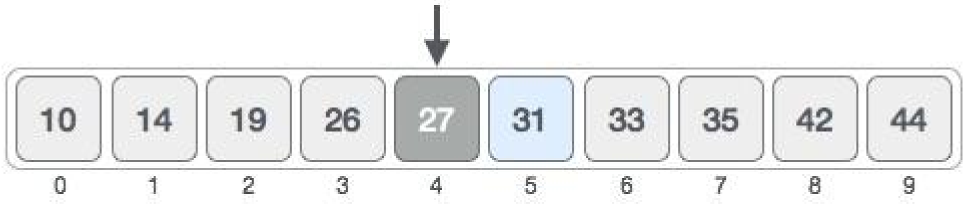
Here is the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



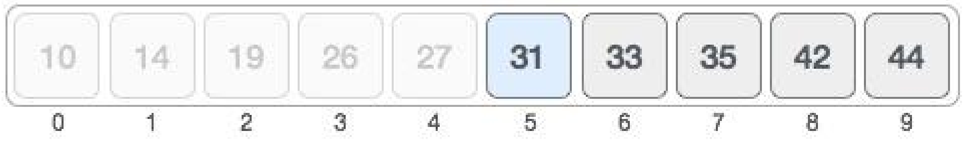
First, we shall determine half of the array by using this formula.

mid = (beg + end) / 2

Here it is, (0 + 9) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



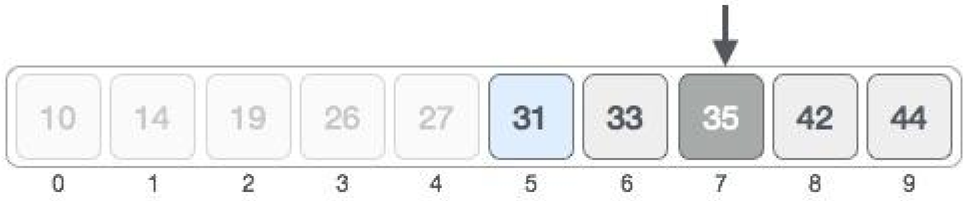
Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.



We change our low to mid + 1 and find the new mid value again.

beg = mid + 1 =5
mid = (beg + end) / 2 = (5 + 9) / 2 =7

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



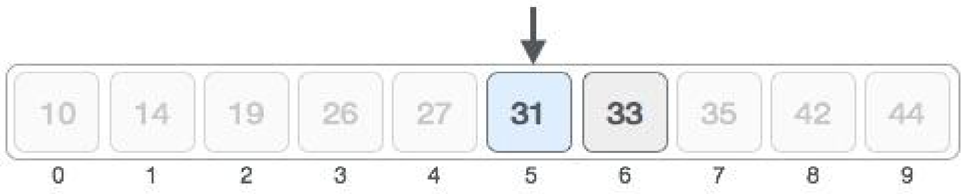
The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Now, we change out end=mid -1 and find new mid value again.

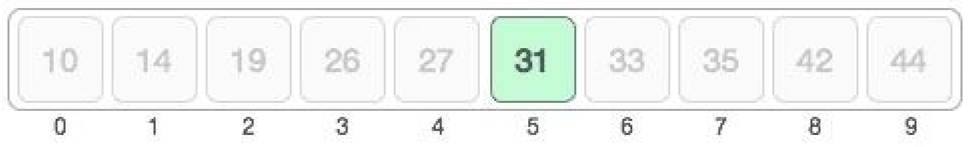
end=6

mid= (beg+mid) / 2= (5+6) / 2 =5

Hence, new mid is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

.

**Complexity of Binary Search Algorithm**

The complexity of the binary search algorithm can be expressed as f(n), where n is the number of elements in the array. The complexity of the algorithm is calculated depending on the number of comparisons that are made.

In the binary search algorithm, we see that with each comparison, the size of the segment where search has to be made is reduced to half.

Thus, we can say that, in order to locate a particular value in the array, the total number of comparisons that will be made is given as $2^{f(n)} > n$ or $f(n) = \log_2 N$

## 8.3 Fibonacci Search

**Fibonacci series:**

- Fibonacci series are series in which the first two terms are 0 and 1 and then each successive term is the sum of previous two terms.
- Fibonacci Numbers are recursively defined as F(n) = F(n-1) + F(n-2), F(0) = 0, F(1) = 1.
- In the Fibonacci series given below, each number is called a Fibonacci number.
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

**Fibonacci Search:**

- The Fibonacci series and concept can be used to search for a given value in a list of numbers. Such a search algorithm which is based on Fibonacci numbers is called Fibonacci search
- It was developed by Kiefer in 1953.
- Fibonacci Search is a comparison-based technique that uses Fibonacci numbers to search an element in a sorted array.

**Similarities with Binary Search:**

- Works for sorted arrays
- A Divide and Conquer Algorithm.
- Has O(Log n) time complexity.

**Differences with Binary Search:**

- Fibonacci Search divides given array in unequal parts
- Binary Search uses division operator to divide range. Fibonacci Search doesn't use /, but uses + and -. The division operator may be costly on some CPUs.

- Fibonacci Search examines relatively closer elements in subsequent steps. So when input array is big Fibonacci Search can be useful.

The key advantage of Fibonacci search over binary search is that comparison dispersion is low.

## Algorithm:

Let the searched element be VAL
Let arr[1..n] be the input array and element to be searched be VAL.

1. Find the smallest Fibonacci Number greater than or equal to n.
   Let this number be $F_m$
   Let the two Fibonacci numbers preceding it be $F_{m-1}$ and $F_{m-2}$

2. While the array has elements to be inspected:

   a. Compare VAL with the last element of the range covered by $F_{m-2}$

   b. If VAL matches, return index

   c. If VAL is less than the element, move the three Fibonacci variables **two Fibonacci down**, indicating elimination of approximately rear two-third of remaining array.

   d. If VAL is greater than the element, move the three Fibonacci variables **one Fibonacci down. Reset offset to index.** Together these indicate elimination of approximately front one-third of the remaining array.

3. Since there might be a single element remaining for comparison, check if $F_{m-1}$ is 1. If Yes, compare VAL with that remaining element. If match, return index.

Let us understand the algorithm with below example:

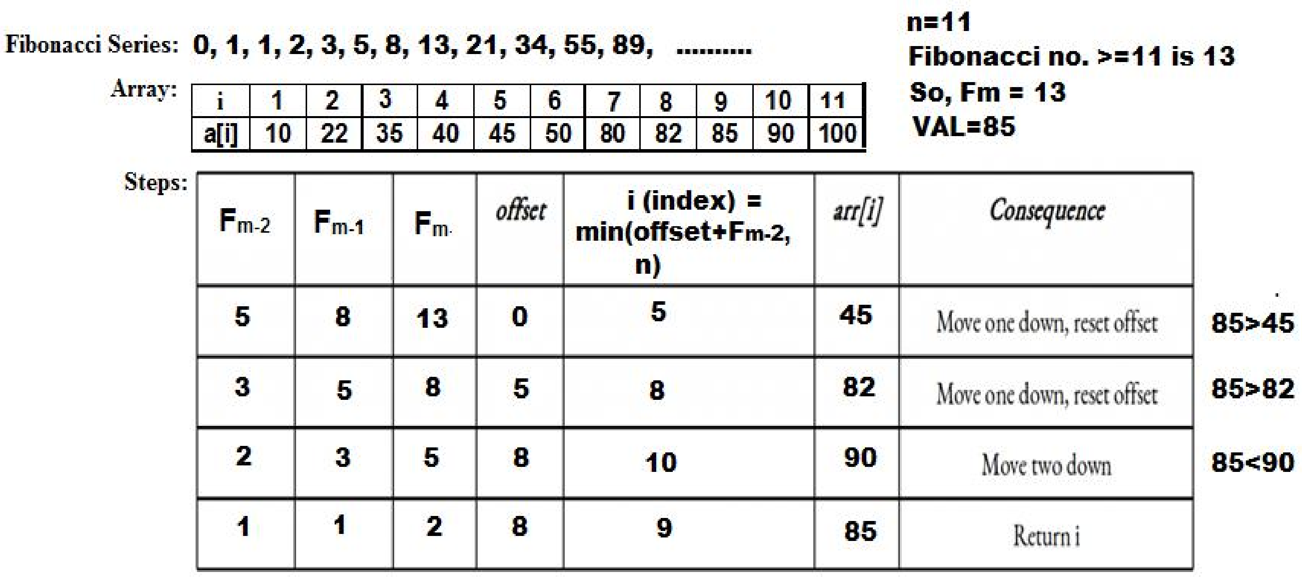
| **i** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **a[i]** | **10** | **22** | **35** | **40** | **45** | **50** | **80** | **82** | **85** | **90** | **100** |

Illustration assumption: 1-based indexing.
Target element VAL is 85.
Length of array n = 11.
Smallest Fibonacci number greater than or equal to 11 is 13.

As per our illustration, $F_{m-2} = 5$, $F_{m-1} = 8$, and $F_m = 13$.

Another implementation detail is the offset variable (zero initialized). It marks the range that has been eliminated, starting from the front. We will update it time to time.

Now since the offset value is an index and all indices including it and below it have been eliminated, it only makes sense to add something to it. Since $F_{m-2}$ marks approximately one-third of our array, as well as the indices it marks are sure to be valid ones, we can add $F_{m-2}$ to offset and check the element at index i = min(offset + $F_{m-2}$, n).



Fibonacci Series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ..........

n=11
Fibonacci no. >=11 is 13
So, Fm = 13
VAL=85

Array:

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| a[i] | 10 | 22 | 35 | 40 | 45 | 50 | 80 | 82 | 85 | 90 | 100 |

Steps:

| $F_{m-2}$ | $F_{m-1}$ | $F_m$ | *offset* | i (index) = min(offset+$F_{m-2}$, n) | *arr[i]* | *Consequence* | |
|---|---|---|---|---|---|---|---|
| 5 | 8 | 13 | 0 | 5 | 45 | Move one down, reset offset | 85>45 |
| 3 | 5 | 8 | 5 | 8 | 82 | Move one down, reset offset | 85>82 |
| 2 | 3 | 5 | 8 | 10 | 90 | Move two down | 85<90 |
| 1 | 1 | 2 | 8 | 9 | 85 | Return i | |

**Time Complexity of Fibonacci Search:**

- Time Complexity of Fibonacci search is O( log n).
- Fibonacci search reduces the searching elements.
- It eliminates front one-third array elements from searching array or rear two-third array elements from searching array.

**Sorting-** Insertion sort, Selection sort, Exchange (Bubble sort, quick sort), distribution (radix sort), merging (Merge sort) algorithms

## 9. Sorting

- Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending.
- Sorting algorithm is a method to rearrange the list of elements either in ascending or descending order, which can be numerical, lexicographical, or any user-defined order.
- Sorting is a process through which the data is arranged in ascending or descending order.

That is, if A is an array, then the elements of A are arranged in a sorted order (ascending order) in such a way that A[0] < A[1] < A[2] < ...... < A[N].

For example, if we have an array that is declared and initialized as
int A[] = {21, 34, 11, 9, 1, 0, 22};
Then the sorted array (ascending order) can be given as:
A[] = {0, 1, 9, 11, 21, 22, 34};

There are two types of sorting:

- **Internal Sorting**
- **External Sorting**

***Internal sorting*** which deals with sorting the data stored in the computer's memory.
If the data to be sorted remains in main memory and also the sorting is carried out in main memory it is called internal sorting. Internal Sorting takes place in the main memory of a computer. The internal sorting methods are applied to small collection of data. It means that, the entire collection of data to be sorted in small enough that the sorting can take place within main memory.
The following are some internal sorting techniques:
1. Insertion sort
2. Selection sort
3. Merge Sort
4. Radix Sort
5. Quick Sort
6. Heap Sort
7. Bubble Sort

***External sorting*** which deals with sorting the data stored in files. External sorting is applied when there is voluminous data that cannot be stored in the memory.

If the data resides in secondary memory and is brought into main memory in blocks for sorting and then result is returned back to secondary memory is called external sorting.

External sorting is required when the data being sorted do not fit into the main memory (usually RAM) and instead they must reside in the slower external memory (usually a hard drive).

The following are some external sorting techniques:

1. Two-Way External Merge Sort
2. K-way External Merge Sort

## 9.1 Insertion Sort

Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time.

The main idea behind insertion sort is that it inserts each item into its proper place in the final list. To save memory, most implementations of the insertion sort algorithm work by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place.

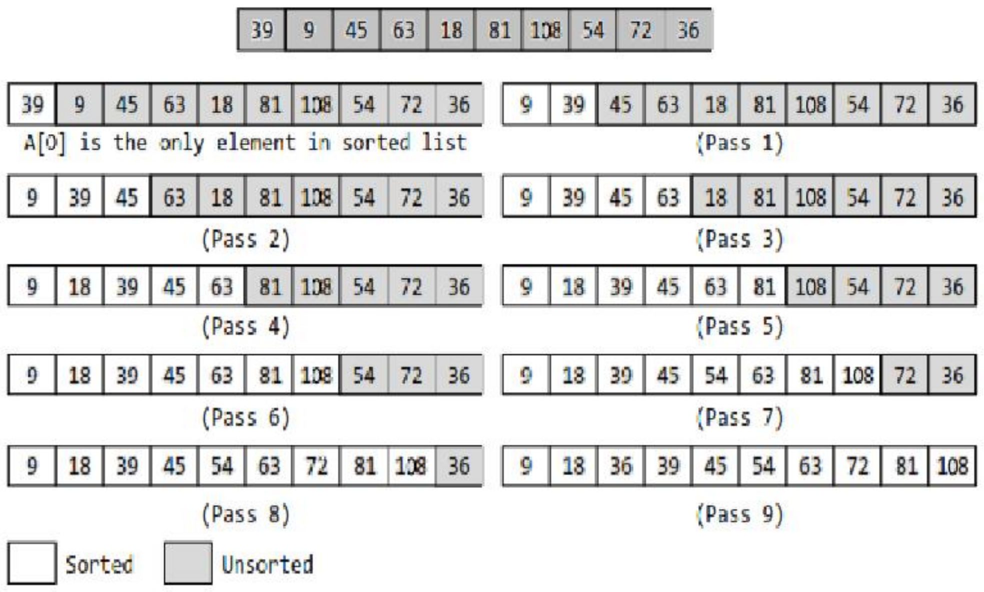
Insertion sort is less efficient as compared to other more advanced algorithms such as quick sort, heap sort, and merge sort.

**Technique**

- Insertion sort works as follows:
- The array of values to be sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are n elements in the array. Initially, the element with index 0 is in the sorted set. Rest of the elements are in the unsorted set.
- The first element of the unsorted partition has array index 1.
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

**Example** Consider an array of integers given below. We will sort the values in the array using insertion sort.

***Solution***

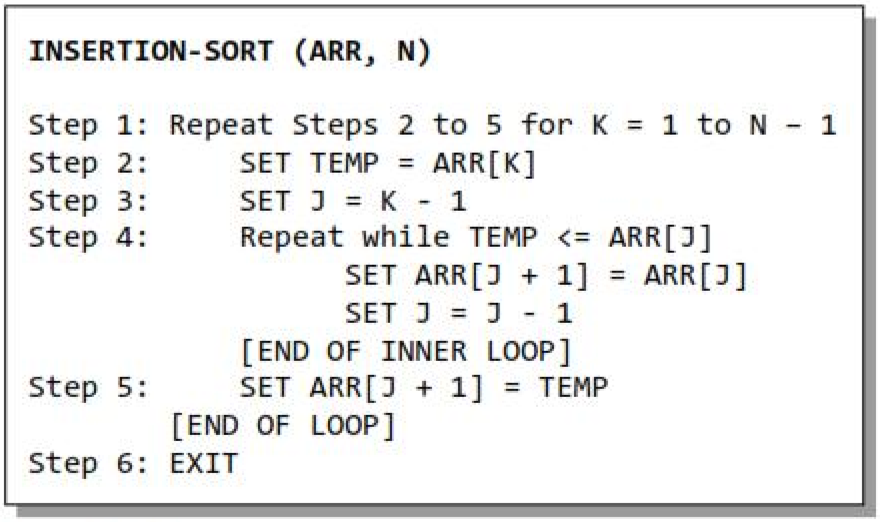


Initially, A[0] is the only element in the sorted set.
In Pass 1, A[1] will be placed either before or after A[0], so that the array A is sorted.
In Pass 2, A[2] will be placed either before A[0], in between A[0] and A[1], or after A[1].
In Pass 3, A[3] will be placed in its proper place. In Pass N–1, A[N–1] will be placed in its proper place to keep the array sorted.

To insert an element A[K] in a sorted list A[0], A[1], ..., A[K–1], we need to compare A[K] with A[K–1], then with A[K–2], A[K–3], and so on until we meet an element A[J] such that A[J] <= A[K]. In order to insert A[K] in its correct position, we need to move elements A[K–1], A[K–2], ..., A[J] by one position and then A[K] is inserted at the (J+1) location.

The algorithm for insertion sort is given in the following Fig.

In the algorithm,
Step 1 executes a for loop which will be repeated for each element in the array.
In Step 2, we store the value of the K element in TEMP.
In Step 3, we set the J index in the array.
In Step 4, a for loop is executed that will create space for the new element from the unsorted list to be stored in the list of sorted elements.
Finally, in Step 5, the element is stored at the (J+1)th location.



```
INSERTION-SORT (ARR, N)

Step 1: Repeat Steps 2 to 5 for K = 1 to N - 1
Step 2:       SET TEMP = ARR[K]
Step 3:       SET J = K - 1
Step 4:       Repeat while TEMP <= ARR[J]
                     SET ARR[J + 1] = ARR[J]
                     SET J = J - 1
              [END OF INNER LOOP]
Step 5:       SET ARR[J + 1] = TEMP
          [END OF LOOP]
Step 6: EXIT
```

**Figure 14.7** Algorithm for insertion sort

**Complexity of Insertion Sort**

For insertion sort:

The best case occurs when the array is already sorted. In this case, the running time of the algorithm has a linear running time (i.e., O(n)). This is because, during each iteration, the first element from the unsorted set is compared only with the last element of the sorted set of the array.

The worst case of the insertion sort algorithm occurs when the array is sorted in the reverse order. In the worst case, the first element of the unsorted set has to be compared with almost every element in the sorted set. Therefore, in the worst case, insertion sort has a quadratic running time (i.e., O($n^2$)).

The average case also has a quadratic running time(i.e., O($n^2$))..

**C Program for Insertion Sort:**

```
#include <stdio.h>
void insertionsort(int a[20], int n)
{
   int temp, j, k;
   for (k=1; k<n; k++)
   {
      temp=a[k];
      j=k-1;
      while((temp<=a[j]) && (j>=0))
      {
         a[j+1]=a[j];
         j--;
      }
      a[j+1]=temp;
   }
}

int main()
{
   int a[20],n,i;
    printf("Enter no. of elements:");
    scanf("%d",&n);
```

**Output:**

```
Enter no. of elements: 5
Enter elements: 45    11     2     56    7
Sorted  elements: 2    7     11     45    56
```

## 9.2 Bubble Sort

Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment (in case of arranging elements in ascending order).

In *bubble sorting*, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged. This process will continue till the list of unsorted elements exhausts.

This procedure of sorting is called bubble sorting because elements 'bubble' to the top of the list. At the end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).

## Technique

The basic methodology of the working of bubble sort is given as follows:

(a) In Pass 1, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–2] is compared with A[N–1]. Pass 1 involves n–1 comparisons and places the biggest element at the highest index of the array.
(b) In Pass 2, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–3] is compared with A[N–2]. Pass 2 involves n–2 comparisons and places the second biggest element at the second highest index of the array.
(c) In Pass 3, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–4] is compared with A[N–3]. Pass 3 involves n–3 comparisons and places the third biggest element at the third highest index of the array.
(d) In Pass n–1, A[0] and A[1] are compared so that A[0]<A[1]. After this step, all the elements of the array are arranged in ascending order.

**Example 14.2** To discuss bubble sort in detail, let us consider an array A[] that has the following elements:
A[] = {30, 52, 29, 87, 63, 27, 19, 54}

**Pass 1:**
(a) Compare 30 and 52. Since 30 < 52, no swapping is done.
(b) Compare 52 and 29. Since 52 > 29, swapping is done.
30, **29, 52**, 87, 63, 27, 19, 54
(c) Compare 52 and 87. Since 52 < 87, no swapping is done.
(d) Compare 87 and 63. Since 87 > 63, swapping is done.
30, 29, 52, **63, 87**, 27, 19, 54
(e) Compare 87 and 27. Since 87 > 27, swapping is done.
30, 29, 52, 63, **27, 87**, 19, 54
(f) Compare 87 and 19. Since 87 > 19, swapping is done.
30, 29, 52, 63, 27, **19**, **87**, 54
(g) Compare 87 and 54. Since 87 > 54, swapping is done.
30, 29, 52, 63, 27, 19, **54, 87**
Observe that after the end of the first pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted.

**Pass 2:**
(a) Compare 30 and 29. Since 30 > 29, swapping is done.
**29, 30**, 52, 63, 27, 19, 54, 87
(b) Compare 30 and 52. Since 30 < 52, no swapping is done.
(c) Compare 52 and 63. Since 52 < 63, no swapping is done.
(d) Compare 63 and 27. Since 63 > 27, swapping is done.
29, 30, 52, **27, 63**, 19, 54, 87
(e) Compare 63 and 19. Since 63 > 19, swapping is done.
29, 30, 52, 27, **19, 63**, 54, 87
(f) Compare 63 and 54. Since 63 > 54, swapping is done.

29, 30, 52, 27, 19, **54, 63**, 87

Observe that after the end of the second pass, the second largest element is placed at the second highest index of the array. All the other elements are still unsorted.

**Pass 3:**

(a) Compare 29 and 30. Since 29 < 30, no swapping is done.
(b) Compare 30 and 52. Since 30 < 52, no swapping is done.
(c) Compare 52 and 27. Since 52 > 27, swapping is done.
29, 30, **27, 52**, 19, 54, 63, 87
(d) Compare 52 and 19. Since 52 > 19, swapping is done.
29, 30, 27**, 19, 52**, 54, 63, 87
(e) Compare 52 and 54. Since 52 < 54, no swapping is done.

Observe that after the end of the third pass, the third largest element is placed at the third highest index of the array. All the other elements are still unsorted.

**Pass 4:**

(a) Compare 29 and 30. Since 29 < 30, no swapping is done.
(b) Compare 30 and 27. Since 30 > 27, swapping is done.
29, **27, 30**, 19, 52, 54, 63, 87
(c) Compare 30 and 19. Since 30 > 19, swapping is done.
29, 27, **19, 30**, 52, 54, 63, 87
(d) Compare 30 and 52. Since 30 < 52, no swapping is done.

Observe that after the end of the fourth pass, the fourth largest element is placed at the fourth highest index of the array. All the other elements are still unsorted.

**Pass 5:**

(a) Compare 29 and 27. Since 29 > 27, swapping is done.
**27, 29,** 19, 30, 52, 54, 63, 87
(b) Compare 29 and 19. Since 29 > 19, swapping is done.
27, **19, 29**, 30, 52, 54, 63, 87
(c) Compare 29 and 30. Since 29 < 30, no swapping is done.

Observe that after the end of the fifth pass, the fifth largest element is placed at the fifth highest index of the array. All the other elements are still unsorted.

**Pass 6:**

(a) Compare 27 and 19. Since 27 > 19, swapping is done.
**19, 27,** 29, 30, 52, 54, 63, 87
(b) Compare 27 and 29. Since 27 < 29, no swapping is done.

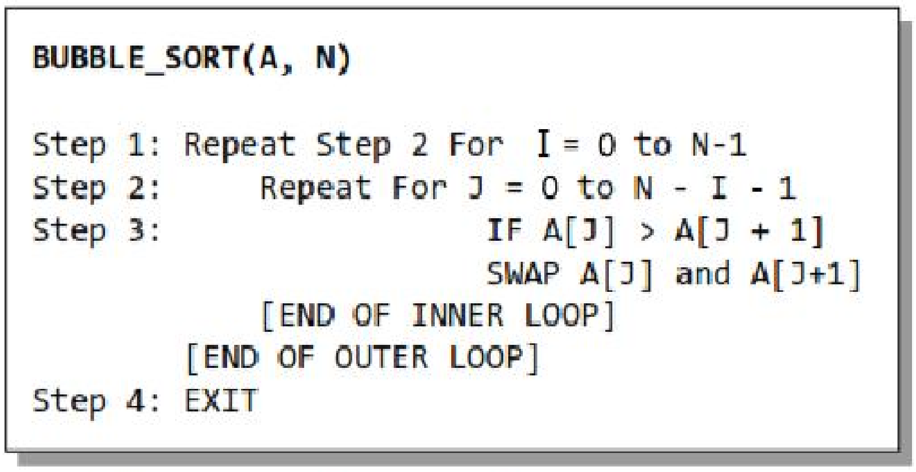
Observe that after the end of the sixth pass, the sixth largest element is placed at the sixth largest index of the array. All the other elements are still unsorted.

**Pass 7:**

(a) Compare 19 and 27. Since 19 < 27, no swapping is done.

Figure 14.6 shows the algorithm for bubble sort.

In this algorithm, the outer loop is for the total number of passes which is N–1. The inner loop will be executed for every pass. However, the frequency of the inner loop will decrease with every pass because after every pass, one element will be in its correct position. Therefore, for every pass, the inner loop will be executed N–I times, where N is the number of elements in the array and I is the count of the pass.



```
BUBBLE_SORT(A, N)

Step 1: Repeat Step 2 For  I = 0 to N-1
Step 2:      Repeat For J = 0 to N - I - 1
Step 3:                    IF A[J] > A[J + 1]
                           SWAP A[J] and A[J+1]
             [END OF INNER LOOP]
         [END OF OUTER LOOP]
Step 4: EXIT
```

**Figure** Algorithm for bubble sort

## Complexity of Bubble Sort

The complexity of any sorting algorithm depends upon the number of comparisons.
In bubble sort, there are N–1 passes in total.
In the first pass, N–1 comparisons are made to place the highest element in its correct position.
Then, in Pass 2, there are N–2 comparisons and the second highest element is placed in its position.
Therefore, to compute the complexity of bubble sort, we need to calculate the total number of comparisons. It can be given as:

$f(n) = (n – 1) + (n – 2) + (n – 3) + ..... + 3 + 2 + 1$
$f(n) = n (n – 1)/2$
$f(n) = O(n2)$

Therefore, the complexity of bubble sort algorithm is $O(n^2)$.

It means the time required to execute bubble sort is proportional to $n^2$ , where n is the total number of elements in the array.

## Program for Bubble sort:

```
#include <stdio.h>
void bubblesort( int a[10], int n)
{
        for(i=0;i<n;i++)
        {
                for(j=0;j<n–i–1;j++)
                {
                        if(a[j] > a[j+1])
                        {
                                temp = a[j];
                                a[j] = a[j+1];
                                a[j+1] = temp;
                        }
                }
        }
}

int main()
{
        int i, n, a[10];
        printf("\n Enter the number of elements in the array : ");
                scanf("%d", &n);
         printf("\n Enter the elements: ");
        for(i=0;i<n;i++)
                scanf("%d", &a[i]);

        bubblesort(a,n);

        printf("\n The array sorted in ascending order is :\n");
        for(i=0;i<n;i++)
                printf("%d\t", arr[i]);

        return 0;
}
```

**Output:**

```
Enter the number of elements in the array : 5
Enter the elements: 34      5      2      78      1
The array sorted in ascending order is : 1      2      5      34      78
```

**Working:**

```
34      5      2      78      1

Pass 1
        34      5      2      78      1
        5      43      2      78      1
        5      2      43      78      1
        5      2      43      78      1
```

| | | | | |
|---|---|---|---|---|
| 5 | 2 | 43 | 1 | 78 |

Pass 2

| | | | | |
|---|---|---|---|---|
| **5** | **2** | 43 | 1 | 78 |
| 2 | **5** | **43** | 1 | 78 |
| 2 | 5 | **43** | **1** | 78 |
| 2 | 5 | 1 | 43 | 78 |

Pass 3

| | | | | |
|---|---|---|---|---|
| **2** | **5** | 1 | 43 | 78 |
| 2 | **5** | **1** | 43 | 78 |
| 2 | 1 | 5 | 43 | 78 |

Pass 4

| | | | | | |
|---|---|---|---|---|---|
| **2** | **1** | 5 | 43 | 78 | |
| 1 | 2 | 5 | 43 | 78 | ---→sorted |

## 9.3 Selection Sort

- ➢ Selection sort is a simple sorting algorithm.
- ➢ Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.
- ➢ This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end.
- ➢ Selection sort is a sorting algorithm that has a quadratic running time complexity of O(n2)
- ➢ It inefficient to be used on large lists
- ➢ Selection sort performs worse than insertion sort algorithm better than bubble sort.

### Technique

Consider an array ARR with N elements.

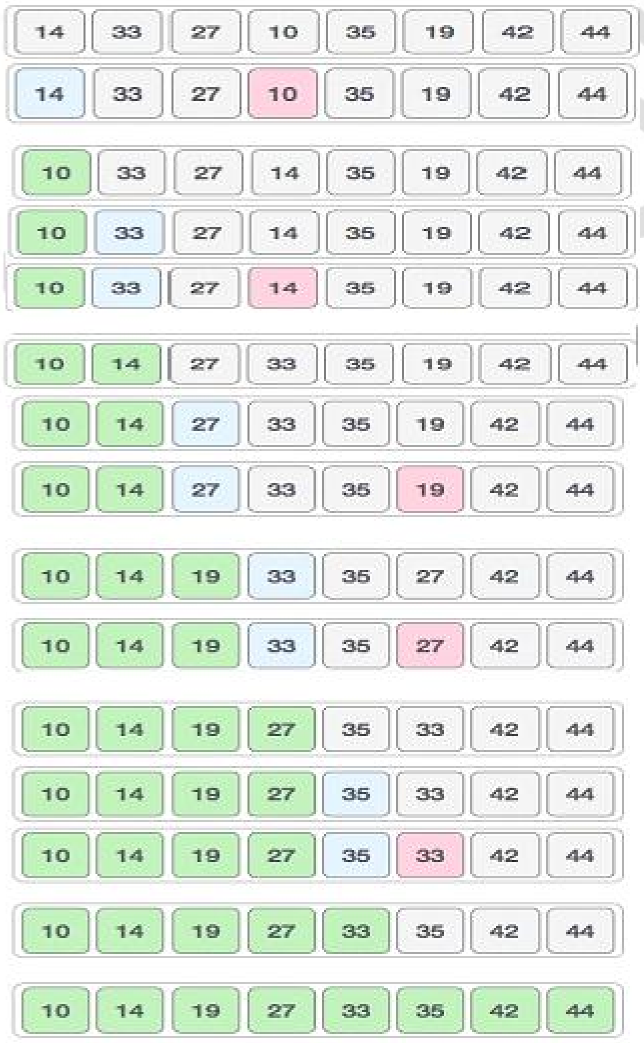
Selection sort works as follows:

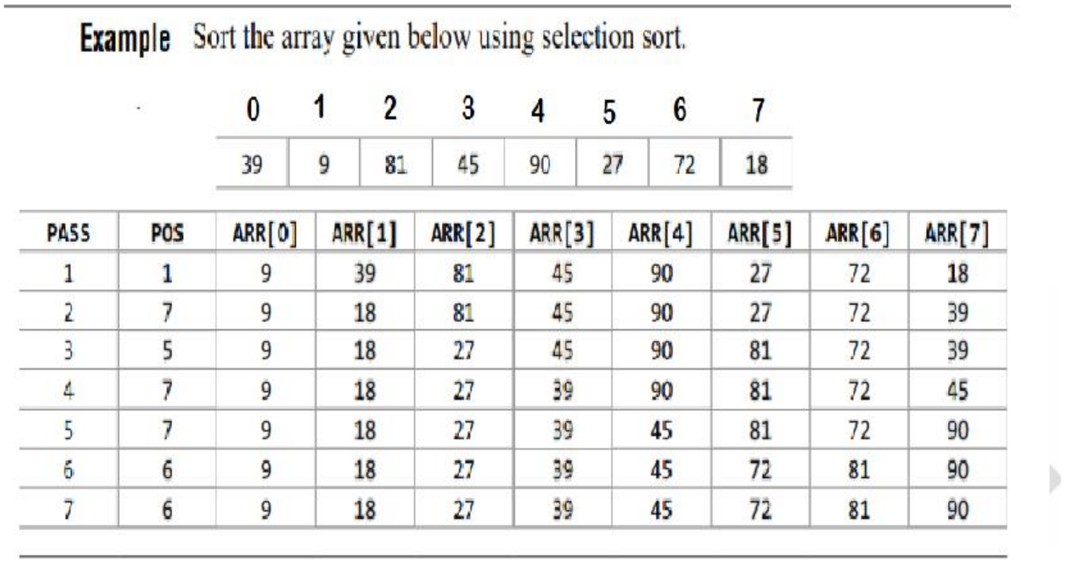
- First find the smallest value in the array and place it in the first position.
- Then, find the second smallest value in the array and place it in the second position.
- Repeat this procedure until the entire array is sorted.

Therefore,

- In Pass 1, find the position POS of the smallest value in the array and then swap ARR[POS] and ARR[0]. Thus, ARR[0] is sorted.
- In Pass 2, find the position POS of the smallest value in sub-array of N–1 elements. Swap ARR[POS] with ARR[1]. Now, ARR[0] and ARR[1] is sorted.
- In Pass N–1, find the position POS of the smaller of the elements ARR[N–2] and ARR[N–1]. Swap ARR[POS] and ARR[N–2] so that ARR[0], ARR[1], ..., ARR[N–1] is sorted.

## Working:



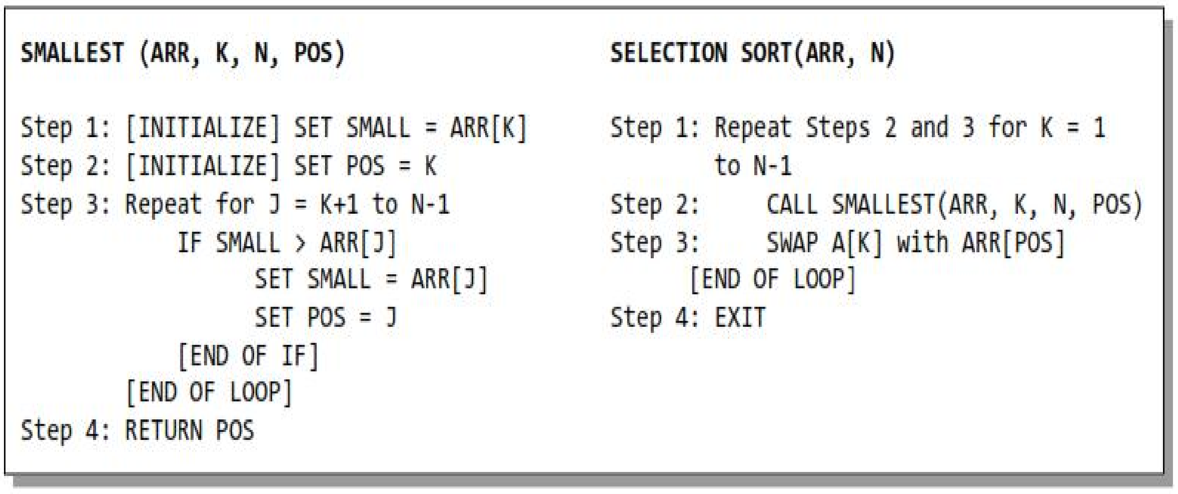


**Example** Sort the array given below using selection sort.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 39 | 9 | 81 | 45 | 90 | 27 | 72 | 18 |

| PASS | POS | ARR[0] | ARR[1] | ARR[2] | ARR[3] | ARR[4] | ARR[5] | ARR[6] | ARR[7] |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 9 | 39 | 81 | 45 | 90 | 27 | 72 | 18 |
| 2 | 7 | 9 | 18 | 81 | 45 | 90 | 27 | 72 | 39 |
| 3 | 5 | 9 | 18 | 27 | 45 | 90 | 81 | 72 | 39 |
| 4 | 7 | 9 | 18 | 27 | 39 | 90 | 81 | 72 | 45 |
| 5 | 7 | 9 | 18 | 27 | 39 | 45 | 81 | 72 | 90 |
| 6 | 6 | 9 | 18 | 27 | 39 | 45 | 72 | 81 | 90 |
| 7 | 6 | 9 | 18 | 27 | 39 | 45 | 72 | 81 | 90 |

## Algorithm



```
SMALLEST (ARR, K, N, POS)

Step 1: [INITIALIZE] SET SMALL = ARR[K]
Step 2: [INITIALIZE] SET POS = K
Step 3: Repeat for J = K+1 to N-1
            IF SMALL > ARR[J]
                  SET SMALL = ARR[J]
                  SET POS = J
            [END OF IF]
        [END OF LOOP]
Step 4: RETURN POS
```

```
SELECTION SORT(ARR, N)

Step 1: Repeat Steps 2 and 3 for K = 1
        to N-1
Step 2:     CALL SMALLEST(ARR, K, N, POS)
Step 3:     SWAP A[K] with ARR[POS]
      [END OF LOOP]
Step 4: EXIT
```

**Figure** Algorithm for selection sort

## Complexity of Selection Sort

Selection sort is a sorting algorithm that is independent of the original order of elements in the array.

➢In Pass 1, selecting the element with the smallest value requires n–1 comparisons. Then, the smallest value is swapped with the element in the first position.

➢In Pass 2, selecting the second smallest value requires scanning the remaining n – 1 elements and so on.

➢Therefore,
$(n – 1) + (n – 2) + ... + 2 + 1= n(n – 1) / 2 = O(n^2)$ comparisons

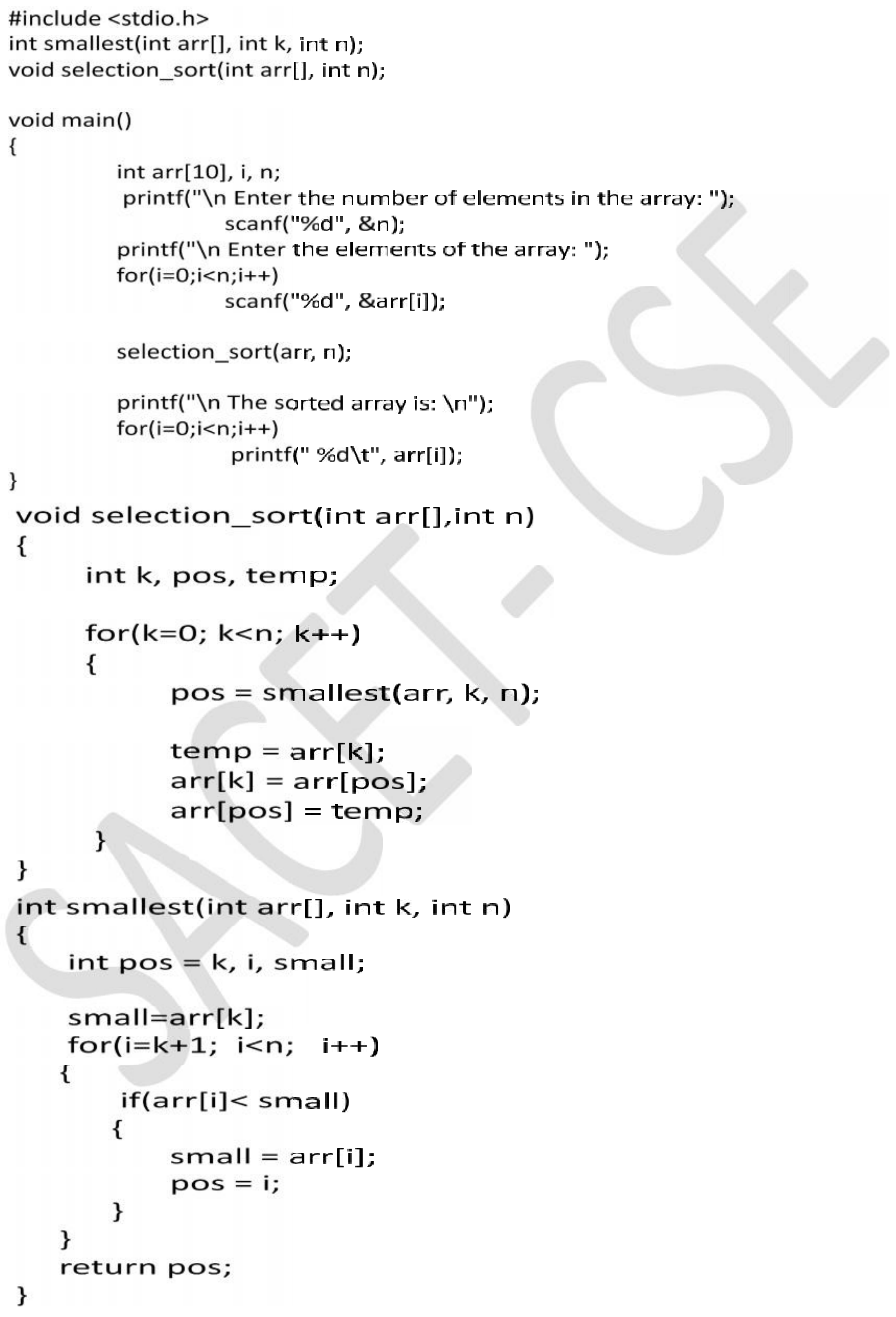
## Advantages of Selection Sort

➢ It is simple and easy to implement.

➢ It can be used for small data sets.

➢ It is 60 per cent more efficient than bubble sort.

## Program on Selection Sort:



```
#include <stdio.h>
int smallest(int arr[], int k, int n);
void selection_sort(int arr[], int n);

void main()
{
            int arr[10], i, n;
             printf("\n Enter the number of elements in the array: ");
                        scanf("%d", &n);
            printf("\n Enter the elements of the array: ");
            for(i=0;i<n;i++)
                        scanf("%d", &arr[i]);

            selection_sort(arr, n);

            printf("\n The sorted array is: \n");
            for(i=0;i<n;i++)
                         printf(" %d\t", arr[i]);
}
 void selection_sort(int arr[],int n)
 {
        int k, pos, temp;

        for(k=0; k<n; k++)
        {
                pos = smallest(arr, k, n);

                temp = arr[k];
                arr[k] = arr[pos];
                arr[pos] = temp;
         }
 }
 int smallest(int arr[], int k, int n)
 {
      int pos = k, i, small;

      small=arr[k];
      for(i=k+1;  i<n;   i++)
     {
          if(arr[i]< small)
         {
              small = arr[i];
              pos = i;
         }
     }
     return pos;
 }
```

## Output:

```
Enter number of elements in the array: 5

Enter the elements of the array: 25 34 1 6 5

The sorted array is: 1 5 6 25 34
```

## 9.4. MergeSort

Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm.

- ***Divide means*** partitioning the n-element array to be sorted into two sub-arrays of n/2 elements. If A is an array containing zero or one element, then it is already sorted. However, if there are more elements in the array, divide A into two sub-arrays, A1 and A2, each containing about half of the elements of A.
- ***Conquer means sorting the two sub-arrays recursively using merge sort.***
- ***Combine means merging the two sorted sub-arrays of size*** n/2 to produce the sorted array of n elements.

Merge sort algorithm focuses on two main concepts to improve its performance (running time):

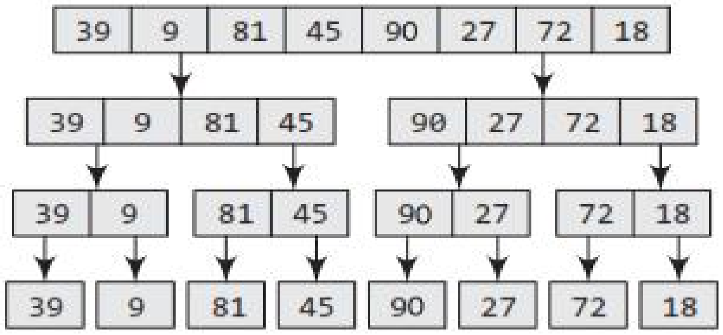
- A smaller list takes fewer steps and thus less time to sort than a large list.
- As number of steps is relatively less, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists.

The basic steps of a merge sort algorithm are as follows:

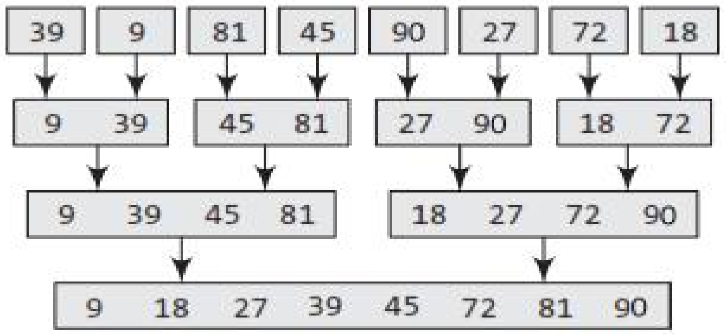
- If the array is of length 0 or 1, then it is already sorted.
- Otherwise, divide the unsorted array into two sub-arrays of about half the size.
- Use merge sort algorithm recursively to sort each sub-array.
- Merge the two sub-arrays to form a single sorted list.

**Example** Sort the array given below using merge sort.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 39 | 9 | 81 | 45 | 90 | 27 | 72 | 18 |



(Divide and Conquer the array)



(Combine the elements to form a sorted array)

## Algorithm:

```
MERGE_SORT(ARR, BEG, END)
Step 1: IF BEG < END
            SET MID = (BEG + END)/2
            CALL MERGE_SORT (ARR, BEG, MID)
            CALL MERGE_SORT (ARR, MID + 1, END)
            MERGE (ARR, BEG, MID, END)
        [END OF IF]
Step 2: END
```

**Figure 14.9** Algorithm for merge sort

```
MERGE (ARR, BEG, MID, END)

Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
Step 2: Repeat while (I <= MID) AND (J<=END)
              IF ARR[I] < ARR[J]
                    SET TEMP[INDEX] = ARR[I]
                    SET I = I + 1
              ELSE
                    SET TEMP[INDEX] = ARR[J]
                    SET J = J + 1
              [END OF IF]
              SET INDEX = INDEX + 1
        [END OF LOOP]
Step 3: [Copy the remaining elements of right sub-array, if any]
              IF I > MID
                 Repeat while J <= END
                    SET TEMP[INDEX] = ARR[J]
                    SET INDEX = INDEX + 1, SET J = J + 1
                 [END OF LOOP]
        [Copy the remaining elements of left sub-array, if any]
              ELSE
                 Repeat while I <= MID
                    SET TEMP[INDEX] = ARR[I]
                    SET INDEX = INDEX + 1, SET I = I + 1
                 [END OF LOOP]
              [END OF IF]
Step 4: [Copy the contents of TEMP back to ARR] SET K=0
Step 5: Repeat while K < INDEX
                 SET ARR[K] = TEMP[K]
                 SET K = K + 1
        [END OF LOOP]
Step 6: END
```

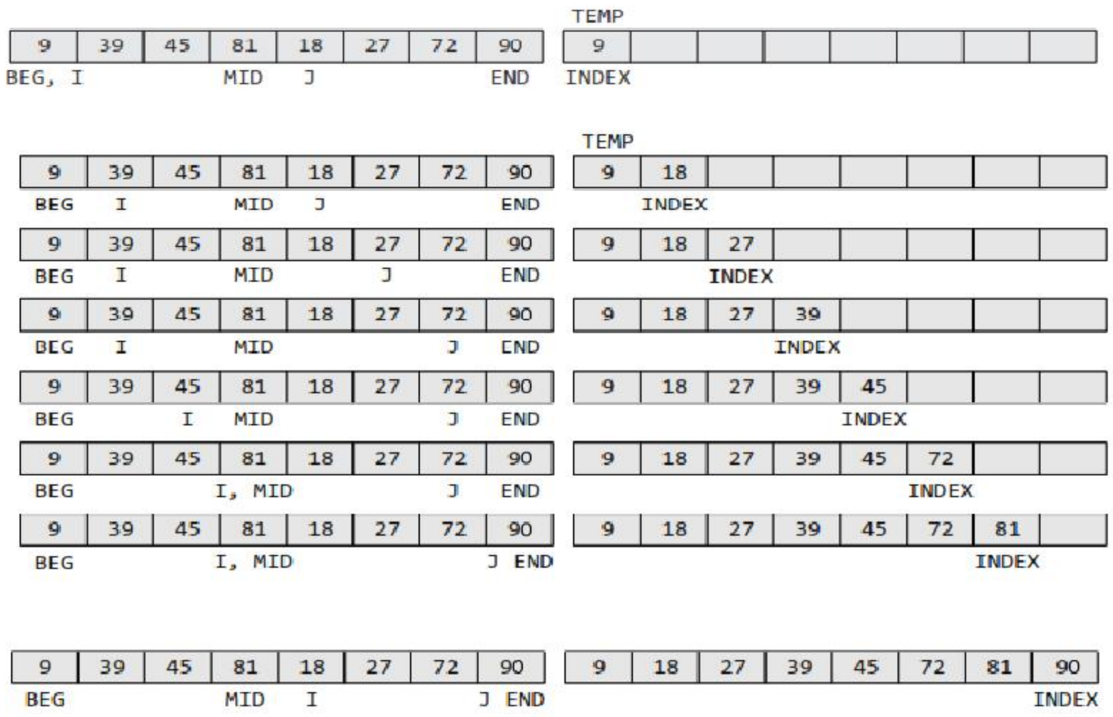
## Merge Sort Program:

```
#include <stdio.h>
#define size 100
void merge(int a[], int, int, int);
void merge_sort(int a[],int, int);
void main()
{
            int arr[size], i, n;
            printf("\n Enter the number of elements in the array : ");
            scanf("%d", &n);
            printf("\n Enter the elements of the array: ");
            for(i=0;i<n;i++)
                        scanf("%d", &arr[i]);

            merge_sort(arr, 0, n-1);

            printf("\n The sorted array is: \n");
            for(i=0;i<n;i++)
                        printf(" %d\t", arr[i]);
      }
     void merge(int arr[], int beg, int mid, int end)
     {
          int i=beg, j=mid+1, index=beg, temp[size], k;
          while((i<=mid) && (j<=end))
          {
                if(arr[i] < arr[j])
                {
           temp[index] = arr[i];
           i++;
                }
                else
                {
                     temp[index] = arr[j];
                     j++;
                }
                 index++;
           }
     if(i>mid)
      {
           while(j<=end)
           {
                   temp[index] = arr[j];
                   j++;
                   index++;
             }
      }
      else
      {
          while(i<=mid)
```

```
        {
            temp[index] = arr[i];
            i++;
            index++;
         }
     }
     for(k=beg;k<index;k++)
         arr[k] = temp[k];
   }
   void merge_sort(int arr[], int beg, int end)
   {
        int mid;
         if(beg<end)
        {
                 mid = (beg+end)/2;
                 merge_sort(arr, beg, mid);
                 merge_sort(arr, mid+1, end);
                 merge(arr, beg, mid, end);
        }
   }
```



## Time Complexity of Merge Sort:

The running time of merge sort in the average case and the worst case can be given as O(n log n).

Although merge sort has an optimal time complexity, it needs an additional space of O(n) for the temporary array TEMP.

## 9.5 Radix Sort

Radix sort is a linear sorting algorithm for integers and uses the same concept for sorting names in alphabetical order.

When we have a list of sorted names, the *radix* is 26 (or 26 buckets) because there are 26 letters in the English alphabet. So radix sort is also known as bucket sort. Observe that words are first sorted according to the right most letter of the name. That is, 26 classes are used to arrange the names, where the first class stores the names that begin with A, the second class contains the names with B, and so on.During the second pass, names are grouped according to the second right most letter. This process is continued till the nth pass, where n is the length of the name with maximum number of letters.After every pass, all the names are collected in order of buckets. That is, first pick up the names in the first bucket that contains the names beginning with A. In the second pass, collect the names from the second bucket, and so on.

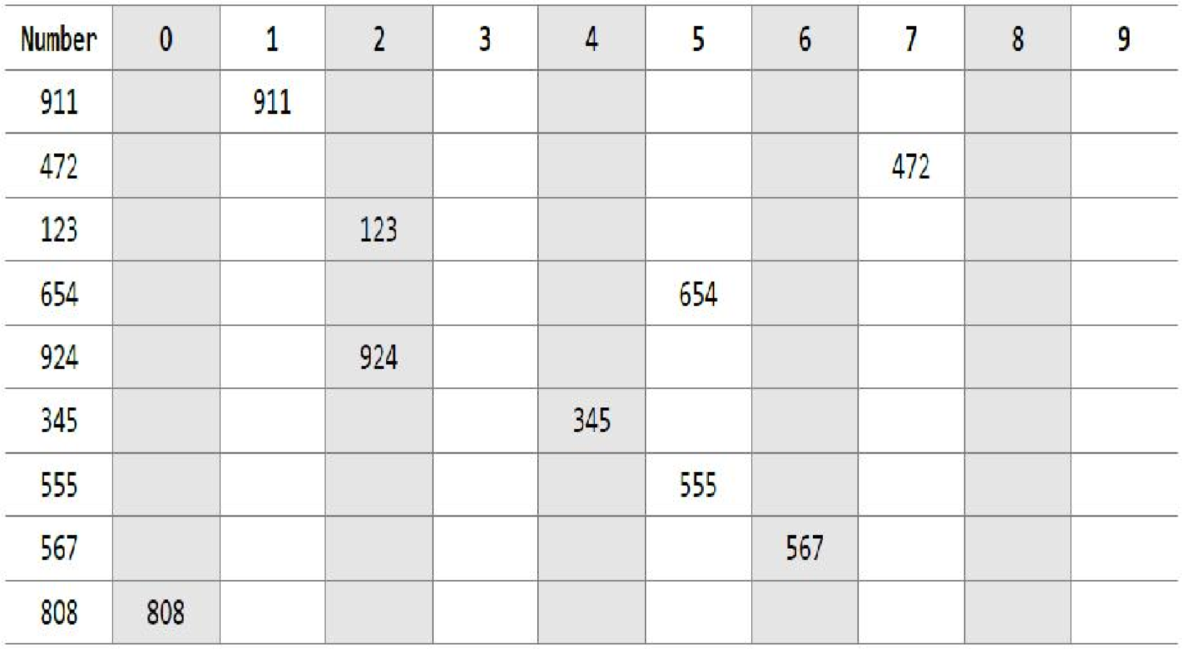
When radix sort is used on integers, sorting is done on each of the digits in the number. The sorting procedure proceeds by sorting the least significant to the most significant digit. While sorting the numbers, we have ten buckets, each for one digit (0, 1, 2, …, 9) The number of passes will depend on the length of the number having maximum number of digits.

**Example :** Sort the numbers given below using radix sort.

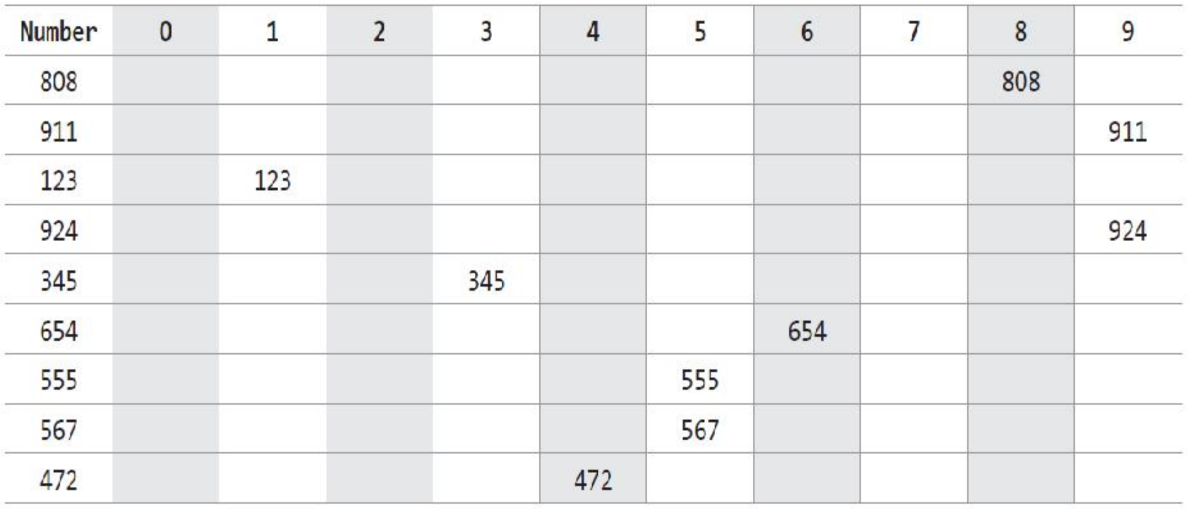
**345, 654, 924, 123, 567, 472, 555, 808, 911**

In the first pass, the numbers are sorted according to the digit at ones place. The buckets are pictured upside down as shown below.

| Number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 345 | | | | | | 345 | | | | |
| 654 | | | | | 654 | | | | | |
| 924 | | | | | 924 | | | | | |
| 123 | | | | 123 | | | | | | |
| 567 | | | | | | | | 567 | | |
| 472 | | | 472 | | | | | | | |
| 555 | | | | | | 555 | | | | |
| 808 | | | | | | | | | 808 | |
| 911 | | 911 | | | | | | | | |



| Number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 911 | | 911 | | | | | | | | |
| 472 | | | | | | | | 472 | | |
| 123 | | | 123 | | | | | | | |
| 654 | | | | | | 654 | | | | |
| 924 | | | 924 | | | | | | | |
| 345 | | | | | 345 | | | | | |
| 555 | | | | | | 555 | | | | |
| 567 | | | | | | | 567 | | | |
| 808 | 808 | | | | | | | | | |



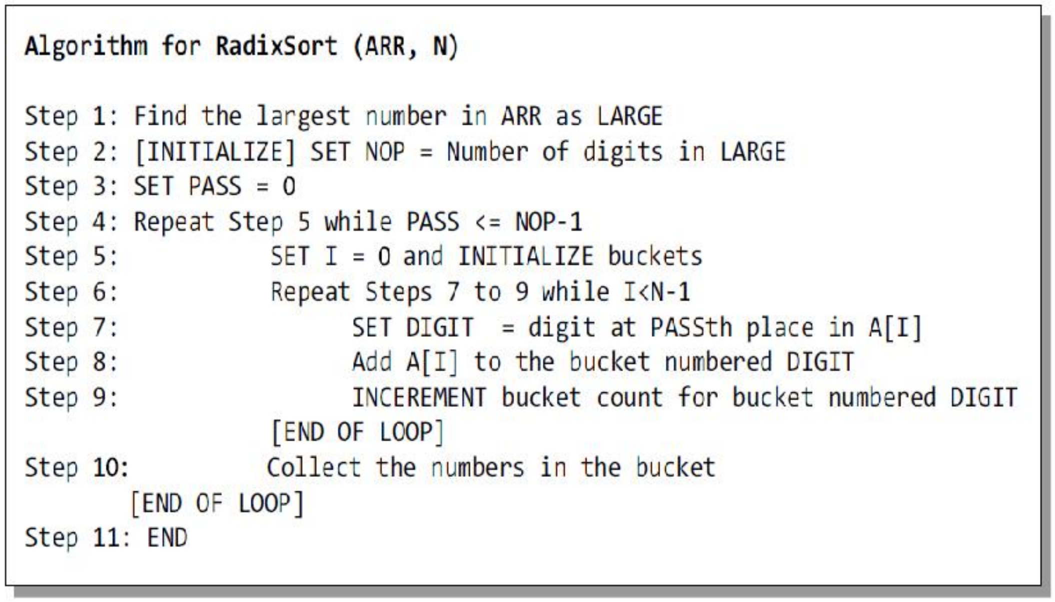
| Number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 808 | | | | | | | | | 808 | |
| 911 | | | | | | | | | | 911 |
| 123 | | 123 | | | | | | | | |
| 924 | | | | | | | | | | 924 |
| 345 | | | | 345 | | | | | | |
| 654 | | | | | | | 654 | | | |
| 555 | | | | | | 555 | | | | |
| 567 | | | | | | 567 | | | | |
| 472 | | | | | 472 | | | | | |

The numbers are collected bucket by bucket. The new list thus formed is the final sorted result.

After the third pass, the list can be given as

**123, 345, 472, 555, 567, 654, 808, 911, 924.**

## Radix Sort Algorithm



```
Algorithm for RadixSort (ARR, N)

Step 1: Find the largest number in ARR as LARGE
Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE
Step 3: SET PASS = 0
Step 4: Repeat Step 5 while PASS <= NOP-1
Step 5:           SET I = 0 and INITIALIZE buckets
Step 6:           Repeat Steps 7 to 9 while I<N-1
Step 7:                 SET DIGIT  = digit at PASSth place in A[I]
Step 8:                 Add A[I] to the bucket numbered DIGIT
Step 9:                 INCEREMENT bucket count for bucket numbered DIGIT
                  [END OF LOOP]
Step 10:          Collect the numbers in the bucket
        [END OF LOOP]
Step 11: END
```

**Figure 14.11** Algorithm for radix sort

## Complexity of Radix Sort

- ➢ To calculate the complexity of radix sort algorithm, assume that there are n numbers that have to be sorted and k is the number of digits in the largest number.
- ➢ In this case, the radix sort algorithm is called a total of k times. The inner loop is executed n times.
- ➢ Hence, the entire radix sort algorithm takes O(kn) time to execute.
- ➢ When radix sort is applied on a data set of finite size (very small set of numbers), then the algorithm runs in O(n) asymptotic time.

## Pros and Cons of Radix Sort

- ➢ Radix sort is a very simple algorithm. When programmed properly, radix sort is one of the fastest sorting algorithms for numbers or strings of letters.

But there are certain trade-offs for radix sort that can make it less preferable as compared to other sorting algorithms.

- ➢ Radix sort takes more space than other sorting algorithms. Besides the array of numbers, we need 10 buckets to sort numbers, 26 buckets to sort strings containing only characters, and at least 40 buckets to sort a string containing alphanumeric characters.
- ➢ Another drawback of radix sort is that the algorithm is dependent on digits or letters.

## . 9.6 Quick Sort

- Quick sort is a widely used sorting algorithm developed by C. A. R. Hoare
- in the average case  and best case it requires O(n log n) comparisons to sort an array of n elements.
- In the worst case, it has a quadratic running time given as O($n^2$).
- The quick sort algorithm is faster than other O(n log n) algorithms,
- Quick sort is also known as partition exchange sort.
- This algorithm works by using a divide-and-conquer strategy to divide a single unsorted array into two smaller sub-arrays.

The quick sort algorithm works as follows:

1. Select an element pivot from the array elements.
2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way). After such a partitioning, the pivot is placed in its final position. This is called the *partition* operation.
3. Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.)

## Technique

## Quick Sort Partition Algorithm

**Step 1** − Choose the lowest index value has pivot
**Step 2** − Take two variables ‘left’ and ‘right’ to point left and right of the list respectively
**Step 3** – ‘left’ points to the low index
**Step 4** – ‘right’ points to the high index
**Step 5** − while value at a[left] is less than pivot move ‘left’ forward
**Step 6** − while value at a[right] is greater than pivot move ‘right’ backward
**Step 7** − if both step 5 and step 6 does not match swap a[left] and a[right]
**Step 8** − if left ≥ right, swap pivot and a[right], where partition of the list occurs in such a way that all the elements in the left partition are less than pivot and all the elements of in the right partition are greater than pivot.

### Quick Sort Algorithm

**Step 1** − Make the left-most index value pivot
**Step 2** − partition the array using pivot value
**Step 3** − quicksort left partition recursively
**Step 4** − quicksort right partition recursively

## Example

We are given array of n integers to sort:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 80 | 60 | 50 | 7 | 30 | 100 |

**Pick Pivot Element**

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 40 | 20 | 10 | 80 | 60 | 50 | 7 | 30 | 100 |

**Partitioning Array**

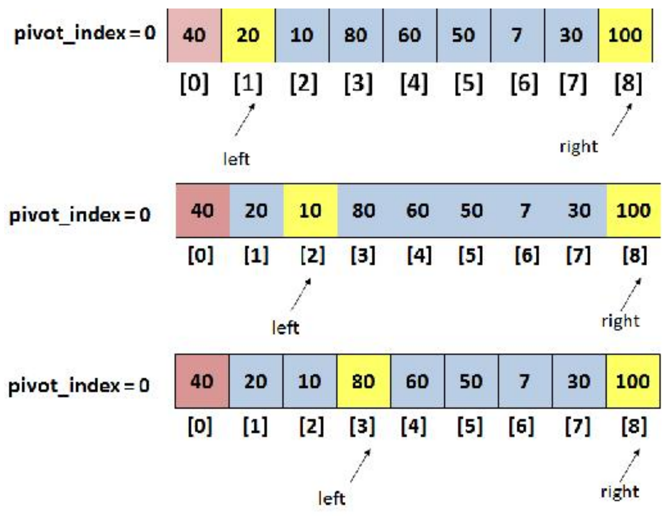
Given a pivot, partition the elements of the array such that the resulting array consists of:

1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

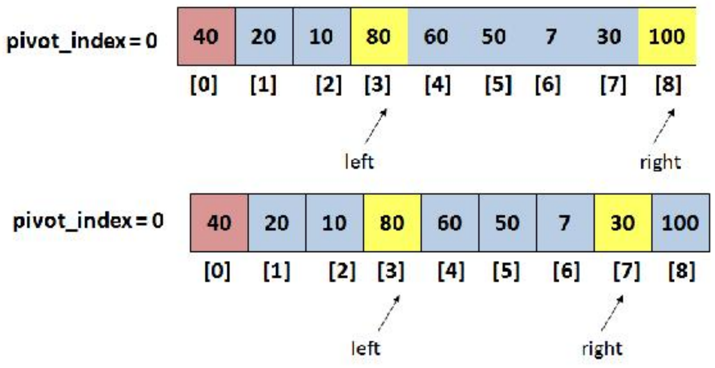
The sub-arrays are stored in the original array a[].

```
1.  While a[left] <= pivot
    left ++
2.  While a[right] > pivot
    right --
3.  If left < right
        swap a[left] and a[right]
4.  While left<right, go to 1.
5.  Swap a[right] and pivot
```

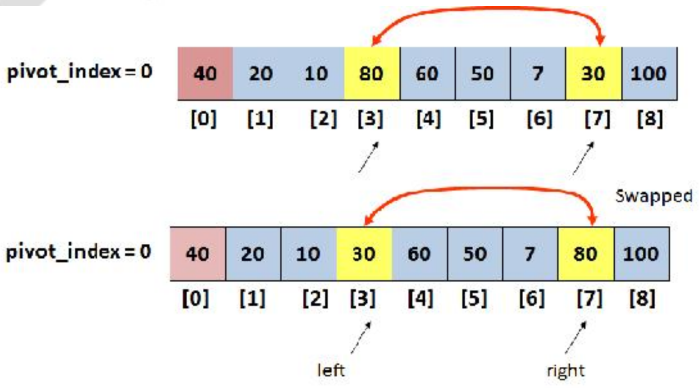
**Move left**



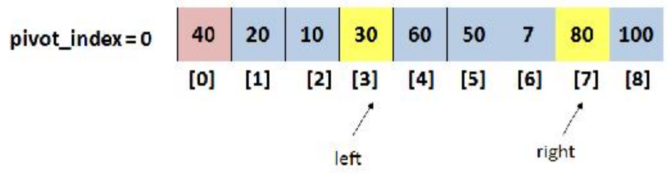
**Move right**

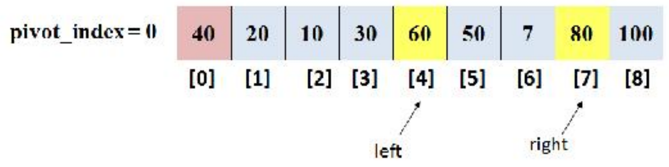


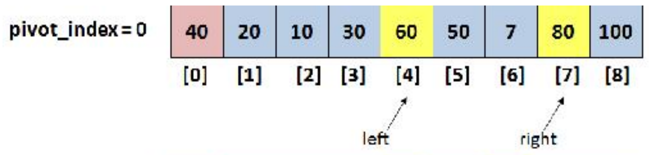
If (left < right) Swap a[left] and a[right],

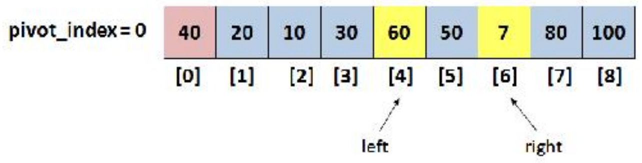


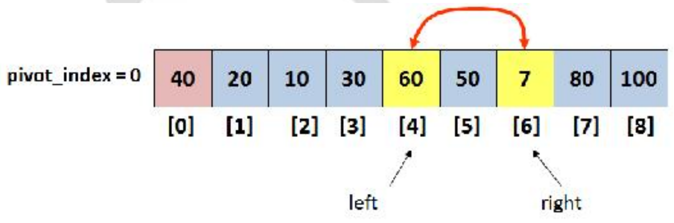
**Move left and right**

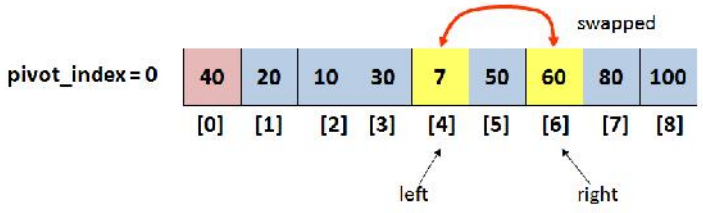


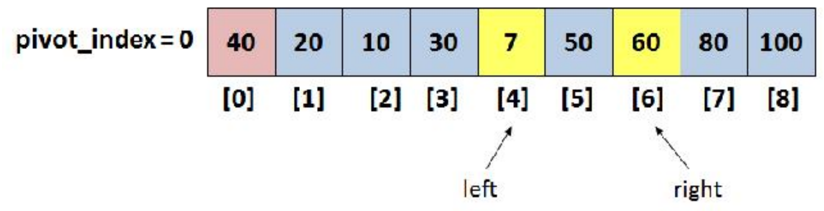


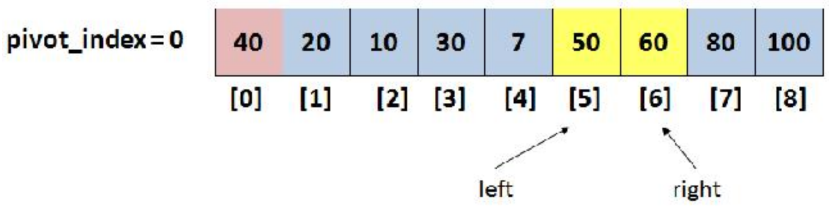


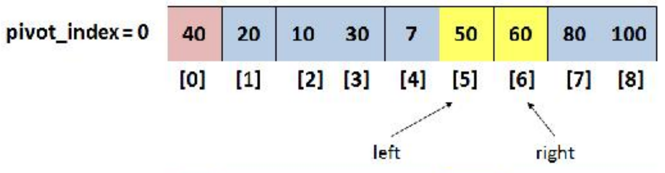


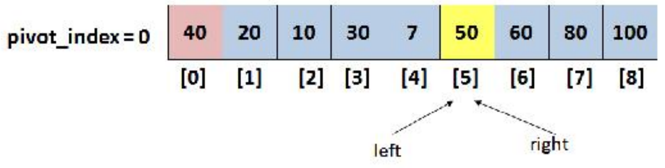


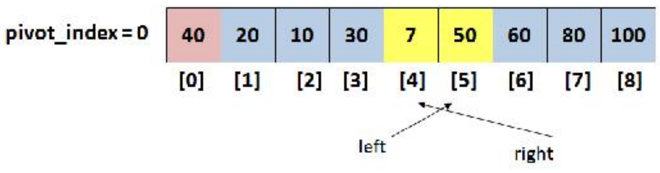




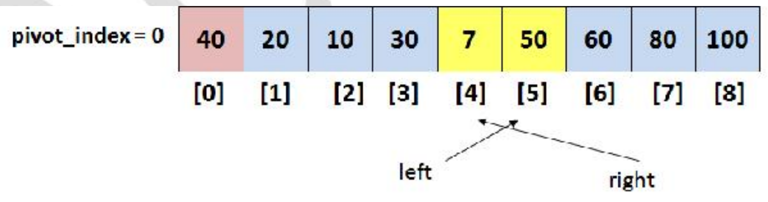


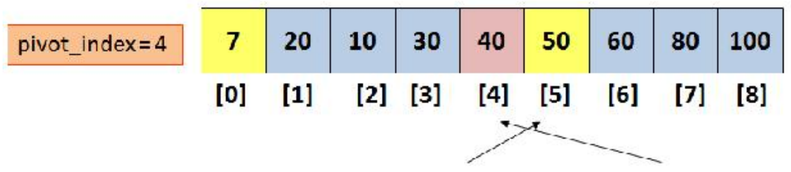




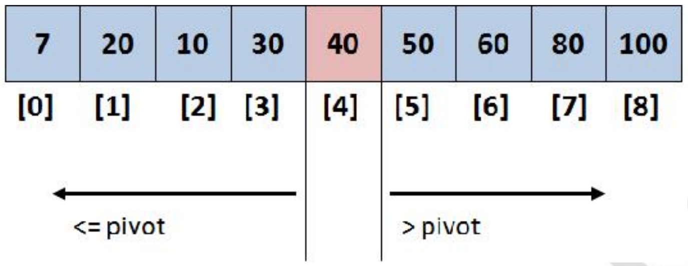


<u>**Here, Left>Right swap a[pivot] and a[right], this results in partitioning**</u>

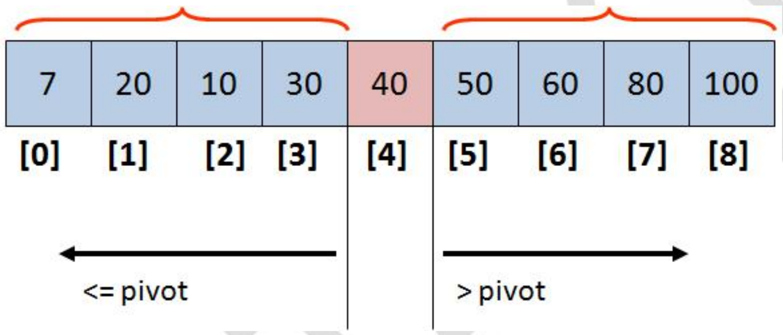




**Partition Result**



**Recursion: Quicksort Sub-arrays**



## Complexity of Quick Sort

- In the average case, the running time of quick sort can be given as **O(n log n).**
- In the best case, every time we partition the array, we divide the list into two nearly equal pieces. That is, the recursive call processes the sub-array of half the size. The resultant time is **O(n log n)** time.
- Practically, the efficiency of quick sort depends on the element which is chosen as the pivot.
- Its worst-case efficiency is given as **O($n^2$).** The worst case occurs when the array is already sorted (either in ascending or descending order) and the left-most element is chosen as the pivot.
- However, many implementations randomly choose the pivot element. The randomized version of the quick sort algorithm always has an algorithmic complexity of O(n log n).

## Quick Sort Program

```
#include<stdio.h>

int partition(int a[25],int beg,int end)
{
    int left, right, pivot, temp;
    pivot=beg;
    left=beg;
     right=end;


      while(left<right)
     {
                                    while(a[left]<=a[pivot]&& left<end)
                                          left++;
                                    while(a[right]>a[pivot])
                                          right--;
                                    if(left<right)
                                    {
                                          temp=a[left];
                                          a[left]=a[right];
                                          a[right]=temp;
                                    }
         }
          temp=a[pivot];
         a[pivot]=a[right];
         a[right]=temp;
    return right;
}
void quicksort(int a[25],int beg, int end)
{
    int j;
    if(beg<end)
    {
       j=partition(a,beg,end);
       quicksort(a,beg,j-1);
       quicksort(a,j+1,end);
    }
}
int main()
{
                                    int i, n, arr[25];
                                    printf("Enter no. of  elements : ");
                                    scanf("%d",&n);
                                     printf("Enter %d elements: ",n);
                                    for(i=0;i<n;i++)
                                           scanf("%d",&arr[i]);
                                    quicksort(arr,0,n-1);
                                    printf("The Sorted Order is: ");
                                    for(i=0;i<n;i++)
                                          printf(" %d",arr[i]);
                                    return 0;
}
```

## Output

```
Enter no. of elements: 8

Enter 8 elements: 67   54    34   12   7   2    16     4

The Sorted Order is: 2   4  7  12  16    34    54    67
```