UNIT-I

1. DATA STRUCTURES

1.1 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.

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.

Data structures are widely applied in the following areas:

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

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.

1.2 CLASSIFICATION OF DATA STRUCTURES:

Data structures are generally categorized into two classes: ***primitive*** and ***non-primitive*** data structures.

1. 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.

2. Non-Primitive Data Structures:

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: ***linear*** and ***non-linear*** data structures.

Linear and Non-linear Data Structures:

a) Linear Data Structure:

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. One way is to have to a linear relationship between elements by means of sequential memory locations. The other way is to have a linear relationship between elements by means of links. C supports a variety of data structures.

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*** (also known as the *subscript*).

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. 1.1.

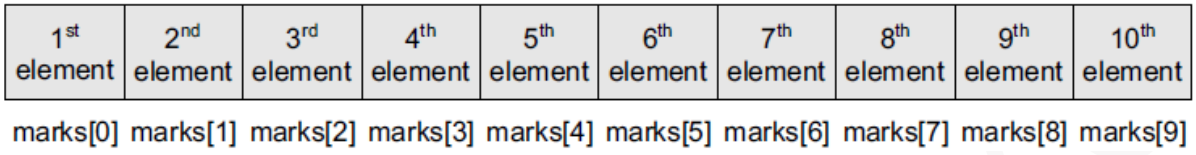


Figure 1.1 Memory representation of an array of 10 elements

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.

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. Therefore, in a linked list, every node contains the following two types of data:

- The value of the node or any other data that corresponds to that 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 1.2 shows a linked list of seven nodes.

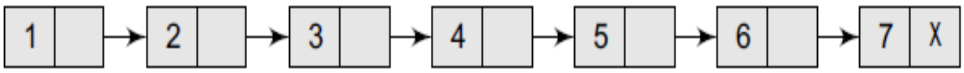


Figure 1.2 Simple linked list

Note:

Advantage: Easier to insert or delete data elements

Disadvantage: Slow search operation and requires more memory space

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 1.3 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.

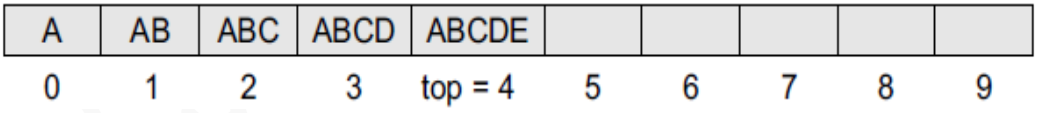


Figure 1.3 Array representation of a stack

In Fig. 1.3, 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: push, pop, and peep. The push operation adds an element to the top of the stack. The pop operation removes the element from the top of the stack. And the peep operation returns the value of the topmost element of the stack (without deleting it).

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

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

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. Like stacks, 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 Fig. 1.4.

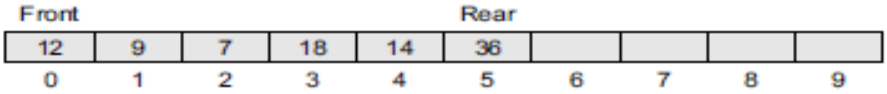


Figure 1.4 Array representation of a queue

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 Fig. 1.5.

Here, front = 0 and rear = 6. Every time a new element is to be added, we will repeat the same procedure.

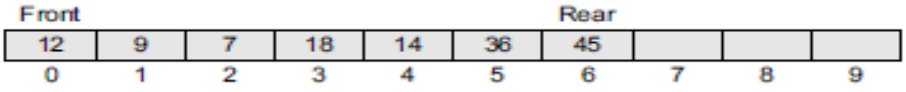


Figure 1.5 Queue after insertion of a new element

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 Fig. 1.6.

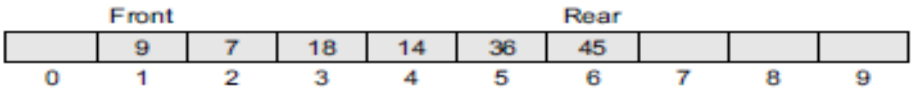


Figure 1.6 Queue after deletion of an element

However, 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. A queue is full when rear = MAX – 1, where MAX is the size of the queue, that is MAX specifies the maximum number of elements in the queue. Note that we have written MAX – 1 because the index starts from 0.

Similarly, 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. If front = NULL and rear = NULL, then there is no element in the queue.

b) Non-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.

TREES:

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 1.7 shows a binary tree, where R is the root node and T1 and T2 are the 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 it is called the right successor of R.

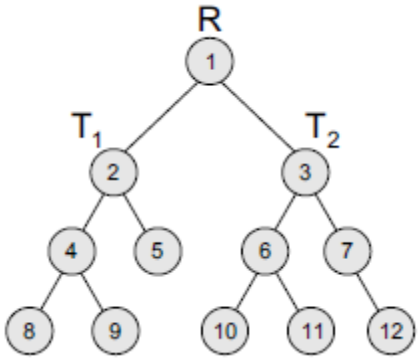


Figure 1.7 Binary tree

In Fig. 1.7, node 2 is the left child and node 3 is the right child of the root node 1. Note that the left sub-tree of the root node consists of the nodes 2, 4, 5, 8, and 9. Similarly, the right sub-tree of the root node consists of the nodes 3, 6, 7, 10, 11, and 12.

Note:

Advantage: Provides quick search, insert, and delete operations.

Disadvantage: Complicated deletion algorithm.

GRAPHS:

A graph is a non-linear data structure which is a collection of *vertices* (also called *nodes*) and *edges* that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of a purely parent-to-child relationship between tree nodes, any kind of complex relationships between the nodes can exist.

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 1.8 shows a graph with five nodes.

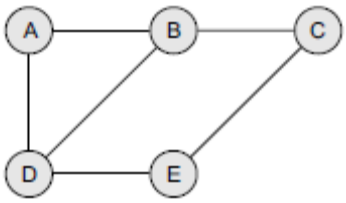


Figure 1.8 Graph

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.

Note that unlike trees, graphs do not have any root node. Rather, every node in the graph can be connected with every another node in the graph. When two nodes are connected via an edge, the two nodes are known as *neighbours*. For example, in Fig. 1.8, node A has two neighbours: B and D.

Note:

Advantage: Best models real-world situations

Disadvantage: Some algorithms are slow and very complex.

1.3 OPERATIONS ON DATA STRUCTURES:

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, or calculating the top three winners by arranging the participants' scores in descending order and then extracting the top three.

Merging:

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

Many a time, two or more operations are applied simultaneously in a given situation. For example, if we want to delete the details of a student whose name is X, then we first have to search the list of students to find whether the record of X exists or not and if it exists then at which location, so that the details can be deleted from that particular location.

1.4 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. For example, stacks and queues are perfect examples

of an ADT. We can implement both these ADTs using an array or a linked list. This demonstrates the 'abstract' nature of stacks and queues.

To further understand the meaning of an abstract data type, we will break the term into 'data type' and 'abstract', and then discuss their meanings.

Data type:

Data type of a variable is the set of values that the variable can take. We have already read the basic data types in C include int, char, float, and double. When we talk about a primitive type (built-in data type), we actually consider two things: a data item with certain characteristics and the permissible operations on that data. For example, an int variable can contain any whole-number value from –32768 to 32767 and can be operated with the operators +, –, *, and /. In other words, the operations that can be performed on a data type are an inseparable part of its identity. Therefore, when we declare a variable of an abstract data type (e.g., stack or a queue), we also need to specify the operations that can be performed on it.

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.

The end-user is not concerned about the details of how the methods carry out their tasks. They are only aware of the methods that are available to them and are only concerned about calling those methods and getting the results. They are not concerned about how they work.

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.

1.5 PRELIMINARIES OF ALGORITHMS:

The typical definition of algorithm is 'a formally defined procedure for performing some calculation'. An algorithm is basically a set of instructions that solve a problem. In general terms, 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. That is, a well-defined algorithm always provides an answer and is guaranteed to terminate.

Characteristics of an Algorithm:

1. **Input:** Externally we have to supply '0' or '1' input.
2. **Output:** After supplying of input we have to produce at least one output.
3. **Finiteness:** The algorithm must be terminate (or) end at some point.
4. **Definiteness:** We have to define clear and unambiguous statements.
5. **Effectiveness:** It should allow only necessary statements, need not to allow unnecessary statements.

Different approaches to designing an algorithm:

There are two main approaches to design an algorithm. They are: top-down approach and bottom-up approach, as shown in Fig. 1.9.

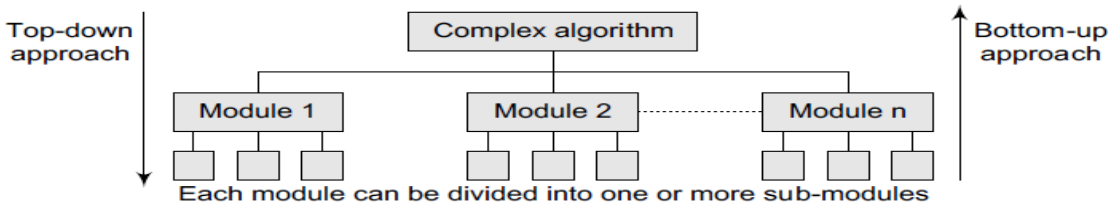


Figure 1.9 Different approaches of designing an algorithm

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 topmost 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.

Control structures used in algorithms:

An algorithm may employ one of the following control structures:

(a) sequence: means that each step of an algorithm is executed in a specified order.

(b) decision: when the execution of a process depends on the outcome of some condition.

(c) repetition: which involves executing one or more steps for a number of times,

1.6 TIME AND SPACE COMPLEXITY:

Analysing an algorithm means determining the amount of resources (such as time and memory) needed to execute it. Algorithms are generally designed to work with an arbitrary number of inputs, so 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. Similarly, 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.

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. Therefore, in this section, we will concentrate on the running time efficiency of algorithms.

1.6.1 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. However, 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.

1.6.2 Time–Space Trade-off:

The best algorithm to solve a particular problem at hand is no doubt the one 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.

1.6.3 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.

1.6.4 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. However, 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:

We have seen that in linear loops, the loop updation statement either adds or subtracts the loop-controlling variable. However, 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. Now, 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*. In order to analyse nested loops, we need to determine the number of iterations each loop completes. 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 to 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 in 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) = n2.**

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.**

ASYMPTOTIC NOTATIONS:

BIG O NOTATION:

The Big O notation, where O stands for 'order of', is concerned with what happens for very large values of n. For example, if a sorting algorithm performs n2 operations to sort just n elements, then that algorithm would be described as an O(n2) algorithm.

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.

We have seen that the 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 as f(n) ∈ O(g(n)) or as f(n) = O(g(n)). Here, n is the problem size and O(g(n)) = {h(n): ∃ positive constants c, n0 such that $0 \leq h(n) \leq cg(n)$, ∀ $n \geq n0$}. Hence, we can say that O(g(n)) comprises a set of all the functions h(n) that are less than or equal to cg(n) for all values of $n \geq n0$.

If $f(n) \leq cg(n)$, $c > 0$, ∀ $n \geq n0$, then f(n) = O(g(n)) and g(n) is an asymptotically tight upper bound for f(n).

Examples of functions in O(n3) include: n2.9, n3, n3 + n, 540n3 + 10.

Examples of functions not in O(n3) include: n3.2, n2, n2 + n, 540n + 10, 2n

To summarize,

- Best case O describes an upper bound for all combinations of input. It is possibly lower than the worst case. For example, when sorting an array the best case is when the array is already correctly sorted.
- Worst case O describes a lower bound for worst case input combinations. It is possibly greater than the best case. For example, when sorting an array the worst case is when the array is sorted in reverse order.

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(nk) where k > 1
- Exponential time algorithm: running time complexity given as O(2n)

Example 2.1 Show that 4n2 = O(n3).

Solution By definition, we have

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

Substituting 4n2 as h(n) and n3 as g(n), we get

$0 \leq 4n2 \leq cn3$

Dividing by n3

$0/n3 \leq 4n2/n3 \leq cn3/n3$

$0 \leq 4/n \leq c$

Now to determine the value of c, we see that 4/n is maximum when n=1. Therefore, c=4.

To determine the value of n0,

$0 \leq 4/n0 \leq 4$

$0 \leq 4/4 \leq n0$

$0 \leq 1 \leq n0$

This means n0=1. Therefore, $0 \leq 4n2 \leq 4n3$, $\forall\ n \geq n0=1$.

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) ∈ Ω(g(n)), where n is the problem size and Ω(g(n)) = {h(n): ∃ positive constants c > 0, n0 such that 0 ≤ cg(n) ≤ h(n), ∀ n ≥ n0}.

Hence, we can say that Ω(g(n)) comprises a set of all the functions h(n) that are greater than or equal to cg(n) for all values of n ≥ n0. If cg(n) ≤ f(n), c > O, ∀ n ≥ nO, then f(n) ∈ Ω(g(n)) and g(n) is an asymptotically tight lower bound for f(n).

Examples of functions in Ω(n2) include: n2, n2.9, n3 + n2, n3

Examples of functions not in Ω(n3) include: n, n2.9, n2

To summarize,

- Best case Ω describes a lower bound for all combinations of input. This implies that the function can never get any better than the specified value. For example, when sorting an array the best case is when the array is already correctly sorted.
- Worst case Ω describes a lower bound for worst case input combinations. It is possibly greater than best case. For example, when sorting an array the worst case is when the array is sorted in reverse order.
- If we simply write Ω, it means same as best case Ω.

Example: Show that 5n2 + 10n = Ω(n2).

Solution By the definition, we can write

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

$0 \le cn2 \le 5n2 + 10n$

Dividing by n2

$0/n2 \le cn2/n2 \le 5n2/n2 + 10n/n2$

$0 \le c \le 5 + 10/n$

Now, $\lim_{n->infinite} 5 + 10/n = 5$. Therefore, $0 \le c \le 5$.

Hence, c = 5

Now to determine the value of n0

$0 \le 5 \le 5 + 10/n0$

$-5 \le 5 - 5 \le 5 + 10/n0 - 5$

$-5 \le 0 \le 10/n0$

So n0 = 1 as $\lim_{n->infinite} \cdot 1/n = 0$

Hence, 5n2 + 10n = Ω(n2) for c=5 and ∀ n ≥ n_O=1.

THETA NOTATION (Θ):

Theta notation provides an asymptotically tight bound for f(n). Θ notation is simply written as, f(n) ∈ Θ(g(n)), where n is the problem size and Θ(g(n)) = {h(n): ∃ positive constants c1, c2, and n0 such that $0 \leq c1g(n) \leq h(n) \leq c2g(n)$, ∀ $n \geq n0$}.

Hence, we can say that Θ(g(n)) comprises a set of all the functions h(n) that are between c1g(n) and c2g(n) for all values of $n \geq n0$. If f(n) is between c1g(n) and c2g(n), ∀ $n \geq n0$, then f(n) ∈ Θ(g(n)) and g(n) is an asymptotically tight bound for f(n) and f(n) is amongst h(n) in the set.

To summarize,

- The best case in Θ notation is not used.
- Worst case Θ describes asymptotic bounds for worst case combination of input values.
- If we simply write Θ, it means same as worst case Θ.

Example 2.7 Show that $n2/2 - 2n = \Theta(n2)$.

Solution By the definition, we can write

$c1g(n) \leq h(n) \leq c2g(n)$

$c1n2 \leq n2/2 - 2n \leq c2n2$

Dividing by n2, we get

$c1n2/n2 \leq n2/2n2 - 2n/n2 \leq c2n2/n2$

$c1$

$\leq 1/2 - 2/n \leq c2$

This means $c2 = 1/2$ because $\lim_{n->infinite} 1/2 - 2/n = 1/2$ (Big O notation)

To determine c1 using Ω notation, we can write

$0 < c1 \leq 1/2 - 2/n$

We see that $0 < c1$ is minimum when $n = 5$. Therefore,

$0 < c1 \leq 1/2 - 2/5$

Hence, $c1 = 1/10$

Now let us determine the value of n0

$1/10 \leq 1/2 - 2/n0 \leq 1/2$

$2/n0 \leq 1/2 - 1/10 \leq 1/2$

$2/n0 \leq 2/5 \leq ½$

$n0 \geq 5$

You may verify this by substituting the values as shown below.

$c1n2 \leq n2/2 - 2n \leq c2n2$

$c1 = 1/10$, $c2 = 1/2$ and $n0 = 5$

$1/10(25) \leq 25/2 - 20/2 \leq 25/2$

$5/2 \leq 5/2 \leq 25/2$

Thus, in general, we can write, $1/10n2 \leq n2/2 - 2n \leq 1/2n2$ for $n \geq 5$.

OTHER USEFUL NOTATIONS:

There are other notations like little o notation and little ω notation which have been discussed below.

Little o Notation:

This notation provides a non-asymptotically tight upper bound for f(n). To express a function using this notation, we write $f(n) \in o(g(n))$ where $o(g(n)) = \{h(n) : \exists$ positive constants c, n0 such that for any $c > 0$, $n0 > 0$, and $0 \leq h(n) \leq cg(n), \forall\ n \geq n0\}$.

This is unlike the Big O notation where we say for some $c > 0$ (not any). For example, $5n3 = O(n3)$ is asymptotically tight upper bound but $5n2 = o(n3)$ is non-asymptotically tight bound for f(n).

Examples of functions in o(n3) include: n2.9, n3 / log n, 2n2

Examples of functions not in o(n3) include: 3n3, n3, n3 / 1000

Example: Show that $n3 / 1000 \neq o(n3)$.

Solution By definition, we have

$0 \leq h(n) < cg(n)$, for any constant $c > 0$

$0 \leq n3 / 1000 \leq cn3$

This is in contradiction with selecting any $c < 1/1000$.

An imprecise analogy between the asymptotic comparison of functions f(n) and g(n) and the relation between their values can be given as:

$f(n) = O(g(n)) \approx f(n) \leq g(n)$ $f(n) = o(g(n)) \approx f(n) < g(n)$ $f(n) = \Theta(g(n)) \approx f(n) = g(n)$

Little Omega Notation *(w)*:

This notation provides a non-asymptotically tight lower bound for f(n). It can be simply written as, $f(n) \in \omega(g(n))$, where $\omega(g(n)) = \{h(n) : \exists$ positive constants c, n0 such that for any $c > 0$, $n0 > 0$, and $0 \leq cg(n) < h(n), \forall\ n \geq n0\}$.

This is unlike the Ω notation where we say for some $c > 0$ (not any). For example, $5n3 = \Omega(n3)$ is asymptotically tight upper bound but $5n2 = \omega(n3)$ is non-asymptotically tight bound for f(n).

Example of functions in $\omega(g(n))$ include: $n3 = \omega(n2)$, $n3.001 = \omega(n3)$, $n2logn = \omega(n2)$

Example of a function not in ω(g(n)) is 5n2 ≠ ω(n2) (just as 5≠5)

Example: Show that 50n3/100 ≠ ω(n3).

Solution By definition, we have

$0 \leq cg(n) < h(n)$, for any constant $c > 0$

$0 \leq cn3 < 50n3/100$

Dividing by n3, we get

$0 \leq c < 50/100$

This is a contradictory value as for any value of c as it cannot be assured to be less than 50/100 or 1/2.

An imprecise analogy between the asymptotic comparison of functions f(n) and g(n) and the relation between their values can be given as:

$f(n) = \Omega(g(n)) \approx f(n) \geq g(n)$ $f(n) = \omega(g(n)) \approx f(n) > g(n)$

2. SEARCHING

INTRODUCTION TO SEARCHING:

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. However, 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: ***linear search*** and ***binary search*****.** The algorithm that should be used depends entirely on how the values are organized in the array.

2.1 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(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
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
```

Figure 2.1 Algorithm for linear search

Figure 2.1 shows the algorithm for linear search.

(a) In Steps 1 and 2 of the algorithm, we initialize the value of POS and I.

(b) 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).

(c) 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. However, 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.

Complexity of Linear Search Algorithm:

Linear search executes in O(n) time where n is the number of elements in the array. Obviously, 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. Likewise, 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. However, the performance of the linear search algorithm can be improved by using a sorted array.

Programming Example:

Write a program to search an element in an array using the linear search technique.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 20 // Added so the size of the array can be altered more easily
int main(int argc, char *argv[]) {
int arr[size], num, i, n, found = 0, pos = -1;
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", &arr[i]);
}
printf("\n Enter the number that has to be searched : ");
scanf("%d", &num);
for(i=0;i<n;i++)
{
        if(arr[i] == num)
        {
                found =1;
                pos=i;
                printf("\n %d is found in the array at position= %d", num,i+1);
/* +1 added in line 23 so that it would display the number in the first place in
the array as in position 1 instead of 0 */
                break;
        }
}
if (found == 0)
{
```

```
        printf("\n %d does not exist in the array", num);
        return 0;
    }
```

2.2 BINARY SEARCH:

Binary search is a searching algorithm that works efficiently with a sorted list. 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.

Take another analogy. How do we find words in a dictionary? We first open the dictionary somewhere in the middle. Then, we compare the first word on that page with the desired word whose meaning we are looking for. If the desired word comes before the word on the page, we look in the first half of the dictionary, else we look in the second half. Again, we open a page in the first half of the dictionary and compare the first word on that page with the desired word and repeat the same procedure until we finally get the word. The same mechanism is applied in the binary search.

Now, let us consider how this mechanism is applied to search for a value in a sorted array.

Consider an array A[] that is declared and initialized as

```
int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

and the value to be searched is VAL = 9. The algorithm will proceed in the following manner.

BEG = 0, END = 10, MID = (0 + 10)/2 = 5

Now, VAL = 9 and A[MID] = A[5] = 5

A[5] is less than VAL, therefore, we now search for the value in the second half of the array. So, we change the values of BEG and MID.

Now, BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 =16/2 = 8

VAL = 9 and A[MID] = A[8] = 8

A[8] is less than VAL, therefore, we now search for the value in the second half of the segment. So, again we change the values of BEG and MID.

Now, BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9

Now, VAL = 9 and A[MID] = 9.

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.

However, 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 VAL < A[MID], 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 VAL > A[MID], 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.

Figure 2.2 shows the algorithm for binary search.

(a) In Step 1, we initialize the value of variables, BEG, END, and POS.
(b) In Step 2, a while loop is executed until BEG is less than or equal to END.
(c) In Step 3, the value of MID is calculated.
(d) 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.
(e) 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.

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
        END = upper_bound, 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
```

Figure 2.2 Algorithm for binary search

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 a**s 2f(n) > n or f(n) = log2n.**

Programming Example:

Write a program to search an element in an array using binary search.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 10 // Added to make changing size of array easier
int smallest(int arr[], int k, int n); // Added to sort array
void selection_sort(int arr[], int n); // Added to sort array
int main(int argc, char *argv[])
{
```

```
    int arr[size], num, i, n, beg, end, mid, found=0;
    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", &arr[i]);
    }
    selection_sort(arr, n); // Added to sort the array
    printf("\n The sorted array is: \n");
    for(i=0;i<n;i++)
    printf(" %d\t", arr[i]);
    printf("\n\n Enter the number that has to be searched: ");
    scanf("%d", &num);
    beg = 0, end = n-1;
    while(beg<=end)
    {
        mid = (beg + end)/2;
        if (arr[mid] == num)
        {
            printf("\n %d is present in the array at position %d",
        num, mid+1);
            found =1;
            break;
        }
        else if (arr[mid]>num)
            end = mid-1;
        else
            beg = mid+1;
    }
    if (beg > end && found == 0)
        printf("\n %d does not exist in the array", num);
    return 0;
}
```

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

2.3 FIBONACCI SEARCH:

We are all well aware of the Fibonacci series in which the first two terms are 0 and 1 and then each successive term is the sum of previous two terms. 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, ...

The same series and concept can be used to search for a given value in a list of numbers. Sucha search algorithm which is based on Fibonacci numbers is called Fibonacci

search and was developed by Kiefer in 1953. The search follows a divide-and-conquer technique and narrows down possible locations with the help of Fibonacci numbers.

Fibonacci search is similar to binary search. It also works on a sorted list and has a run time complexity of **O(log n).** However, unlike the binary search algorithm, Fibonacci search does not divide the list into two equal halves rather it subtracts a Fibonacci number from the index to reduce the size of the list to be searched. So, the key advantage of Fibonacci search over binary search is that comparison dispersion is low.

3. SORTING

INTRODUCTION TO 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. 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;

3.1 INSERTION SORT:

Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time. We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards while playing bridge.

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.

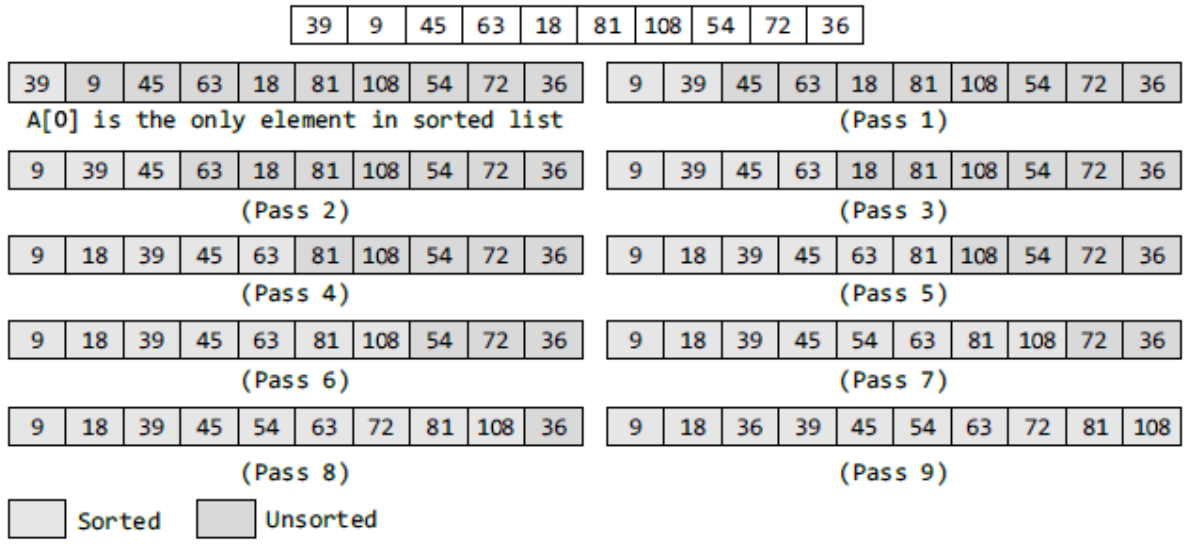
Technique:

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 (assuming LB = 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 (if LB = 0).
- 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)th location. The algorithm for insertion sort is given in Fig. 3.1.

```
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 3.1 Algorithm for insertion sort

In the algorithm,

(a) In Step 1 executes a for loop which will be repeated for each element in the array.

(b) In Step 2, we store the value of the Kth element in TEMP.

(c) In Step 3, we set the Jth index in the array.

(d) 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.

(e) Finally, In Step 5, the element is stored at the (J+1)th location.

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.

Similarly, 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. Furthermore, every iteration of the inner loop will have to shift the elements of the sorted set of the array before inserting the next element.

Therefore, in the worst case, insertion sort has a quadratic running time (i.e., O(n2)). Even in the average case, the insertion sort algorithm will have to make at least (K–1)/2 comparisons. Thus, the average case also has a quadratic running time.

Advantages of Insertion Sort:

The advantages of this sorting algorithm are as follows:

- It is easy to implement and efficient to use on small sets of data.
- It can be efficiently implemented on data sets that are already substantially sorted.
- It performs better than algorithms like selection sort and bubble sort. Insertion sort algorithm is simpler than shell sort, with only a small trade-off in efficiency. It is over twice as fast as the bubble sort and almost 40 per cent faster than the selection sort.
- It requires less memory space (only O(1) of additional memory space).
- It is said to be online, as it can sort a list as and when it receives new elements.

Programming Example:

Write a program to sort an array using insertion sort algorithm.

```
#include <stdio.h>
#include <conio.h>
#define size 5
void insertion_sort(int arr[], int n);
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]);
        }
        insertion_sort(arr, n);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
        getch();
```

```
}
void insertion_sort(int arr[], int n)
{
        int i, j, temp;
        for(i=1;i<n;i++)
        {
                temp = arr[i];
                j = i-1;
                while((temp < arr[j]) && (j>=0))
                {
                        arr[j+1] = arr[j];
                        j--;
                }
                arr[j+1] = temp;
        }
}
```

Output:

Enter the number of elements in the array : 5

Enter the elements of the array : 500 1 50 23 76

The sorted array is :

1 23 50 76 500

3.2 SELECTION SORT:

Selection sort is a sorting algorithm that has a quadratic running time complexity of O(n2), thereby making it inefficient to be used on large lists. Although selection sort performs worse than insertion sort algorithm, it is noted for its simplicity and also has performance advantages over more complicated algorithms in certain situations. Selection sort is generally used for sorting files with very large objects (records) and small keys.

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.

Example: Sort the array given below using selection sort.

	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

The algorithm for selection sort is shown in Fig. 3.2 In the algorithm, during the Kth pass, we need to find the position POS of the smallest elements from ARR[K], ARR[K+1], ..., ARR[N]. To find the smallest element, we use a variable SMALL to hold the smallest value in the sub-array ranging from ARR[K] to ARR[N]. Then, swap ARR[K] with ARR[POS]. This procedure is repeated until all the elements in the array are sorted.

```
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 3.2 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.

(a) In Pass 1, selecting the element with the smallest value calls for scanning all n elements; thus, n–1 comparisons are required in the first pass. Then, the smallest value is swapped with the element in the first position.

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

Therefore, $(n – 1) + (n – 2) + ... + 2 + 1 = n(n – 1) / 2 = O(n2)$ 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.

However, in case of large data sets, the efficiency of selection sort drops as compared to insertion sort.

Programming Example:

Write a program to sort an array using selection sort algorithm.

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

```
void selection_sort(int arr[], int n);
void main(int argc, char *argv[])
{
	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]);
}
int smallest(int arr[], int k, int n)
{
	int pos = k, small=arr[k], i;
	for(i=k+1;i<n;i++)
	{
		if(arr[i]< small)
		{
			small = arr[i];
			pos = i;
		}
	}
	return pos;
}
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;
        }
    }
```

3.3 EXCHANGE (BUBBLE SORT, QUICK SORT):

3.3.1 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 so that the element is placed before the bigger one. 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. Note that 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).

Note:

If the elements are to be sorted in descending order, then in first pass the smallest element is moved to the highest index of the array.

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: 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.

Observe that the entire list is sorted now.

```
BUBBLE_SORT(A, N)

Step 1: Repeat Step 2 For I = 0 to N-1
Step 2:     Repeat For J = 0 to N - I
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 3.3 Algorithm for bubble sort

Figure 3.3 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.

Complexity of Bubble Sort:

The complexity of any sorting algorithm depends upon the number of comparisons. In bubble sort, we have seen that 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:

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

$f(n) = n (n – 1)/2$

$f(n) = n2/2 + O(n) = O(n2)$

Therefore, the complexity of bubble sort algorithm is O(n2). It means the time required to execute bubble sort is proportional to n2, where n is the total number of elements in the array.

Programming Example:

Write a program to enter *n* numbers in an array. Redisplay the array with elements being sorted in ascending order.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, n, temp, j, arr[10];
        clrscr();
        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", &arr [i]);
        }
        for(i=0;i<n;i++)
```

```
        {
            for(j=0;j<n–i–1;j++)
            {
                if(arr[j] > arr[j+1])
                {
                    temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }
        }
        printf("\n The array sorted in ascending order is :\n");
        for(i=0;i<n;i++)
            printf("%d\t", arr[i]);
        getch();
        return 0;
    }
```

Output:

Enter the number of elements in the array : 10

Enter the elements : 8 9 6 7 5 4 2 3 1 10

The array sorted in ascending order is :

1 2 3 4 5 6 7 8 9 10

Bubble Sort Optimization:

Consider a case when the array is already sorted. In this situation no swapping is done but we still have to continue with all n–1 passes. We may even have an array that will be sorted in 2 or 3 passes but we still have to continue with rest of the passes. So once we have detected that the array is sorted, the algorithm must not be executed further. This is the optimization over the original bubble sort algorithm. In order to stop the execution of further passes after the array is sorted, we can have a variable flag which is set to TRUE before each pass and is made FALSE when a swapping is performed. The code for the optimized bubble sort can be given as:

```
void bubble_sort(int *arr, int n)
{
        int i, j, temp, flag = 0;
        for(i=0; i<n; i++)
        {
                for(j=0; j<n–i–1; j++)
                {
                        if(arr[j]>arr[j+1])
                        {
                                flag = 1;
                                temp = arr[j+1];
                                arr[j+1] = arr[j];
                                arr[j] = temp;
                        }
                }
                if(flag == 0) // array is sorted
                return;
        }
}
```

Complexity of Optimized Bubble Sort Algorithm:

In the best case, when the array is already sorted, the optimized bubble sort will take O(n) time.

In the worst case, when all the passes are performed, the algorithm will perform slower than the original algorithm. In average case also, the performance will see an improvement. Compare it with the complexity of original bubble sort algorithm which takes O(n2) in all the cases.

3.3.2 QUICK SORT:

Quick sort is a widely used sorting algorithm developed by C. A. R. Hoare that makes O(n log n) comparisons in the average case to sort an array of n elements. However, in the worst case, it has a quadratic running time given as O(n2). Basically, the quick sort algorithm is faster than other O(n log n) algorithms, because its efficient implementation can minimize

the probability of requiring quadratic time. Quick sort is also known as partition exchange sort.

Like merge 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.)

Like merge sort, the *base case* of the recursion occurs when the array has zero or one element because in that case the array is already sorted. After each iteration, one element (pivot) is always in its final position. Hence, with every iteration, there is one less element to be sorted in the array.

Thus, the main task is to find the pivot element, which will partition the array into two halves. To understand how we find the pivot element, follow the steps given below. (We take the first element in the array as pivot.)

Technique:

Quick sort works as follows:

1. Set the index of the first element in the array to loc and left variables. Also, set the index of the last element of the array to the right variable. That is, loc = 0, left = 0, and right = n–1 (where n in the number of elements in the array)
2. Start from the element pointed by right and scan the array from right to left, comparing each element on the way with the element pointed by the variable loc. That is, a[loc] should be less than a[right].
    (a) If that is the case, then simply continue comparing until right becomes equal to loc. Once right = loc, it means the pivot has been placed in its correct position.

(b) However, if at any point, we have a[loc] > a[right], then interchange the two values and jump to Step 3.

(c) Set loc = right

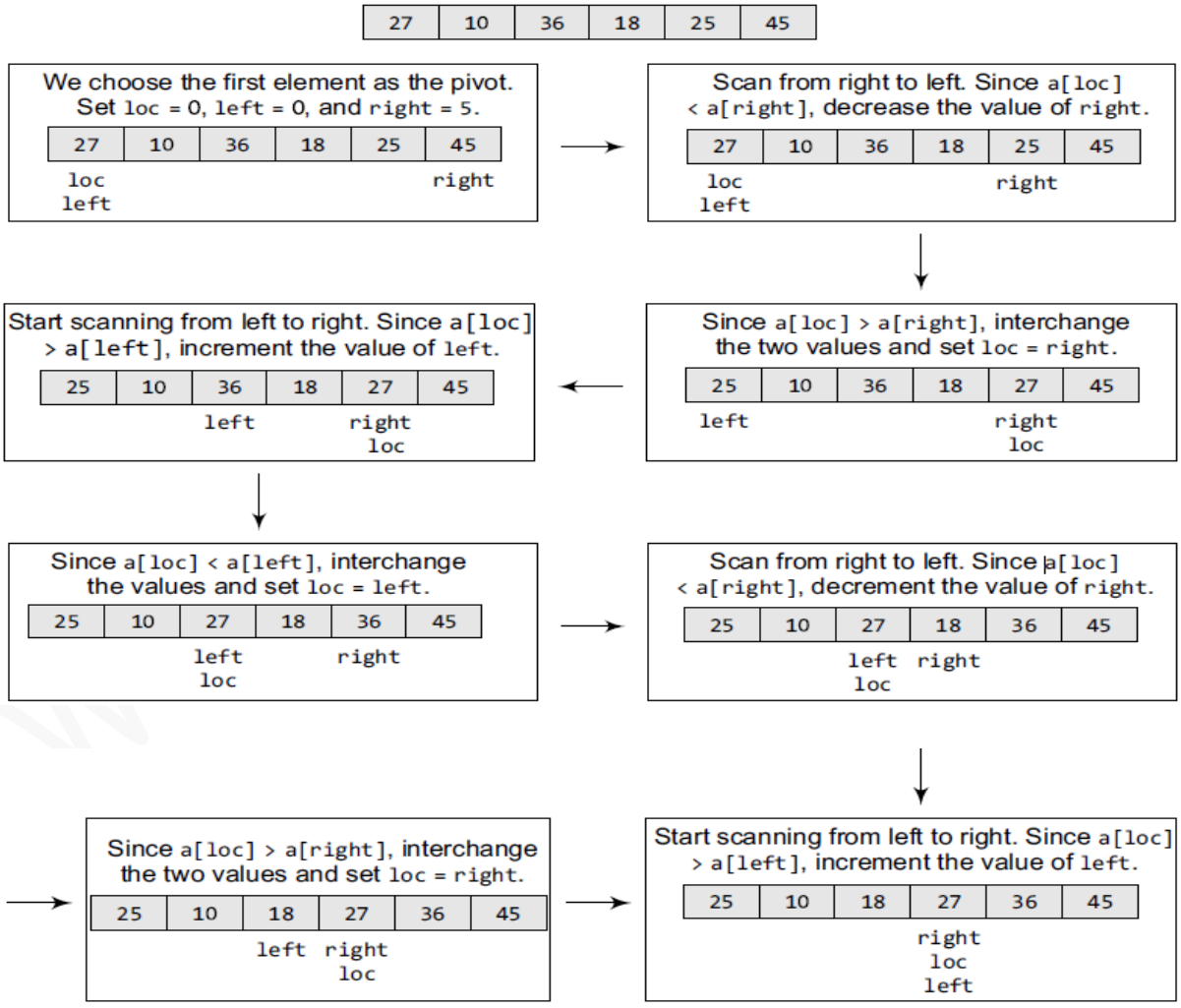
3. Start from the element pointed by left and scan the array from left to right, comparing each element on the way with the element pointed by loc. That is, a[loc] should be greater than a[left].

(a) If that is the case, then simply continue comparing until left becomes equal to loc. Once left = loc, it means the pivot has been placed in its correct position.

(b) However, if at any point, we have a[loc] < a[left], then interchange the two values and jump to Step 2.

(c) Set loc = left.

Example: Sort the elements given in the following array using quick sort algorithm



Now left = loc, so the procedure terminates, as the pivot element (the first element of the array, that is, 27) is placed in its correct position. All the elements smaller than 27 are placed before it and those greater than 27 are placed after it.

The left sub-array containing 25, 10, 18 and the right sub-array containing 36 and 45 are sorted in the same manner.

The quick sort algorithm (Fig. 3.4) makes use of a function Partition to divide the array into two sub-arrays.

```
PARTITION (ARR, BEG, END, LOC)

Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG = 0
Step 2: Repeat Steps 3 to 6 while FLAG = 0
Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC != RIGHT
                SET RIGHT = RIGHT - 1
        [END OF LOOP]
Step 4: IF LOC = RIGHT
                SET FLAG = 1
        ELSE IF ARR[LOC] > ARR[RIGHT]
                SWAP ARR[LOC] with  ARR[RIGHT]
                SET LOC = RIGHT
        [END OF IF]
Step 5: IF FLAG = 0
                Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
                SET LEFT = LEFT + 1
                [END OF LOOP]
Step 6:         IF LOC = LEFT
                        SET FLAG = 1
                ELSE IF ARR[LOC] < ARR[LEFT]
                        SWAP ARR[LOC] with  ARR[LEFT]
                        SET LOC = LEFT
                [END OF IF]
        [END OF IF]
Step 7: [END OF LOOP]
Step 8: END

QUICK_SORT (ARR, BEG, END)

Step 1: IF (BEG < END)
            CALL PARTITION (ARR, BEG, END, LOC)
            CALL QUICKSORT(ARR, BEG, LOC - 1)
            CALL QUICKSORT(ARR, LOC + 1, END)
        [END OF IF]
Step 2: END
```

Figure 3.4 Algorithm for quick sort

Complexity of Quick Sort:

In the average case, the running time of quick sort can be given as O(n log n). The partitioning of the array which simply loops over the elements of the array once uses O(n) time.

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. At the most, only log n nested calls can be made before we reach a sub-array of size 1. It means the depth of the call tree is O(log n). And because at each level, there can only be O(n), the resultant time is given as 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(n2). 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).

Pros and Cons of Quick Sort:

It is faster than other algorithms such as bubble sort, selection sort, and insertion sort. Quick sort can be used to sort arrays of small size, medium size, or large size. On the flip side, quick sort is complex and massively recursive.

Programming Example:

Write a program to implement quick sort algorithm.

```
#include <stdio.h>
#include <conio.h>
#define size 100
int partition(int a[], int beg, int end);
void quick_sort(int a[], int beg, int end);
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]);
        }
        quick_sort(arr, 0, n-1);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
        getch();
}
int partition(int a[], int beg, int end)
{
        int left, right, temp, loc, flag;
        loc = left = beg;
        right = end;
        flag = 0;
        while(flag != 1)
        {
                while((a[loc] <= a[right]) && (loc!=right))
                right--;
                if(loc==right)
                flag =1;
                else if(a[loc]>a[right])
                {
                        temp = a[loc];
                        a[loc] = a[right];
                        a[right] = temp;
                        loc = right;
                }
                if(flag!=1)
                {
                        while((a[loc] >= a[left]) && (loc!=left))
                        left++;
                        if(loc==left)
                                flag =1;
```

```
                else if(a[loc] <a[left])
                {
                        temp = a[loc];
                        a[loc] = a[left];
                        a[left] = temp;
                        loc = left;
                }
            }
        }
        return loc;
}
void quick_sort(int a[], int beg, int end)
{
        int loc;
        if(beg<end)
        {
            loc = partition(a, beg, end);
            quick_sort(a, beg, loc-1);
            quick_sort(a, loc+1, end);
        }
}
```

3.4 DISTRIBUTION (RADIX SORT):

Radix sort is a linear sorting algorithm for integers and uses the concept of 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 first 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 letter. After the second pass, names are sorted on the first two letters. 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) and the number of passes will depend on the length of the number having maximum number of digits.

```
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 3.5 Algorithm for radix sort

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								

After this pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. In the second pass, the numbers are sorted according to the digit at the tens place. The buckets are pictured upside down.

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									

In the third pass, the numbers are sorted according to the digit at the hundreds place. The buckets are pictured upside down.

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.

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. This feature compromises with the flexibility to sort input of any data type. For every different data type, the algorithm has to be rewritten. Even if the sorting order changes, the algorithm has to be rewritten. Thus, radix sort takes more time to write and writing a general purpose radix sort algorithm that can handle all kinds of data is not a trivial task.

Radix sort is a good choice for many programs that need a fast sort, but there are faster sorting algorithms available. This is the main reason why radix sort is not as widely used as other sorting algorithms.

Programming Example:

Write a program to implement radix sort algorithm.

```
##include <stdio.h>
#include <conio.h>
#define size 10
int largest(int arr[], int n);
void radix_sort(int arr[], int n);
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]);
        }
        radix_sort(arr, n);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
        getch();
}
int largest(int arr[], int n)
{
        int large=arr[0], i;
        for(i=1;i<n;i++)
        {
                if(arr[i]>large)
                large = arr[i];
        }
        return large;
}
void radix_sort(int arr[], int n)
{
        int bucket[size][size], bucket_count[size];
        int i, j, k, remainder, NOP=0, divisor=1, large, pass;
        large = largest(arr, n);
        while(large>0)
        {
                NOP++;
                large/=size;
        }
        for(pass=0;pass<NOP;pass++) // Initialize the buckets
        {
                for(i=0;i<size;i++)
                        bucket_count[i]=0;
                for(i=0;i<n;i++)
```

```
                {
                        // sort the numbers according to the digit at passth place
                        remainder = (arr[i]/divisor)%size;
                        bucket[remainder][bucket_count[remainder]] = arr[i];
                        bucket_count[remainder] += 1;
                }
                // collect the numbers after PASS pass
                i=0;
                for(k=0;k<size;k++)
                {
                        for(j=0;j<bucket_count[k];j++)
                        {
                                arr[i] = bucket[k][j];
                                i++;
                        }
                }
                divisor *= size;
        }
}
```

3.5 MERGING (MERGE SORT) ALGORITHMS:

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):

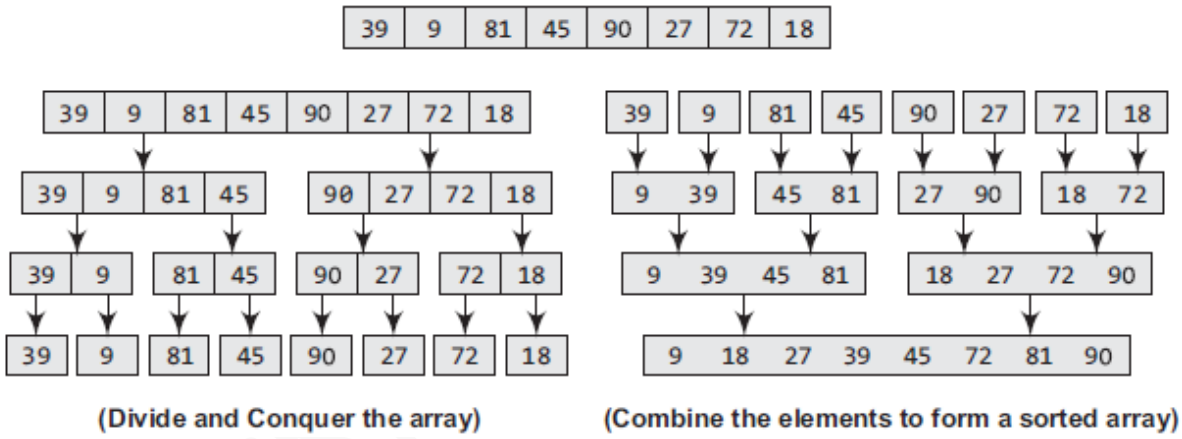
- A smaller list takes fewer steps and thus less time to sort than a large list.
- As number of steps is relatively less, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists.

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.

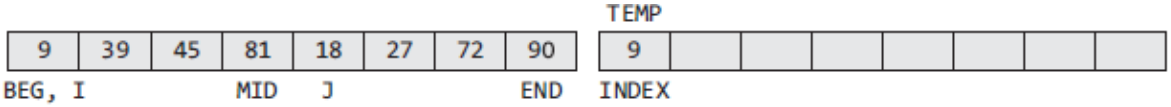
Example: Sort the array given below using merge sort.

Solution

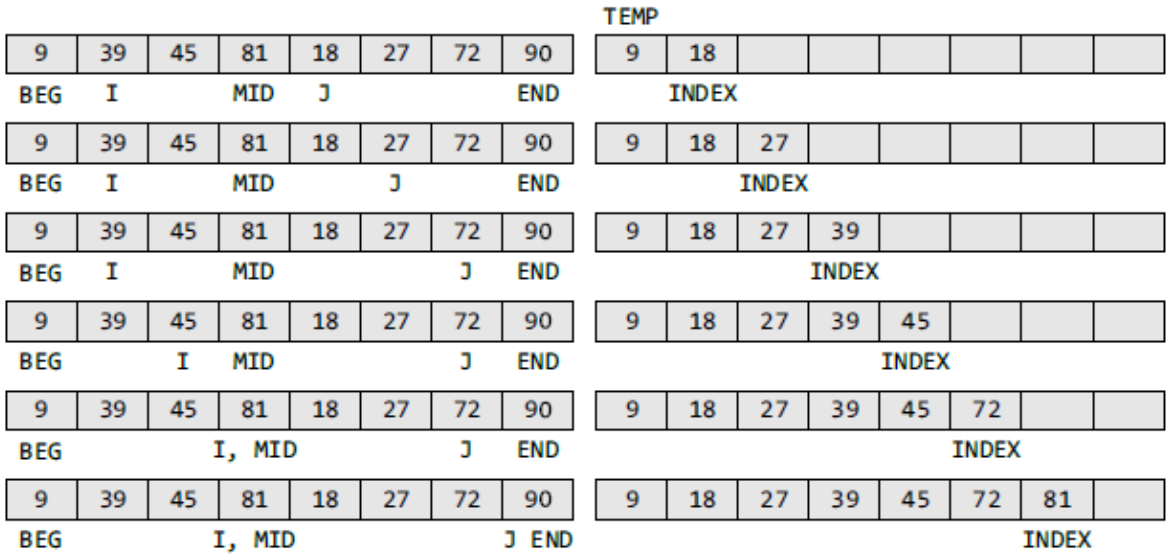


The merge sort algorithm (Fig. 3.6) uses a function merge which combines the sub-arrays to form a sorted array. While the merge sort algorithm recursively divides the list into smaller lists, the merge algorithm conquers the list to sort the elements in individual lists. Finally, the smaller lists are merged to form one list.

To understand the merge algorithm, consider the figure below which shows how we merge two lists to form one list. For ease of understanding, we have taken two sub-lists each containing four elements. The same concept can be utilized to merge four sub-lists containing two elements, or eight sub-lists having one element each.



Compare ARR[I] and ARR[J], the smaller of the two is placed in TEMP at the location specified by INDEX and subsequently the value I or J is incremented.



When I is greater than MID, copy the remaining elements of the right sub-array in TEMP.


```
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(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 3.6 Algorithm for 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.

Programming Example:

Write a program to implement merge sort.

```
#include <stdio.h>
#include <conio.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]);
```

```
	getch();
}
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);
        }
}
```