Unit – I

Syllabus:

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

INTRODUCTION:

- A *data structure* is a particular way of storing and organizing data in a computer so that it can be used efficiently.
- Some common examples of data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables
- Today computer programmers do not write programs just to solve a problem but to write an efficient program.
- 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.
  2. Quantify the resource constraints for each operation.
  3. Select the data structure that best meets these requirements.

- The term *data* means a value or set of values. It specifies either the value of a variable or a constant (e.g., marks of students, name of an employee, address of a customer, value of *pi*, etc.).
- A *record* is a collection of data items. For example, the name, address, course, and marks obtained are individual data items. But all these data items can be grouped together to form a record.
- A *file* is a collection of related records. For example, if there are 60 students in a class, then there are 60 records of the students. All these related records are stored in a file.

CLASSIFICATION OF DATA STRUCTURES:

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

Primitive and Non-primitive Data Structures:

- Primitive data structures are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character, and boolean. The terms ‘data type, basic data type’, and ‘primitive data type’ are often used interchangeably.

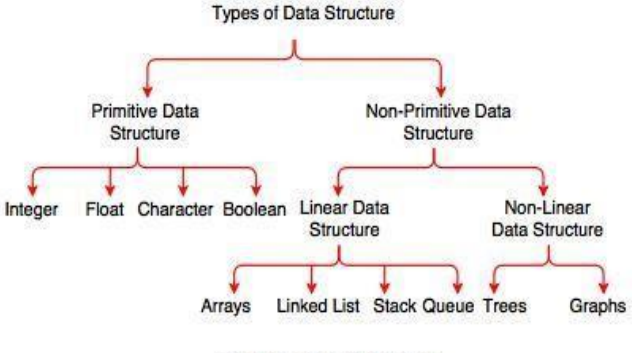


Fig. Types of Data Structure

- 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 Structures:

- If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure.
    - Examples include arrays, linked lists, stacks, and queues.
    - Linear data structures can be represented in memory in two different ways. 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.
    - Each element is attached with its next and previous adjacent.
    - A linear data structure only has one level and performs linear searching in the data structure. We can therefore traverse all elements in a single run.
    - Because computer memory is linearly arranged, linear data structures are simple to implement.

- 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.
    - In a nonlinear structure, only one level of data is not used.
    - So, it is impossible to traverse the entire structure in one run.
    - Nonlinear data structures can be straightforward to design and implement compared to linear data structures.
    - They make use of computer memory effectively when compared to a linear structure.
    - Examples include trees and graphs.

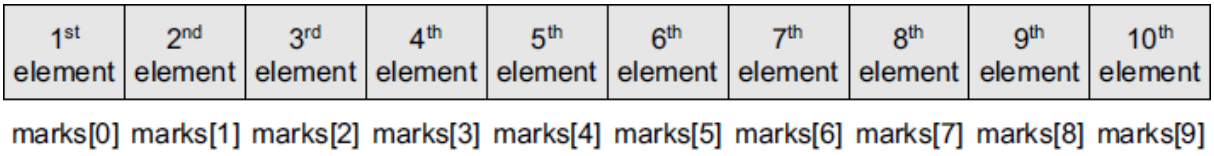
Basis of	Linear Data Structure	Nonlinear Data Structure
Data Arrangements:	Linear data structures are data elements that are organized linearly. Each element is attached to the next and previous adjacent.	Data elements in a nonlinear data system are attached hierarchically.
Levels:	A linear data structure only has one level.	Multilevels are possible in

		nonlinear data structures.
Implementation:	It is much easier than nonlinear data structures.	It is more complicated than a linear data structure but can be implemented.
Transferring Data Elements:	Data elements in linear data structures can only be traversed once	Nonlinear data structures can be traversed over multiple runs, but data elements cannot be traversed in one run.
Memory Usage:	Memory is not used efficiently in a linear data structure.	Memory is used efficiently in nonlinear data structures.
Examples:	Its examples are queue, linked list, array, stack, etc.	Its examples include graphs and trees.
Usage:	The application of linear data structures is mainly used in software development.	Artificial Intelligence and Image Processing are two examples of applications for nonlinear 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: datatype name[size];

  Ex: int marks[10];



limitations:

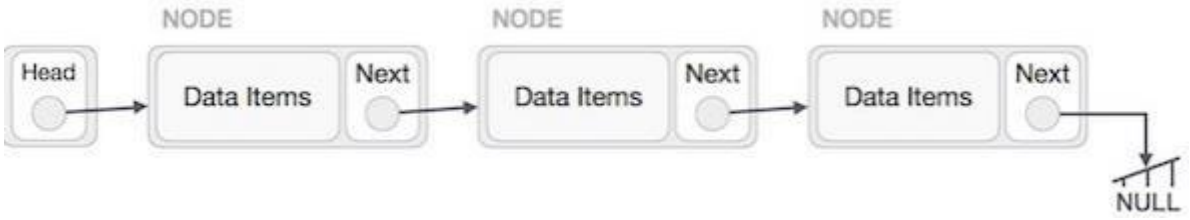
- Arrays are of fixed size.
- Data elements are stored in contiguous memory locations which may not be always available.
- Insertion and deletion of elements can be problematic because of shifting of elements from their positions.

Linked Lists:

- linked list is a dynamic data structure in which elements (called *nodes*) form a sequential list.
- 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.
- Every node contains the following

  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



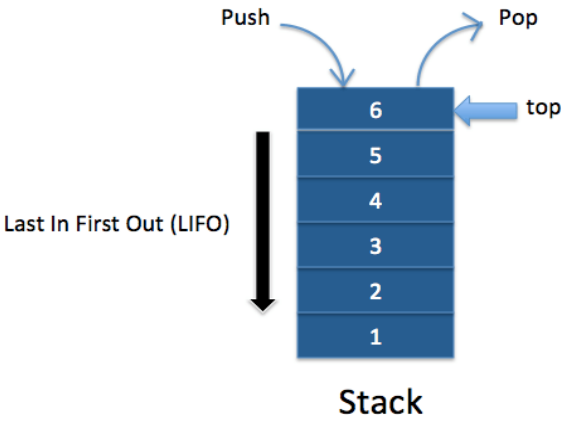
- The first node in the list is pointed by Head/Start/First. The last node in the list contains a NULL pointer to indicate that it is the end or *tail* of the list.

  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.
- Stacks can be implemented using arrays or linked lists.
- Every stack has a variable top associated with it. Top is used to store the address of the topmost element of the stack.

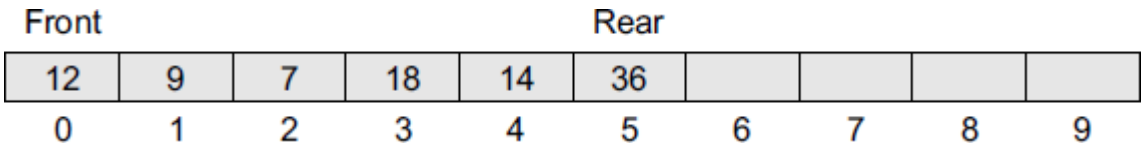


- It is this position from where the element will be added or deleted. There is another variable MAX, which is used to store the maximum number of elements that the stack can store.
- If top = NULL, then it indicates that the stack is empty and if top = MAX–1, then the stack is full.
- A stack supports 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).

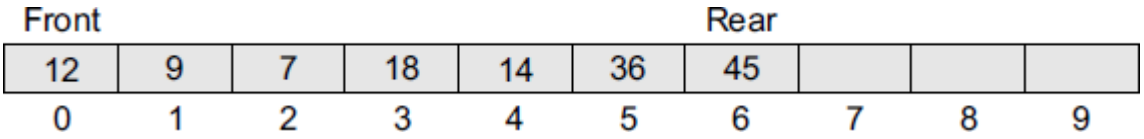
Queues:

- A Queue is a linear data structure in which insertion can be done at rear end and deletion of elements can be dome at front end.
- 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.

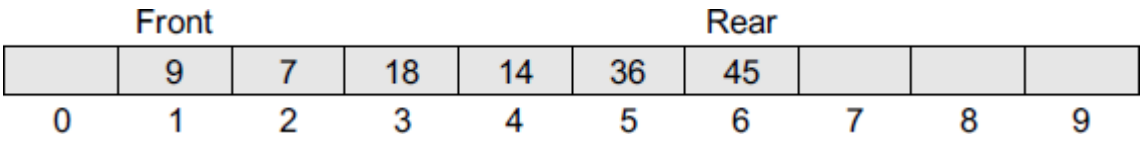
- Like stacks, queues can be implemented by using either arrays or linked lists.



Insert element into the Queue:



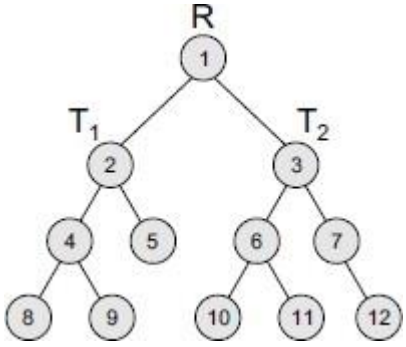
Delete element from Queue:



- A queue is full when rear = MAX – 1, 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.

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.



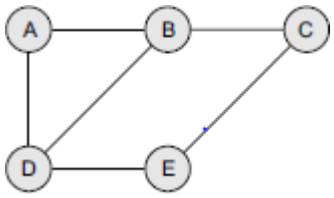
- Here R is the root node and T1 and T2 are the left and right subtrees of R. If T1 is non-empty, then T1 is said to be the left successor of R. Likewise, if T2 is non-empty, then it is called the right successor of R.

  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 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 do not have any root node. Rather, every node in the graph can be connected with every another node in the graph.

  Advantage: Best models real-world situations

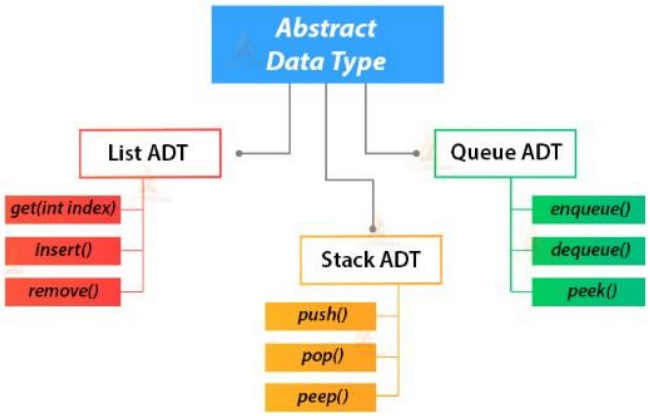
  Disadvantage: Some algorithms are slow and very complex

OPERATIONS ON DATA STRUCTURES:

- This section discusses the different operations that can be performed on the various data structures previously mentioned.
- ***Traversing*** It means to access each data item exactly once so that it can be processed. For example, to print the names of all the students in a class.
- ***Searching*** It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items. For example, to find the names of all the students who secured 100 marks in mathematics.
- ***Inserting*** It is used to add new data items to the given list of data items. For example, to add the details of a new student who has recently joined the course.
- ***Deleting*** It means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course.
- ***Sorting*** Data items can be arranged in some order like ascending order or descending order depending on the type of application. For example, arranging the names of students in a class in an alphabetical order, 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.

ABSTRACT DATA TYPE:

- An *abstract data type* (ADT) is a data structure, focusing on what it does and ignoring how it does its job. (or) Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations.
- Ex: stacks ADT and queues ADT. the user is concerned only with the type of data and the operations that can be performed on it.
- We can implement both these ADTs using an array or a linked list.



Advantage of using ADTs

- Modification of a program is simple, 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.

PRELIMINARIES OF ALGORITHM:

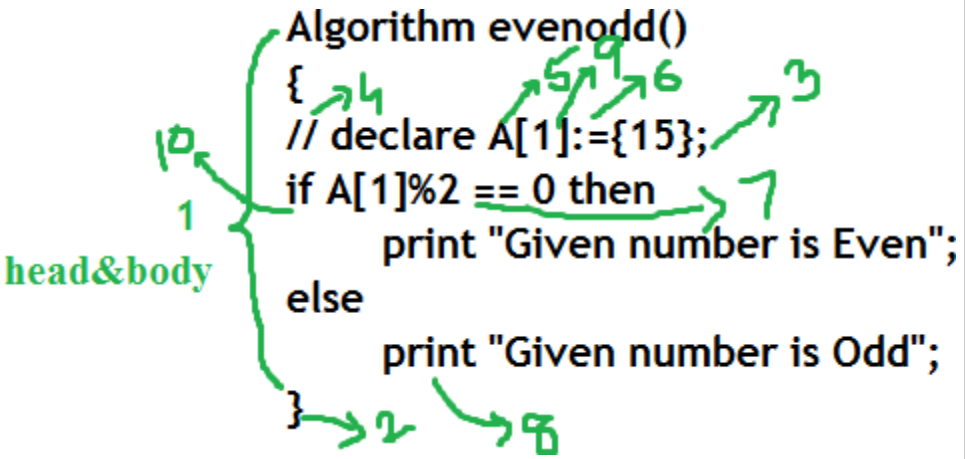
- Algorithm is step by step logical procedure for solving a problem.
- In Algorithm each step is called *Instruction.*
- An Algorithm is any well-defined computational procedure that take some values as inputs and produce some values as output.
- An Algorithm is a sequence of computational steps that transform input into output.
- An Algorithm has 5 basic properties:
  1. *Input:*An Algorithm has take '0' or more number of inputs that can be supplied as externally.
  2. *Output:* An Algorithm must produce at least one output.
  3. *Definiteness:* Each instruction in the algorithm must be clear.
  4. *Finiteness:* An algorithm must terminate after a finite number of steps.
  5. *Effectiveness:* Each operation should be effective. i.e the operations must be terminate after finite amount of time.

Structure of an Algorithm:

1. Algorithm is a procedure consisting of heading and body. In body part we are writing statements and in the head part we are writing the following.

   Syntax: Algorithm name_of_Algo (param1,param2, …);

2. The beginning and ending of block should be indicated by '{' and '}' or 'start' and 'end' respectively.
3. Every statement in the algorithm should be end with semicolon (;).
4. Single line comments are written using '//' as beginning of comments.
5. The identifier should begin with character and it may be combination of alpha numeric.



6. Assignment operator (:=) we can use as follows

   Variable := expression (or) value;
7. There are other type of operators such as Boolean operators (TRUE/FALSE), logical operators (AND,OR,NOT) and relational operators (<,>,<=,>=,…..)
8. The input and output we can write it as read and print respectively.
9. The Array index are stored with in [ ] brackets. The index of array starts from '0' to 'N-1'.

   Syntax: datatype Aray_name[size];
10. The conditional statements such as if-then (or) if-then-else are written as follows.

    if(condition) then statements;

    if(condition) then

    statements;

    else

    statements;

TIME AND SPACE COMPLEXITY:

- The efficiency of an algorithm can be computed by measuring the performance of an algorithm. We can measure the performance of an algorithm in Two(2) ways.
  1. Time Complexity
  2. Space Complexity

1. Time Complexity:

- The time complexity of an algorithm is the amount of computing time required by an algorithm to run its completion.
- There are 2 types of computing time 1. Compile time 2. Run time

- The time complexity generally computed at run time (or) execution time.
- The time complexity can be calculated in terms of frequency count.
- Frequency count is a count denoting the number of times the statement should be executed.
- The time complexity can be calculated as

  Comments – 0
  Assignment / return statement – 1
  Conditional (or) Selection Constructs – 1

Example 1: Sum of the elements in an Array

Statements	Step count/ Execution	Frequency	Total Steps
Algorithm Addition (A,n)	0	-	0
{	0	-	0
//A is an array of size 'n'	0	-	0
Sum :=0;	1	1	1
for i:=1 to n do	1	n+1	n+1
Sum:=Sum+A[i];	1	n	n
return Sum;	1	1	1
}	0	-	0
Total		**2n+3**	

Example 2: Subtraction of two matrices

Statements	Step count/ Execution	Frequency	Total Steps
Algorithm Subtract (A,B,C,m,n)	0	-	0
{	0	-	0
for i:=1 to m do	1	m+1	m+1
for j:=1 to n do	1	m(n+1)	mn+m
C[i,j] := A[i,j] – B[i,j];	1	mn	mn
}	0	-	0
Total		**2mn+2m+1**	

2. Space Complexity:

- Space Complexity can be defined as amount of memory (or) space required by an Algorithm to run.
- To compute the space complexity we use 2 factors i. Constant ii. Instance characteristics.
- The space requirement S(p) can be given as $S(p) = C+S_p$

  Where C- Constant, it denotes the space taken for input and output.

  S_p – Amount of space taken by an instruction, variable and identifiers.

Example 1: Sum of three numbers

```
Algorithm Add(a,b,c)
{
//a,b,c are float type variables
return a+b+c;
}
```

- The space required for this algorithm is: Assume a,b,c are occupies 1 word size each, total size comes to be **3.**

Example 2: Sum of Array values

```
Algorithm Addition (A,n)
{
//A is an array of size 'n'
Sum :=0;
for i:=1 to n do
      Sum:=Sum+A[i];
return Sum;
}
```

- The space required for this algorithm is:

  One word space for each variable then i,sum,n → 3

  For Array A[ ] we require the size → n

  Total space complexity for this algorithm is **S(p) ≥ (n+3)**

What to Analyze in an algorithm:

An Algorithm can require different times to solve different problems of same size

1. Worst case: Maximum amount of time that an algorithm require to solve a problem of size 'n'.

   Normally we can take upper bound as complexity. We try to find worst case behavior.

2. Best case: Minimum amount of time that an algorithm require to solve a problem of size 'n'.

   Normally it is not much useful.

3. Average case: the average amount of time that an algorithm require to solve a problem of size 'n'.

   Some times it is difficult to find. Because we have to check all possible data organizations

Complexity Analysis

Complexity Analysis is the systematic study of the cost of computation, measured either in time units or in operations performed, or in the amount of storage space required.

The goal is to have a meaningful measure that permits comparison of algorithms independent of operating platform. There are two things to consider:

- **Time Complexity**: Determine the approximate number of operations required to solve a problem of size n.
- **Space Complexity:** Determine the approximate memory required to solve a problem of size n.

Complexity analysis involves two distinct phases:

- **Algorithm Analysis**: Analysis of the algorithm or data structure to produce a function T (n) that describes the algorithm in terms of the operations performed in order to measure the complexity of the algorithm.
- **Order of Magnitude Analysis**: Analysis of the function T (n) to determine the general complexity category to which it belongs.

There is no generally accepted set of rules for algorithm analysis. However, an exact count of operations is commonly used.

Analysis Rules:

1. We assume an arbitrary time unit.
2. Execution of one of the following operations takes time 1:
   - Assignment Operation
   - Single Input/Output Operation
   - Single Boolean Operations
   - Single Arithmetic Operations
   - Function Return
3. Running time of a selection statement (if, switch) is the time for the condition evaluation + the maximum of the running times for the individual clauses in the selection.
4. Loops: Running time for a loop is equal to the running time for the statements inside the loop * number of iterations.
   The total running time of a statement inside a group of nested loops is the running time of the statements multiplied by the product of the sizes of all the loops.
   For nested loops, analyze inside out.
   - Always assume that the loop executes the maximum number of iterations possible.
5. Running time of a function call is 1 for setup + the time for any parameter calculations + the time required for the execution of the function body.

Examples:

1.
```
int count(){
  int k=0;
  cout<< "Enter an integer";
  cin>>n;
  for (i=0;i<n;i++)
                k=k+1;
  return 0;}
```

Time Units to Compute

---

1 for the assignment statement: int k=0
1 for the output statement.
1 for the input statement.
In the for loop:
1 assignment, *n+1* tests, and *n* increments.
n loops of 2 units for an assignment, and an addition.
1 for the return statement.

$T(n) = 1+1+1+(1+n+1+n)+2n+1 = 4n+6 = O(n)$

2.
```
int total(int n)
  {
  int sum=0;
  for (int i=1;i<=n;i++)
      sum=sum+1;
   return sum;
   }
```

Time Units to Compute

---

1 for the assignment statement: int sum=0
In the for loop:
1 assignment, *n+1* tests, and *n* increments.
n loops of 2 units for an assignment, and an addition.
1 for the return statement.

$T(n) = 1+ (1+n+1+n)+2n+1 = 4n+4 = O(n)$

3. void func()

```
  {
  int x=0;
  int i=0;
  int j=1;
  cout<< "Enter an Integer value";
  cin>>n;
  while (i<n){
     x++;
      i++;
   }
  while (j<n)
   {
     j++;
   }
 }
```

Time Units to Compute

---

1 for the first assignment statement: x=0;
1 for the second assignment statement: i=0;
1 for the third assignment statement: j=1;
1 for the output statement.
1 for the input statement.
In the first while loop:
n+1 tests
n loops of 2 units for the two increment (addition) operations
In the second while loop:
n tests
n-1 increments

T (n)= *1+1+1+1+1+n+1+2n+n+n-1 = 5n+5 = O(n)*

```
4. int sum (int n)
  {
   int partial_sum = 0;
   for (int i = 1; i <= n; i++)
         partial_sum = partial_sum +(i * i * i);
   return partial_sum;
  }
```

Time Units to Compute

---

1 for the assignment.
1 assignment, *n+1* tests, and *n* increments.
n loops of 4 units for an assignment, an addition, and two multiplications.
1 for the return statement.

T (n)= *1+(1+n+1+n)+4n+1 = 6n+4 = O(n)*

Formal Approach to Analysis

In the above examples we have seen that analysis is a bit complex. However, it can be simplified by using some formal approach in which case we can ignore initializations, loop control, and book keeping.

for Loops: Formally

- In general, a for loop translates to a summation. The index and bounds of the summation are the same as the index and bounds of the for loop.

```
for (int i = 1; i <= N; i++) {
      sum = sum+i;
}
```

$$\sum_{i=1}^{N} 1 = N$$

- Suppose we count the number of additions that are done. There is 1 addition per iteration of the loop, hence N additions in total.

Nested Loops: Formally

- Nested for loops translate into multiple summations, one for each for loop.

```
for (int i = 1; i <= N; i++) {
      for (int j = 1; j <= M; j++) {
            sum = sum+i+j;
      }
}
```

$$\sum_{i=1}^{N}\sum_{j=1}^{M} 2 = \sum_{i=1}^{N} 2M = 2MN$$

- Again, count the number of additions. The outer summation is for the outer for loop.

Consecutive Statements: Formally

- Add the running times of the separate blocks of your code

```
for (int i = 1; i <= N; i++) {
      sum = sum+i;
}
for (int i = 1; i <= N; i++) {
      for (int j = 1; j <= N; j++) {
            sum = sum+i+j;
      }
}
```

$$\left[\sum_{i=1}^{N} 1\right] + \left[\sum_{i=1}^{N}\sum_{j=1}^{N} 2\right] = N + 2N^2$$

Conditionals: Formally

- If (test) s1 else s2: Compute the maximum of the running time for s1 and s2.

```
if (test == 1) {
    for (int i = 1; i <= N; i++) {
        sum = sum+i;
}}
else for (int i = 1; i <= N; i++) {
          for (int j = 1; j <= N; j++) {
              sum = sum+i+j;
}}
```

$$\max\left(\sum_{i=1}^{N} 1, \sum_{i=1}^{N}\sum_{j=1}^{N} 2\right) =$$
$$\max\left(N, 2N^2\right) = 2N^2$$

Example:

Suppose we have hardware capable of executing 10^6 instructions per second. How long would it take to execute an algorithm whose complexity function was:

$T(n) = 2n^2$ on an input size of $n=10^8$?

The total number of operations to be performed would be $T(10^8)$:

$T(10^8) = 2*(10^8)^2 = 2*10^{16}$

The required number of seconds required would be given by

$T(10^8)/10^6$ so:

Running time $=2*10^{16}/10^6 = 2*10^{10}$
The number of seconds per day is 86,400 so this is about 231,480 days (634 years).

Exercises
Determine the run time equation and complexity of each of the following code segments.

1.
```
for (i=0;i<n;i++)
    for (j=0;j<n; j++)
        sum=sum+i+j;
```

2.
```
for(int i=1; i<=n; i++)
        for (int j=1; j<=i; j++)
    sum++;
```
What is the value of the sum if n=20?

3.
```
int  k=0;
 for (int i=0; i<n; i++)
   for (int j=i; j<n; j++)
     k++;
```
What is the value of k when n is equal to 20?

4.
```
int k=0;
 for (int i=1; i<n; i*=2)
   for(int j=1; j<n; j++)
       k++;
```
What is the value of k when n is equal to 20?

5.
```
int x=0;
 for(int i=1;i<n;i=i+5)
     x++;
```
What is the value of x when n=25?

6.
```
int x=0;
 for(int k=n;k>=n/3;k=k-5)
     x++;
```
What is the value of x when n=25?

7.
```
int x=0;
 for (int i=1; i<n;i=i+5)
    for (int k=n;k>=n/3;k=k-5)
        x++;
```
What is the value of x when n=25?

8.
```
int x=0;
 for(int i=1;i<n;i=i+5)
     for(int j=0;j<i;j++)
         for(int k=n;k>=n/2;k=k-3)
             x++;
```

What is the correct big-Oh Notation for the above code segment?

Measures of Times

In order to determine the running time of an algorithm it is possible to define three functions $T_{best}(n)$, $T_{avg}(n)$ and $T_{worst}(n)$ as the best, the average and the worst case running time of the algorithm respectively.

Average Case (T_{avg}): The amount of time the algorithm takes on an "average" set of inputs.
Worst Case (T_{worst}): The amount of time the algorithm takes on the worst possible set of inputs.
Best Case (T_{best}): The amount of time the algorithm takes on the smallest possible set of inputs.

We are interested in the worst-case time, since it provides a bound for all input – this is called the "Big-Oh" estimate.

Asymptotic Analysis

Asymptotic analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

There are five notations used to describe a running time function. ***These are:***

- Big-Oh Notation (O)
- Big-Omega Notation (Ω)
- Theta Notation (Θ)
- Little-o Notation (o)
- Little-Omega Notation (ω)

The Big-Oh Notation

Big-Oh notation is a way of comparing algorithms and is used for computing the complexity of algorithms; i.e., the amount of time that it takes for computer program to run . It's only concerned with what happens for very a large value of n. Therefore only the largest term in the expression (function) is needed. For example, if the number of operations in an algorithm is $n^2 – n$, n is insignificant compared to n^2 for large values of n. Hence the n term is ignored. Of course, for small values of n, it may be important. However, Big-Oh is mainly concerned with large values of n.

Formal Definition: f (n)= O (g (n)) if there exist c, k ∈ $\mathscr{R}^+$ such that for all n≥ k, f (n) ≤ c.g (n).

Examples: The following points are facts that you can use for Big-Oh problems:

- $1<=n$ for all $n>=1$
- $n<=n^2$ for all $n>=1$
- $2^n <=n!$ for all $n>=4$
- $\log_2 n<=n$ for all $n>=2$
- $n<=n\log_2 n$ for all $n>=2$

1. f(n)=10n+5 and g(n)=n. Show that f(n) is O(g(n)).

To show that f(n) is O(g(n)) we must show that constants c and k such that

f(n) <=c.g(n) for all n>=k

Or 10n+5<=c.n for all n>=k

Try c=15. Then we need to show that 10n+5<=15n

Solving for n we get: 5<5n or 1<=n.

So f(n) =10n+5 <=15.g(n) for all n>=1.

(c=15,k=1).

2. $f(n) = 3n^2 +4n+1$. Show that $f(n)=O(n^2)$.

$4n <=4n^2$ for all n>=1 and $1<=n^2$ for all n>=1

$3n^2 +4n+1<=3n^2+4n^2+n^2$ for all n>=1

$<=8n^2$ for all n>=1

So we have shown that $f(n)<=8n^2$ for all $n>=1$

Therefore, f (n) is $O(n^2)$ (c=8,k=1)

Typical Orders

Here is a table of some typical cases. This uses logarithms to base 2, but these are simply proportional to logarithms in other base.

N	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(n^3)$
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4,096
1024	1	10	1,024	10,240	1,048,576	1,073,741,824

Demonstrating that a function f(n) is big-O of a function g(n) requires that we find specific constants c and k for which the inequality holds (and show that the inequality does in fact hold).

Big-O expresses an *upper bound* on the growth rate of a function, for sufficiently large values of n.

An *upper bound* is the best algorithmic solution that has been found for a problem.
" What is the best that we know we can do?"

Exercise:

$f(n) = (3/2)n^2+(5/2)n-3$

Show that $f(n)= O(n^2)$

In simple words, f (n) =O(g(n)) means that the growth rate of f(n) is less than or equal to g(n).

Big-O Theorems

For all the following theorems, assume that f(n) is a function of n and that k is an arbitrary constant.

Theorem 1: k is O(1)
Theorem 2: A polynomial is O(the term containing the highest power of n).

Polynomial's growth rate is determined by the leading term

- If *f(n)* is a polynomial of degree *d*, then *f(n)* is $O(n^d)$

In general, f(n) is big-O of the dominant term of f(n).

Theorem 3: k*f(n) is O(f(n))

Constant factors may be ignored

E.g. $f(n) = 7n^4+3n^2+5n+1000$ is $O(n^4)$

Theorem 4(Transitivity): If f(n) is O(g(n))and g(n) is O(h(n)), then f(n) is O(h(n)).

Theorem 5: For any base b, $\log_b(n)$ is O(logn).

All logarithms grow at the same rate

$\log_b n$ is $O(\log_d n)$ $\forall b, d > 1$

Theorem 6: Each of the following functions is big-O of its successors:

k
$\log_b n$
n
m
$n\log_b n$
n^2
n to higher powers
2^n
3^n
larger constants to the nth power
n!
n^n

$f(n)= 3n\log_b n + 4 \log_b n+2$ is $O(n\log_b n)$ and $)(n^2)$ and $O(2^n)$

Big-Omega Notation

Just as O-notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Formal Definition: A function f(n) is $\Omega(g (n))$ if there exist constants c and $k \in \mathbb{R}+$ such that

$f(n) >= c. g(n)$ for all $n>=k$.

$f(n)= \Omega(g (n))$ means that f(n) is greater than or equal to some constant multiple of g(n) for all values of n greater than or equal to some k.

Example*:* If $f(n) =n^{2,}$ then $f(n)= \Omega(n)$

In simple terms, $f(n)= \Omega(g (n))$ means that the growth rate of f(n) is greater that or equal to g(n).

Theta Notation

A function f (n) belongs to the set of Θ (g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched between c1.g(n) and c2.g(n), for sufficiently large values of n.

Formal Definition: A function f (n) is Θ (g(n)) if it is both $O(g(n))$ and $\Omega (g(n))$. In other words, there exist constants c1, c2, and k >0 such that c1.g (n)<=f(n)<=c2. g(n) for all n >= k

If f(n)= Θ (g(n)), then g(n) is an asymptotically tight bound for f(n).

In simple terms, f(n)= Θ (g(n)) means that f(n) and g(n) have the same rate of growth.

Example:

1. If f(n)=2n+1, then f(n) = Θ (n)

2. f(n) =$2n^2$ then

f(n)=O(n^4)

f(n)=O(n^3)

f(n)=O(n^2)

All these are technically correct, but the last expression is the best and tight one. Since $2n^2$ and n^2 have the same growth rate, it can be written as f(n)= Θ(n^2).

Little-o Notation

Big-Oh notation may or may not be asymptotically tight, for example:

$2n^2$ = O(n^2)

=O(n^3)

f(n)=o(g(n)) means for all c>0 there exists some k>0 such that f(n)<c.g(n) for all n>=k. Informally, f(n)=o(g(n)) means f(n) becomes insignificant relative to g(n) as n approaches infinity.

Example: f(n)=3n+4 is o(n^2)

In simple terms, f(n) has less growth rate compared to g(n).

g(n)= $2n^2$ g(n) =o(n^3), O(n^2), g(n) is not o(n^2).

Little-Omega (ω notation)

Little-omega (ω) notation is to big-omega (Ω) notation as little-o notation is to Big-Oh notation. We use ω notation to denote a lower bound that is not asymptotically tight.

Formal Definition: f(n)= ω (g(n)) if there exists a constant no>0 such that 0<= c. g(n)<f(n) for all n>=k.

Example: $2n^2$=ω(n) but it's not ω (n^2).

1.5. Relational Properties of the Asymptotic Notations

Transitivity

- if f(n)=Θ(g(n)) and g(n)= Θ(h(n)) then f(n)=Θ(h(n)),
- if f(n)=O(g(n)) and g(n)= O(h(n)) then f(n)=O(h(n)),
- if f(n)=Ω(g(n)) and g(n)= Ω(h(n)) then f(n)=Ω (h(n)),

- if f(n)=o(g(n)) and g(n)= o(h(n)) then f(n)=o(h(n)), and
- if f(n)=ω (g(n)) and g(n)= ω(h(n)) then f(n)=ω (h(n)).

Symmetry

- f(n)=Θ(g(n)) if and only if g(n)=Θ(f(n)).

Transpose symmetry

- f(n)=O(g(n)) if and only if g(n)=Ω(f(n)),
- f(n)=o(g(n)) if and only if g(n)=ω(f(n)).

Reflexivity

- f(n)=Θ(f(n)),
- f(n)=O(f(n)),
- f(n)=Ω(f(n)).

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.
- Searching techniques are ***linear search, binary search*** and ***Fibonacci Search***

LINEAR SEARCH:

- Linear search is a technique which traverse the array sequentially to locate given item or search element.
- In Linear search, we access each element of an array one by one sequentially and see weather it is desired element or not. We traverse the entire list and match each element of the list with the item whose location is to be found. If the match found then location of the item is returned otherwise the algorithm return NULL.
- A search is successful then it will return the location of desired element
- If A search will unsuccessful if all the elements are accessed and desired element not found.
- Linear search is mostly used to search an unordered list in which the items are not sorted.

Linear search is implemented using following steps...

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the first element in the list.

Step 3 - If both are matched, then display "Given element is found!!!" and terminate the function

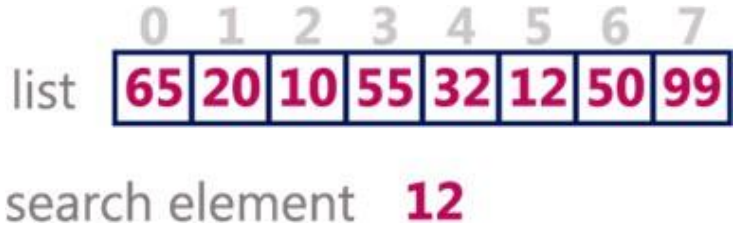
Step 4 - If both are not matched, then compare search element with the next element in the list.

Step 5 - Repeat steps 3 and 4 until search element is compared with last element in the list.

Step 6 - If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

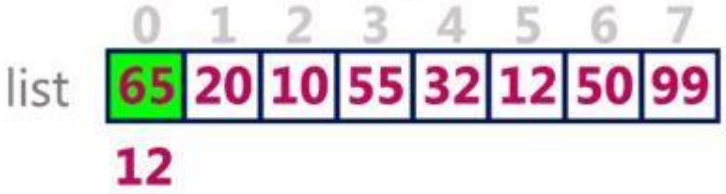
Example:

Consider the following list of elements and the element to be searched...



Step 1:

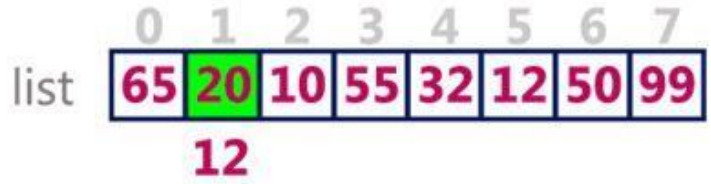
search element (12) is compared with first element (65)



Both are not matching. So move to next element

Step 2:

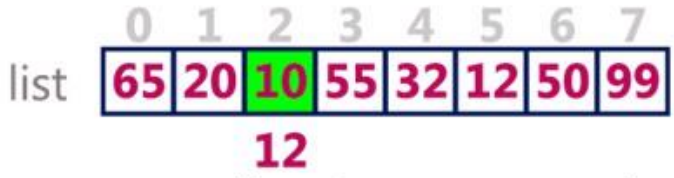
search element (12) is compared with next element (20)



Both are not matching. So move to next element

Step 3:

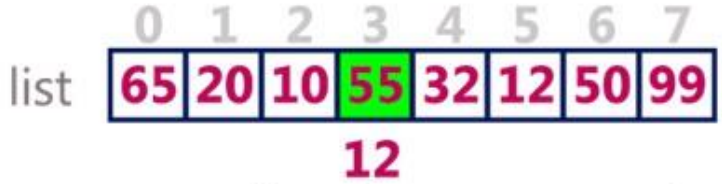
search element (12) is compared with next element (10)



Both are not matching. So move to next element

Step 4:

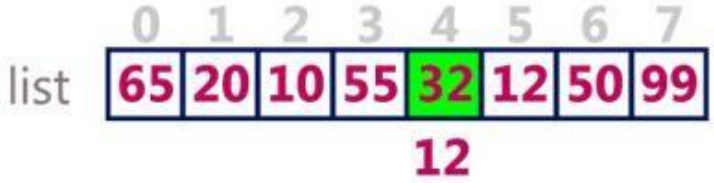
search element (12) is compared with next element (55)



Both are not matching. So move to next element

Step 5:

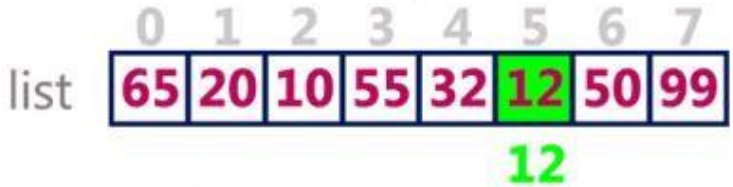
search element (12) is compared with next element (32)



Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)



Both are matching. So we stop comparing and display element found at index 5.

LINEAR SEARCH CODE:

```
#include<stdio.h>
#include<conio.h>
int main(){
   int n,i,key;
   printf("Enter the array size:\n");
   scanf("%d",&n);
   int a[n];
   printf("Enter the array elements:\n");
   for(i=0;i<n;i++){
      scanf("%d",&a[i]);
   }
   printf("Enter the key element to search:\n");
   scanf("%d",&key);
   for(i=0;i<n;i++){
      if(a[i]==key){
         printf("Element %d found at %d\n",key,i+1);
         break;
      }
   }
   if(i==n){
      printf("%d Element not found in array\n",key);
   }
   return 0;
}
```

BINARY SEARCH:

- Binary search is the search technique which works efficiently on the **sorted lists**. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.
- Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

Algorithm:

Step 1 - Read the search element from the user.

Step 2 - Find the middle element in the sorted list.

Step 3 - Compare the search element with the middle element in the sorted list.

Step 4 - If both are matched, then display "Given element is found!!!" and terminate the function.

Step 5 - If both are not matched, then check whether the search element is smaller or larger than the middle element.

Step 6 - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.

Step 7 - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.

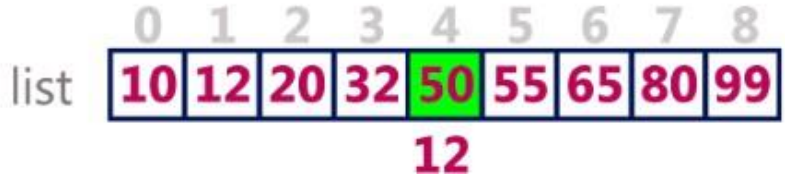
Step 8 - Repeat the same process until we find the search element in the list or until sublist contains only one element.

Step 9 - If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

Example:

Step 1:

search element (12) is compared with middle element (50)



Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

Step 2:

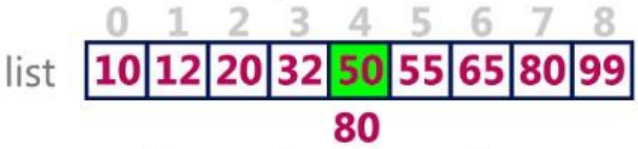
search element (12) is compared with middle element (12)

Both are matching. So the result is "Element found at index 1"

Example 2:

Step 1:

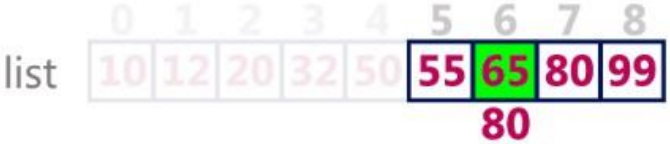
search element (80) is compared with middle element (50)



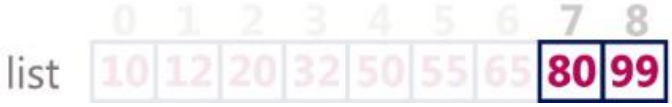
Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

Step 2:

search element (80) is compared with middle element (65)



Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).



Step 3:

search element (80) is compared with middle element (80)

Both are not matching. So the result is "Element found at index 7"

BINARY SEARCH CODE:

```
#include <stdio.h>
int main()
{
  int c, first, last, middle, n, search;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  int array[n];
  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  printf("Enter value to find\n");
  scanf("%d", &search);
  first = 0;
  last = n - 1;
  middle = (first+last)/2;
  while (first <= last) {
    if (array[middle] < search)
      first = middle + 1;
    else if (array[middle] == search) {
      printf("%d found at location %d.\n", search, middle+1);
      break;
    }
    else
      last = middle - 1;
    middle = (first + last)/2;
  }
  if (first > last)
    printf("Not found! %d isn't present in the list.\n", search);
  return 0;
}
```

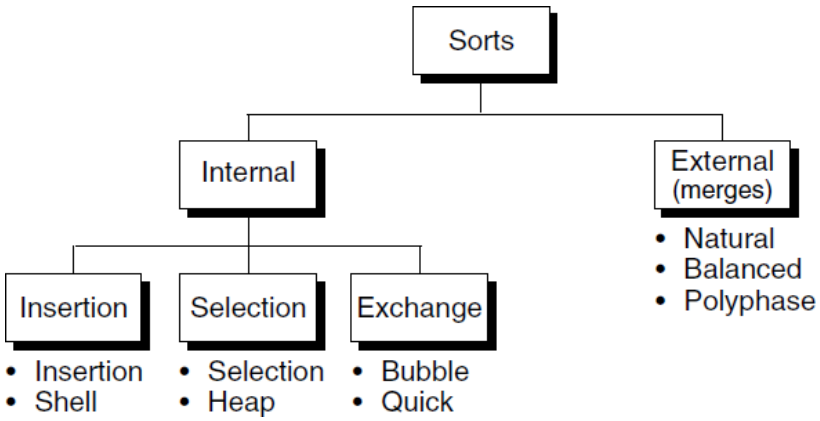
SORTINGS:

- **Definition:** Sorting is a technique to rearrange the list of records(elements) either in ascending or descending order, Sorting is performed according to some key value of each record.

Categories of Sorting:

The sorting can be divided into two categories. These are:

- Internal Sorting
- External Sorting

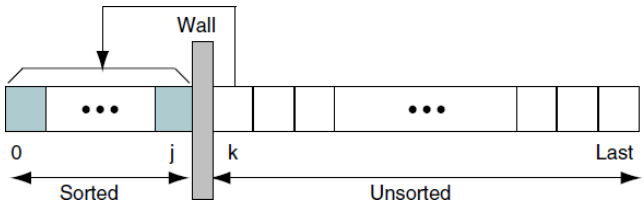

Sorts
Internal
External (merges)
Natural
Balanced
Polyphase
Insertion
Selection
Exchange
Insertion
Shell
Selection
Heap
Bubble
Quick

- **Internal Sorting:** When all the data that is to be sorted can be accommodated at a time in the main memory (Usually RAM). Internal sortings has five different classifications: insertion, selection, exchanging, merging, and distribution sort
- **External Sorting:** When all the data that is to be sorted can't be accommodated in the memory (Usually RAM) at the same time and some have to be kept in auxiliary memory such as hard disk, floppy disk, magnetic tapes etc.

  Ex: Natural, Balanced, and Polyphase.

INSERTION SORT:

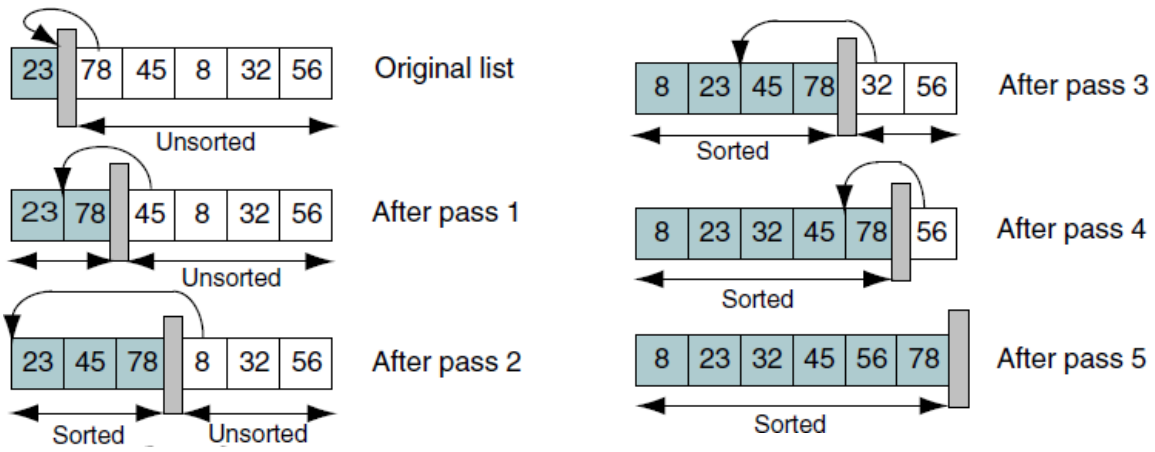
- In Insertion sort the list can be divided into two parts, one is sorted list and other is unsorted list. In each pass the first element of unsorted list is transfers to sorted list by inserting it in appropriate position or proper place.
- The similarity can be understood from the style we arrange a deck of cards. This sort works on the principle of inserting an element at a particular position, hence the name Insertion Sort.



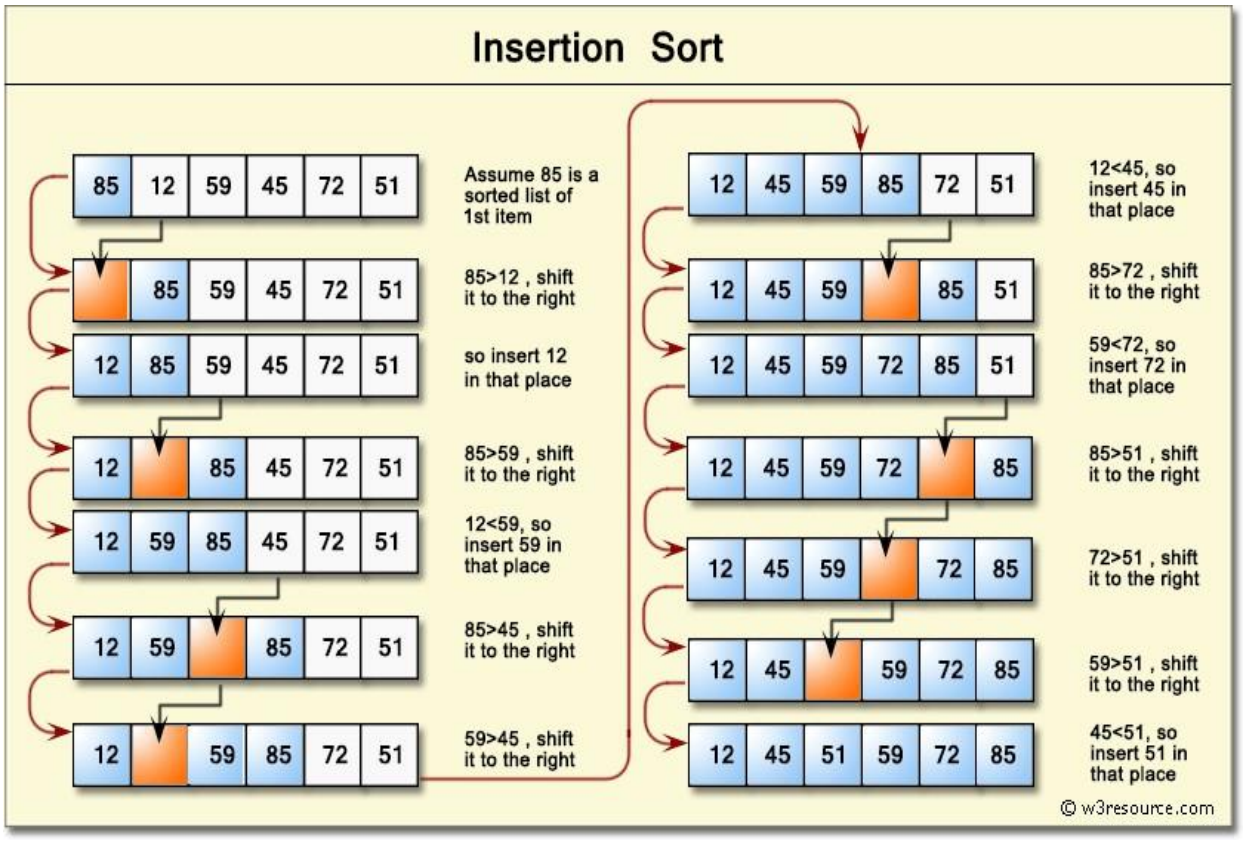
Following are the steps involved in insertion sort:

1. We start by taking the second element of the given array, i.e. element at index 1, the key. The key element here is the new card that we need to add to our existing sorted set of cards
2. We compare the key element with the element(s) before it, in this case, element at index 0:
   - If the key element is less than the first element, we insert the key element before the first element.
   - If the key element is greater than the first element, then we insert it after the first element.
3. Then, we make the third element of the array as key and will compare it with elements to it's left and insert it at the proper position.
4. And we go on repeating this, until the array is sorted.

Example 1:



Example 2:



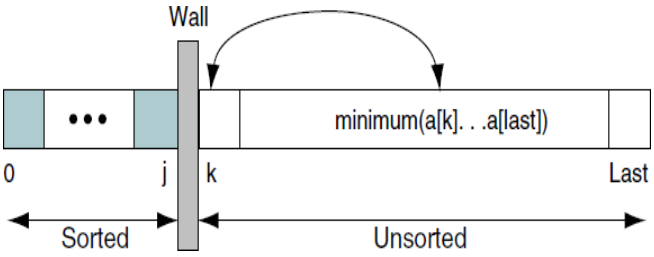
INSERTION SORT CODE:

```
#include<stdio.h>
#include<conio.h>
int main(){
   int n,i,j,key;
   printf("Enter the array size:\n");
   scanf("%d",&n);
   int a[n];
   printf("Enter the array elements:\n");
   for(i=0;i<n;i++){
      scanf("%d",&a[i]);
   }
   printf("Elements in array before sorting:\n");
   for(i=0;i<n;i++){
      printf("%d ",a[i]);
   }
  for (i = 1; i < n; i++)
 {
     key = arr[i];
     j = i - 1;
    while (j >= 0 && arr[j] > key)
   {
        arr[j + 1] = arr[j];
        j = j - 1;
   }
   arr[j + 1] = key;
}
```

```
    printf("Elements in array after sorting:\n");
    for(i=0;i<n;i++){
          printf("%d ",a[i]);
    }
    return 0;
}
```

SELECTION SORT:

- Given a list of data to be sorted, we simply select the smallest item and place it in a sorted list. These steps are then repeated until we have sorted all of the data.
- In first step, the smallest element is search in the list, once the smallest element is found, it is exchanged with the element in the first position.
- Now the list is divided into two parts. One is sorted list other is unsorted list. Find out the smallest element in the unsorted list and it is exchange with the starting position of unsorted list, after that it will added in to sorted list.



- This process is repeated until all the elements are sorted.

Ex: asked to sort a list on paper.

Algorithm:

SELECTION SORT(ARR, N)

Step 1: Repeat Steps 2 and 3 for K = 1 to N-1

Step 2: CALL SMALLEST(ARR, K, N, Loc)

Step 3: SWAP A[K] with ARR[Loc]

Step 4: EXIT

Algorithm for finding minimum element in the list.

SMALLEST (ARR, K, N, Loc)

Step 1: [INITIALIZE] SET Min = ARR[K]

Step 2: [INITIALIZE] SET Loc = K

Step 3: Repeat for J = K+1 to N

IF Min > ARR[J]

SET Min = ARR[J]

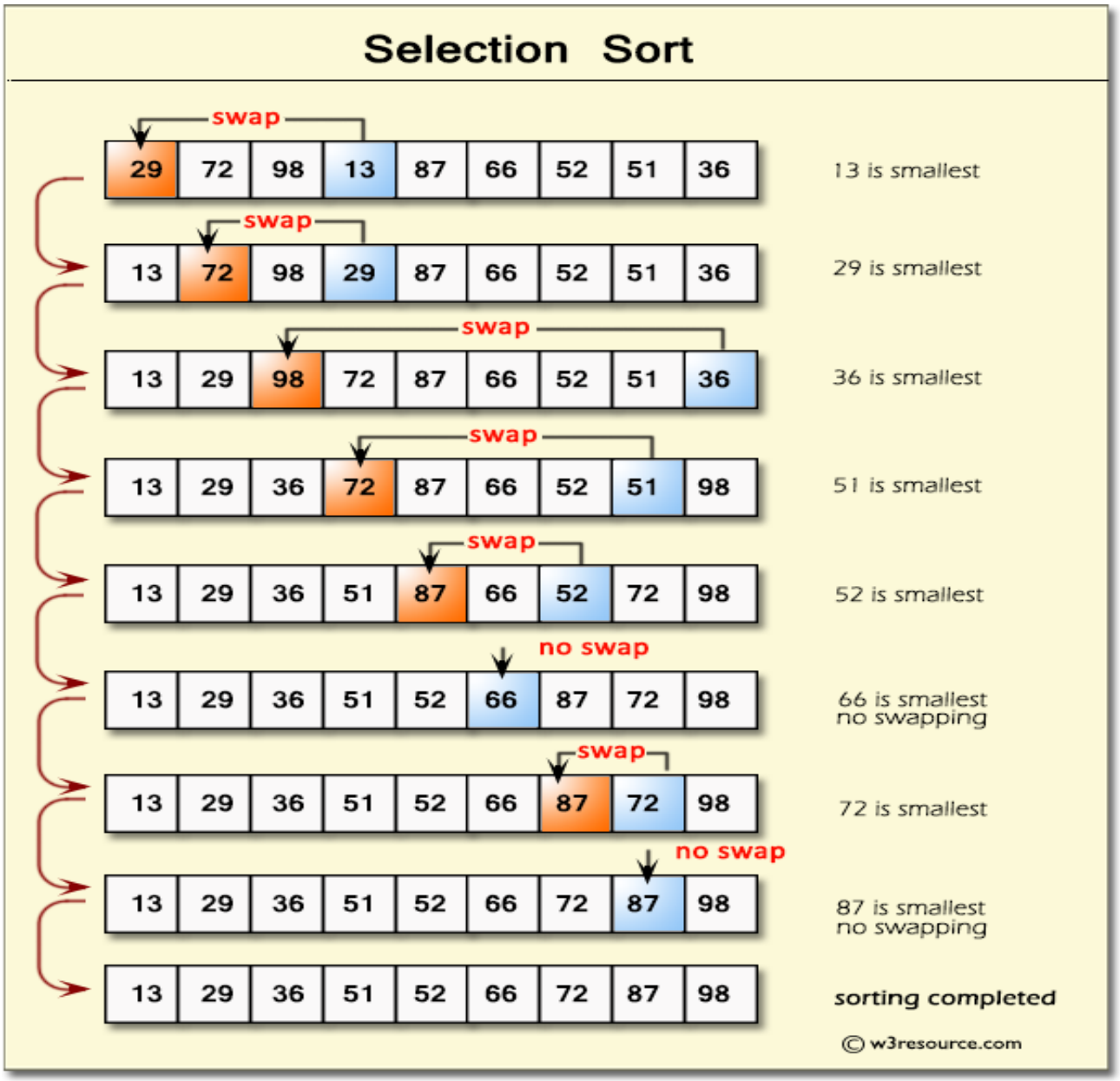
SET Loc = J

[END OF IF]

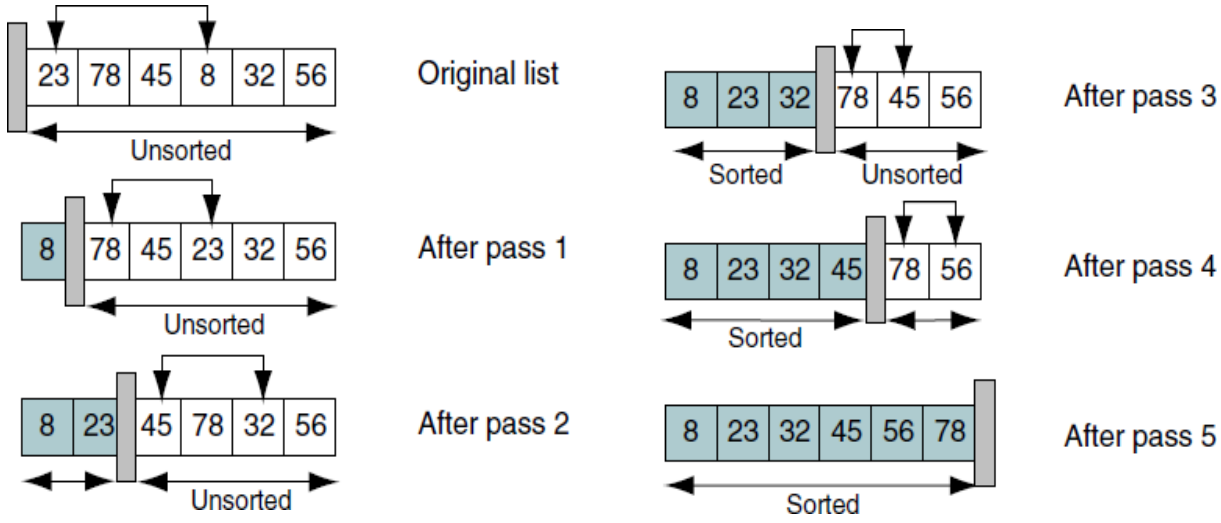
[END OF LOOP]

Step 4: RETURN Loc

Example 1:


Selection Sort
swap
29 72 98 13 87 66 52 51 36
13 is smallest
swap
13 72 98 29 87 66 52 51 36
29 is smallest
swap
13 29 98 72 87 66 52 51 36
36 is smallest
swap
13 29 36 72 87 66 52 51 98
51 is smallest
swap
13 29 36 51 87 66 52 72 98
52 is smallest
no swap
13 29 36 51 52 66 87 72 98
66 is smallest
no swapping
swap
13 29 36 51 52 66 87 72 98
72 is smallest
no swap
13 29 36 51 52 66 72 87 98
87 is smallest
no swapping
13 29 36 51 52 66 72 87 98
sorting completed
© w3resource.com

Example 2: Consider the elements 23,78,45,88,32,56



Time Complexity:

Number of elements in an array is 'N'

Number of passes required to sort is 'N-1'

Number of comparisons in each pass is 1st pass N-1, 2nd Pass N-2 …

Time required for complete sorting is:

$T(n) <= (N-1)*(N-1)$

$T(n) <= (N-1)^2$

Finally, The time complexity is $O(n^2)$.

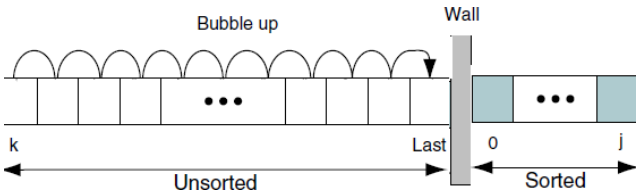
SELECTION SORT CODE:

```
#include<stdio.h>
#include<conio.h>
int main(){
   int n,i,j,position,swap;
   printf("Enter the array size:\n");
   scanf("%d",&n);
   int arr[n];
   printf("Enter the array elements:\n");
   for(i=0;i<n;i++){
      scanf("%d",&arr[i]);
   }
   printf("Elements in array before sorting:\n");
   for(i=0;i<n;i++){
      printf("%d ",arr[i]);
   }
   for (i = 0; i < (n - 1); i++) {
     position = i;
     for (j = i + 1; j < n; j++) {
       if (arr[position] > arr[j])
         position = j;
     }
     if (position != i) {
```

```
            swap = arr[i];
            arr[i] = arr[position];
            arr[position] = swap;
          }
        }
printf("Elements in array after sorting:\n");
    for(i=0;i<n;i++){
          printf("%d ",arr[i]);
    }
    return 0;
}
```

BUBBLE SORT:

- Bubble Sort is also called as Exchange Sort
- In Bubble Sort, each element is compared with its adjacent element
    a) If he first element is larger than the second element then the position of the elements are interchanged.
    b) Otherwise, the position of the elements are not changed.
    c) The same procedure is repeated until no more elements are left for comparison.
- After the 1st pass the largest element is placed at (N-1)th location. Given a list of *n* elements, the bubble sort requires up to *n* – 1 passes to sort the data.



Example 1:

- We take an unsorted array for our example.

- Bubble sort starts with very first two elements, comparing them to check which one is greater.

- In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27. We find that 27 is smaller than 33 and these two values must be swapped.

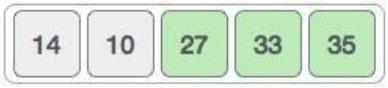
- Next we compare 33 and 35. We find that both are in already sorted positions.

- Then we move to the next two values, 35 and 10. We know then that 10 is smaller 35.

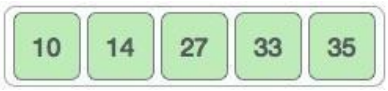
- We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

- To be defined, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this

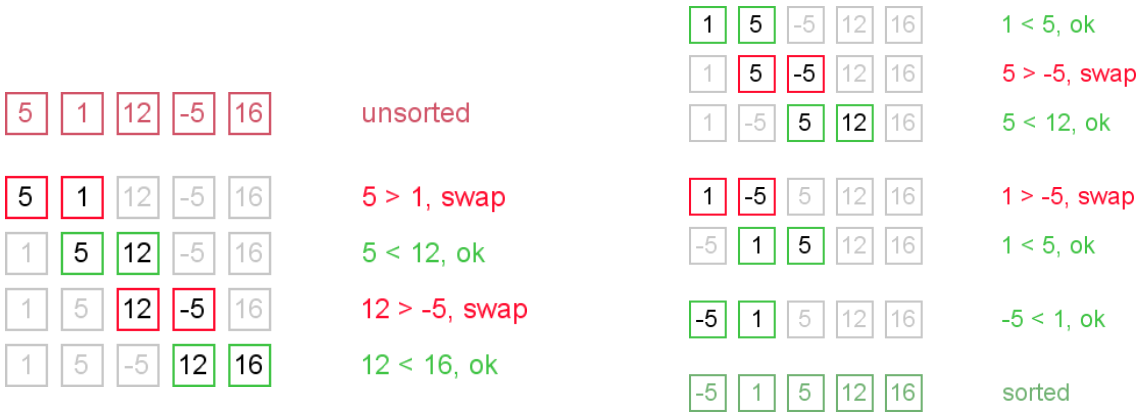
- Notice that after each iteration, at least one value moves at the end.



- And when there's no swap required, bubble sorts learns that an array is completely sorted.



Example 2:



Algorithm:

BUBBLE SORT(ARR, N)

Step 1: Read the array elements

Step 2: i:=0;

Step 3: Repeat step 4 and step 5 until i<n

Step 4: j:=0;

Step 5: Repeat step 6 until j<(n-1)-i

Step 6: if A[j] > A[j+1]

Swap(A[j],A[j+1])

End if

End loop 5

End loop 3

Step 7: EXIT

Time Complexity:

Number of elements in an array is 'N'

Number of passes required to sort is 'N-1'

Number of comparisons in each pass is 1st pass N-1, 2nd Pass N-2 …

Time required for complete sorting is:

$T(n) <= (N-1)*(N-1)$

$T(n) <= (N-1)^2$

Finally, The time complexity

is $O(n^2)$.

BUBBLE SORT CODE:

```
#include<stdio.h>
#include<conio.h>
int main(){
   int n,i,j,temp;
   printf("Enter the array size:\n");
   scanf("%d",&n);
   int arr[n];
   printf("Enter the array elements:\n");
   for(i=0;i<n;i++){
      scanf("%d",&arr[i]);
   }
   printf("Elements in array before sorting:\n");
   for(i=0;i<n;i++){
      printf("%d ",arr[i]);
   }
   for (i = 0; i < n - 1; i++) {
   for (j = 0; j < n - i - 1; j++) {
     if (arr[j] > arr[j + 1]) {
       int temp = arr[j];
       arr[j] = arr[j + 1];
       arr[j + 1] = temp;
     }
    }
  }
   printf("Elements in array after sorting:\n");
   for(i=0;i<n;i++){
        printf("%d ",arr[i]);
   }
   return 0;
}
```

QUICK SORT:

- Quick sort follows **Divide and Conquer** algorithm. It is dividing array in to smaller parts based on partitioning and performing the sort operations on those divided smaller parts. Hence, it works well for large datasets.

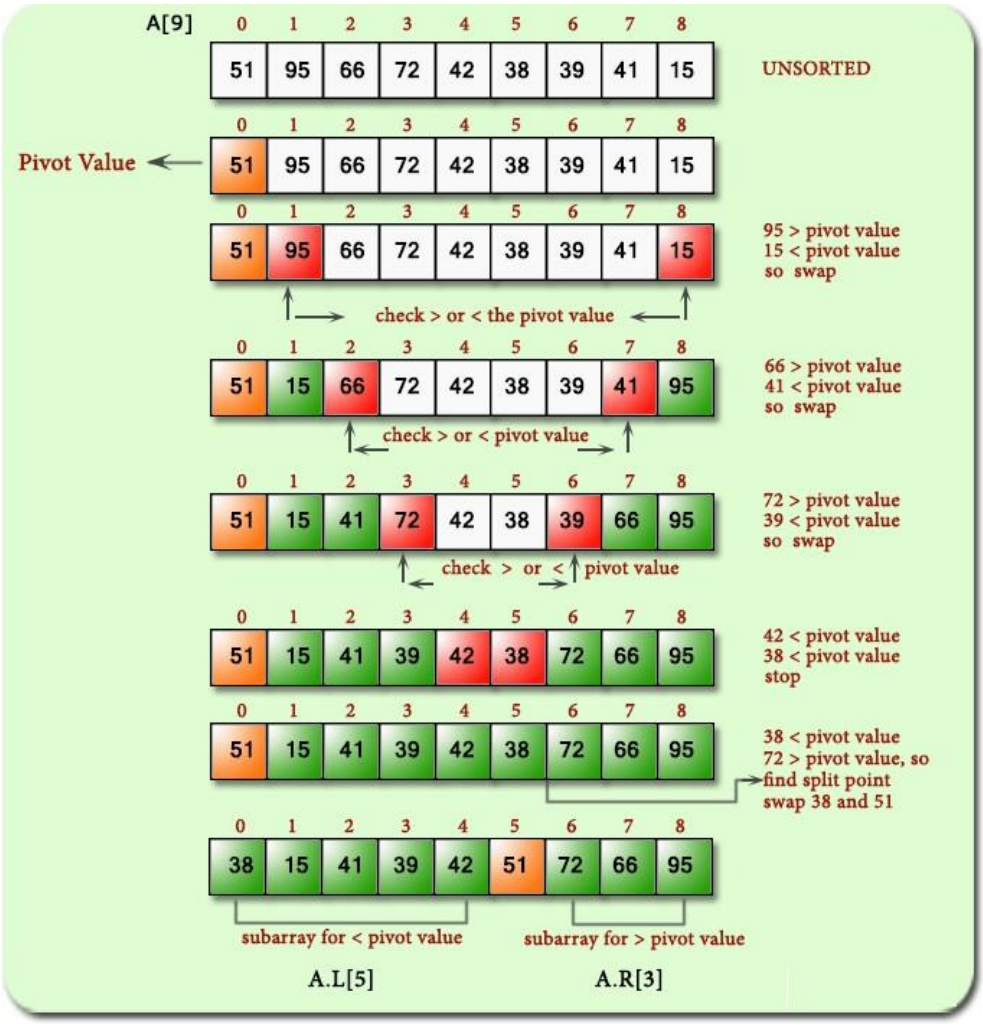
So, here are the steps **how Quick sort** works in simple words.

1. First select an element which is to be called as **pivot** element.
2. Next, compare all array elements with the selected pivot element and arrange them in such a way that, elements less than the pivot element are to its left and greater than pivot is to it's right.
3. Finally, perform the same operations on left and right side elements to the pivot element.

How does Quick Sort Partitioning Work

1. First find the **"pivot"** element in the array.
2. Start the left pointer at first element of the array.
3. Start the right pointer at last element of the array.
4. Compare the element pointing with left pointer and if it is less than the pivot element, then move the left pointer to the right (add 1 to the left index). Continue this until left side element is greater than or equal to the pivot element.
5. Compare the element pointing with right pointer and if it is greater than the pivot element, then move the right pointer to the left (subtract 1 to the right index). Continue this until right side element is less than or equal to the pivot element.
6. Check if left pointer is less than or equal to right pointer, then swap the elements in locations of these pointers.
7. Check if index of left pointer is greater than the index of the right pointer, then swap pivot element with right pointer.

Example:



Algorithm:

```
quickSort(array, lb, ub)
{
   if(lb< ub)
   {
      pivotIndex = partition(arr, lb, ub);
      quickSort(arr, lb, pIndex - 1);
      quickSort(arr, pivotIndex+1, ub);
   }
}
```

QUICK SORT CODE:

```
#include<stdio.h>
#include<conio.h>
void swap(int* a, int* b) {
   int t = *a;
   *a = *b;
   *b = t;
}
int partition(int arr[], int low, int high) {
   int pivot = arr[high];
   int i = (low - 1);

   for (int j = low; j <= high - 1; j++) {
      if (arr[j] < pivot) {
```

```
            i++;
            swap(&arr[i], &arr[j]);
          }
       }
       swap(&arr[i + 1], &arr[high]);
       return (i + 1);
    }
    void quicksort(int arr[], int low, int high) {
       if (low < high) {
          int pi = partition(arr, low, high);
          quicksort(arr, low, pi - 1);
          quicksort(arr, pi + 1, high);
       }
    }
    int main(){
       int n,i,j,temp;
       printf("Enter the array size:\n");
       scanf("%d",&n);
       int arr[n];
       printf("Enter the array elements:\n");
       for(i=0;i<n;i++){
          scanf("%d",&arr[i]);
       }
       printf("Elements in array before sorting:\n");
       for(i=0;i<n;i++){
          printf("%d ",arr[i]);
       }
       quicksort(arr,0,n-1);
    printf("Elements in array after sorting:\n");
    for(i=0;i<n;i++){
          printf("%d ",arr[i]);
    }
    return 0;
}
```

MERGE SORT:

Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

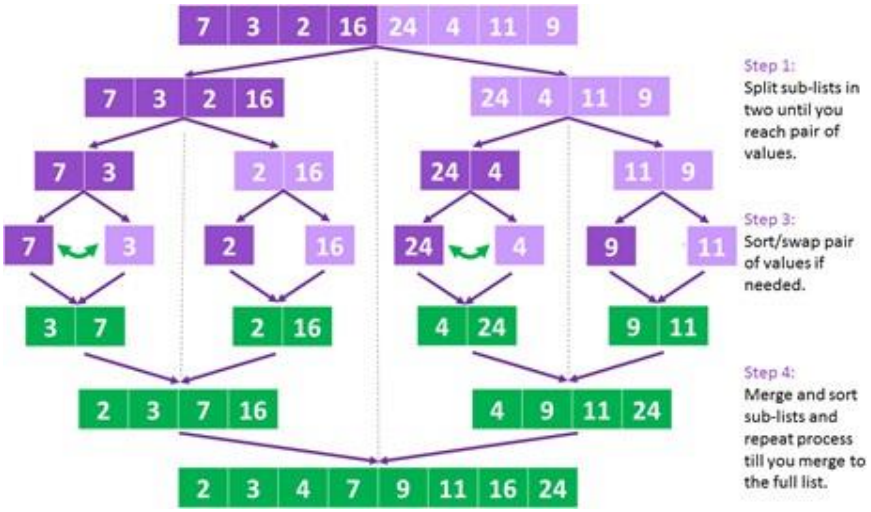
Implementation Recursive Merge Sort:

- The merge sort starts at the Top and proceeds downwards, "split the array into two, make a recursive call, and merge the results.", until one gets to the bottom of the array-tree.

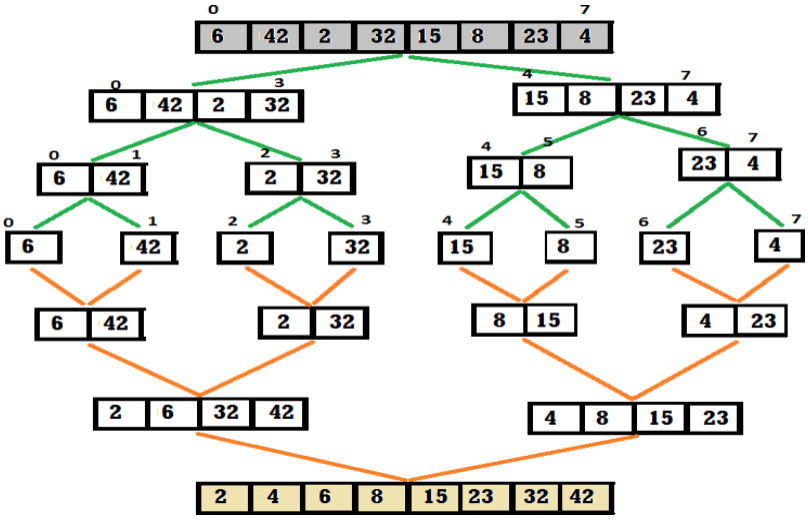
Example: Let us consider an example to understand the approach better.

1. Divide the unsorted list into n sub-lists based on mid value, each array consisting 1 element
2. Repeatedly merge sub-lists to produce newly sorted sub-lists until there is only 1 sub-list remaining. This will be the sorted list

Recursive Mere Sort Example:



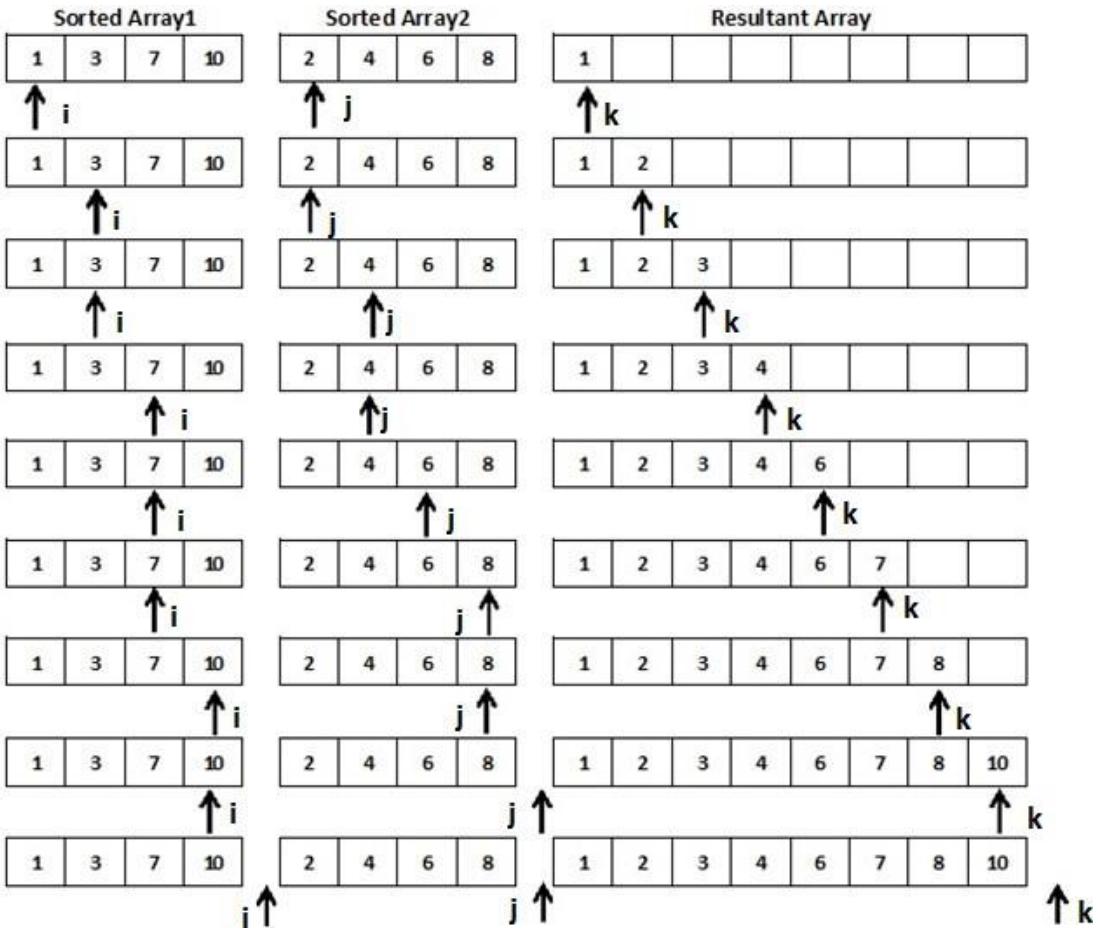
Example 2:



MergeSort Algoritm:

```
MergeSort(A, lb, ub )
{
   If  lb<ub
        {
           mid = floor(lb+ub)/2;
           mergeSort(A, lb, mid)
           mergeSort(A, mid+1, ub)
           merge(A, lb, ub , mid)
        }
}
```

Two- Way Merge Sort:



Merge Algorithm:

Step 1: set i,j,k=0

Step 2: if A[i]<B[j] then

copy A[i] to C[k] and increment i and k

else

copy B[j] to C[k] and increment j and k

Step 3: copy remaining elements of either A or B into Array C.

MERGE SORT CODE:

```
#include<stdio.h>
#include<conio.h>
void merge(int arr[], int l, int m, int r) {
   int i, j, k;
   int n1 = m - l + 1;
   int n2 = r - m;
   int L[n1], R[n2];
   for (i = 0; i < n1; i++)
      L[i] = arr[l + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[m + 1 + j];
   i = 0;
   j = 0;
   k = l;
   while (i < n1 && j < n2) {
```

```
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
void mergesort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergesort(arr, l, m);
        mergesort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
int main(){
    int n,i,j,temp;
    printf("Enter the array size:\n");
    scanf("%d",&n);
    int arr[n];
    printf("Enter the array elements:\n");
    for(i=0;i<n;i++){
        scanf("%d",&arr[i]);
    }
    printf("Elements in array before sorting:\n");
    for(i=0;i<n;i++){
        printf("%d ",arr[i]);
    }
    mergesort(arr,0,n-1);
printf("Elements in array after sorting:\n");
for(i=0;i<n;i++){
        printf("%d ",arr[i]);
}
 return 0;
}
```

Time Complexities All the Searching & Sorting Techniques:

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)