SRI MANAKULA VINAYAGAR ENGINEERING COLLEGE
(An Autonomous Institution)
(Approved by AICTE, New Delhi & Affiliated to Pondicherry University)
(Accredited by NBA-AICTE, New Delhi, ISO 9001:2000 Certified Institution & Accredited by NAAC with "A" Grade)
Madagadipet, Puducherry - 605 107

DEPARTMENT COMPUTER SCIENCE AND ENGINEERING

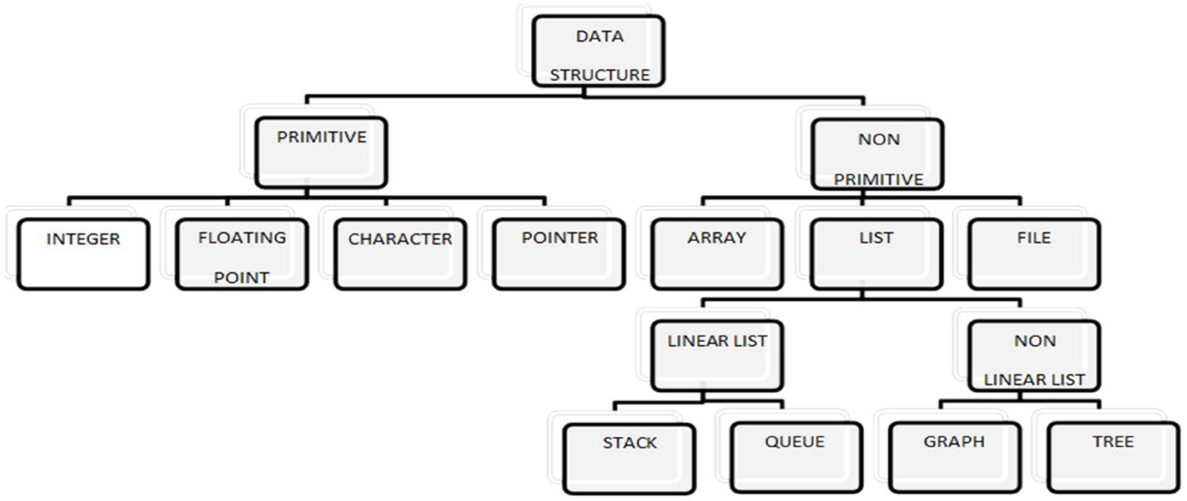
Subject Name: Data Structures
Subject Code: U20EST356

UNIT – I

Introduction: Basic Terminologies: Elementary Data Organizations. Data Structure Operations: insertion, deletion, traversal. Analysis of an Algorithm, Asymptotic Notations, Time-Space trade off. Array and its operations. Searching: Linear Search and Binary Search Techniques and their complexity analysis.

2 Marks

1. **What is Data Structure?**
   - Data structure is a representation of the logical relationship existing between individual elements of data.
   - Data Structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.
   - We can also define data structure as a mathematical or logical model of a particular organization of data items.
   - The representation of particular data structure in the main memory of a computer is called as storage structure.
2. **Draw the classification of data structure?**



3. **What are the two main categories of data structure?**
   1. Primitive Data Structure
   2. Non-primitive data Structure
4. **Define datatype?**
   A particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it.

5. **Define primitive data structure?**
   - Primitive data structures are basic structures and are directly operated upon by machine instructions.
   - Primitive data structures have different representations on different computers.
   - Integers, floats, character and pointers are examples of primitive data structures.

- These data types are available in most programming languages as built in type.

6. **Define non-primitive data structure?**
    - These are more sophisticated data structures.
    - These are derived from primitive data structures.
    - The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items.
7. **List the types of non-primitive data structure?**

    A Non-primitive data type is further divided into Linear and Non-Linear data structure
8. **Define linear data structure?**

- A data structure is said to be Linear, if its elements are connected in linear fashion by means of logically or in sequence memory locations.
- Examples of Linear Data Structure are Stack and Queue.

9. **Define non-linear data structure?**
    - Nonlinear data structures are those data structure in which data items are not arranged in a sequence.
    - Examples of Non-linear Data Structure are Tree and Graph.
10. **What are the ways to represent linear data structure in memory?**
    - There are two ways to represent a linear data structure in memory,
        - o Static memory allocation
        - o Dynamic memory allocation
11. **Define stack?**

    Stack: Stack is a data structure in which insertion and deletion operations are performed at one end only. The insertion operation is referred to as 'PUSH' and deletion operation is referred to as 'POP' operation. Stack is also called as Last in First out (LIFO) data structure.
12. **Define queue?**

    Queue: The data structure which permits the insertion at one end and Deletion at another end, known as Queue. End at which deletion is occurs is known as FRONT end and another end at which insertion occurs is known as REAR end. Queue is also called as First in First out (FIFO) data structure.
13. **Define tree?**

    Tree: A tree can be defined as finite set of data items (nodes) in which data items are arranged in branches and sub branches according to requirement.
    - o Trees represent the hierarchical relationship between various elements.
    - o Tree consist of nodes connected by edge, the node represented by circle and edge lives connecting to circle.
14. **Define graph?**

    Graph is a collection of nodes (Information) and connecting edges (Logical relation) between nodes.
    - o A tree can be viewed as restricted graph.
15. **List various types of graph?**

    Graphs have many types:
    - Un-directed Graph
    - Directed Graph
    - Mixed Graph
    - Multi Graph
    - Simple Graph
    - Null Graph
    - Weighted Graph
16. **Difference between linear and non-linear data structure?**

Linear Data Structure	Non-Linear Data Structure

Every item is related to its previous and next time.	Every item is attached with many other items.
Data is arranged in linear sequence.	Data is not arranged in sequence.
Data items can be traversed in a single run.	Data cannot be traversed in a single run.
Eg. Array, Stacks, linked list, queue.	Eg. tree, graph.
Implementation is easy.	Implementation is difficult.

17. **What are the various operations carried out on data structures?**
    Create, destroy, selection, updation, searching, sorting, merging, splitting, traversal
18. **Define algorithm?**
    - An essential aspect to data structures is algorithms.
    - Data structures are implemented using algorithms.
    - An algorithm is a procedure that you can write as a C function or program, or any other language.
    - An algorithm states explicitly how the data will be manipulated.
19. **What is complexity of an algorithm**?
    - Some algorithms are more efficient than others. We would prefer to choose an efficient algorithm, so it would be nice to have metrics for comparing algorithm efficiency.
    - The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process.
20. **Define time complexity?**
    - Time Complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.
    - "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.
21. **Define space complexity?**
    - Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.
    - Space complexity is sometimes ignored because the space used is minimal and/or obvious, but sometimes it becomes as important an issue as time.
22. **Define worst case analysis?**
    In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched is not present in the array. When x is not present, the search () functions compares it with all the elements of array [] one by one. Therefore, the worst case time complexity of linear search would be
23. **Define best case analysis?**
    In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in worst case is constant (not dependent on n). So time complexity in the best case would be.
24. **Define average case analysis?**

    In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed. So we sum all the cases and divide the sum by (n+1).

25. **Define array with example?**

- An array is a collection of elements of the same type that are referenced by a common name.
- Compared to the basic data type (int, float & char) it is an aggregate or derived data type.
- All the elements of an array occupy a set of contiguous memory locations.
  Example: int studMark[1000];

26. What is two dimensional array with example?

A two dimensional array has two subscripts/indexes. The first subscript refers to the row, and the second, to the column. Its declaration has the following form,

data_type array_name[1st dimension size][2nd dimension size];
For examples,
int xInteger[3][4];
float matrixNum[20][25];

27. What are the basic operations supported by array?

Traverse, insertion, deletion, search and update.

28. What is searching and its types?

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories: sequential and interval.

29. Define linear search?

Linear search in C is to find whether a number is present in an array. If it's present, then at what location it occurs. It is also known as a sequential search. It is straightforward and works as follows: we compare each element with the element to search until we find it or the list ends.

30. Define binary search?

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

5 marks:

1. Explain various operations of data structure briefly?

Design of efficient data structure must take operations to be performed on the data structures into account. The most commonly used operations on data structure are broadly categorized into following types

- **Create**
  The create operation results in reserving memory for program elements. This can be done by declaration statement. Creation of data structure may take place either during compile-time or run-time. malloc() function of C language is used for creation.

- **Destroy**
  Destroy operation destroys memory space allocated for specified data structure. free() function of C language is used to destroy data structure.
- **Selection**
  Selection operation deals with accessing a particular data within a data structure.

- **Updation**
  It updates or modifies the data in the data structure.
- **Searching**

It finds the presence of desired data item in the list of data items, it may also find the locations of all elements that satisfy certain conditions.

- **Sorting**
  Sorting is a process of arranging all data items in a data structure in a particular order, say for example, either in ascending order or in descending order.
- **Merging**
  Merging is a process of combining the data items of two different sorted list into a single sorted list.
- **Splitting**
  Splitting is a process of partitioning single list to multiple list.
- **Traversal**
  Traversal is a process of visiting each and every node of a list in systematic manner.

2. Explain the three types of algorithm analysis?

Worst Case Analysis:

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched is not present in the array. When x is not present, the search () functions compares it with all the elements of array [] one by one. Therefore, the worst case time complexity of linear search would be.

Average Case Analysis:

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed. So we sum all the cases and divide the sum by (n+1).

Best Case Analysis:

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in worst case is constant (not dependent on n). So time complexity in the best case would be.

3. Explain two dimensional array?

A two dimensional array has two subscripts/indexes. The first subscript refers to the row, and the second, to the column. Its declaration has the following form,

```
data_type    array_name[1st dimension size][2nd dimension size];
```

For examples,

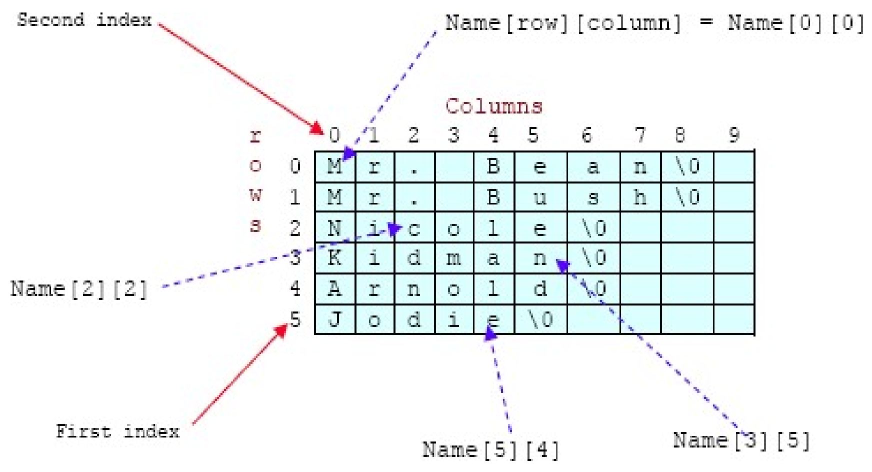
```
int      xInteger[3][4];
float    matrixNum[20][25];
```

The first line declares xInteger as an integer array with 3 rows and 4 columns.
Second line declares a matrixNum as a floating-point array with 20 rows and 25 columns.

If we assign initial string values for the 2D array it will look something like the following,

```
char Name[6][10] = {"Mr. Bean", "Mr. Bush", "Nicole", "Kidman", "Arnold", "Jodie"};
```

Here, we can initialize the array with 6 strings, each with maximum 9 characters long. If depicted in rows and columns it will look something like the following and can be considered as contiguous arrangement in the memory.



- Take note that for strings the null character (\0) still needed.
- From the shaded square area of the figure we can determine the size of the array.
- For an array Name[6][10], the array size is 6 x 10 = 60 and equal to the number of the colored square. In general, for

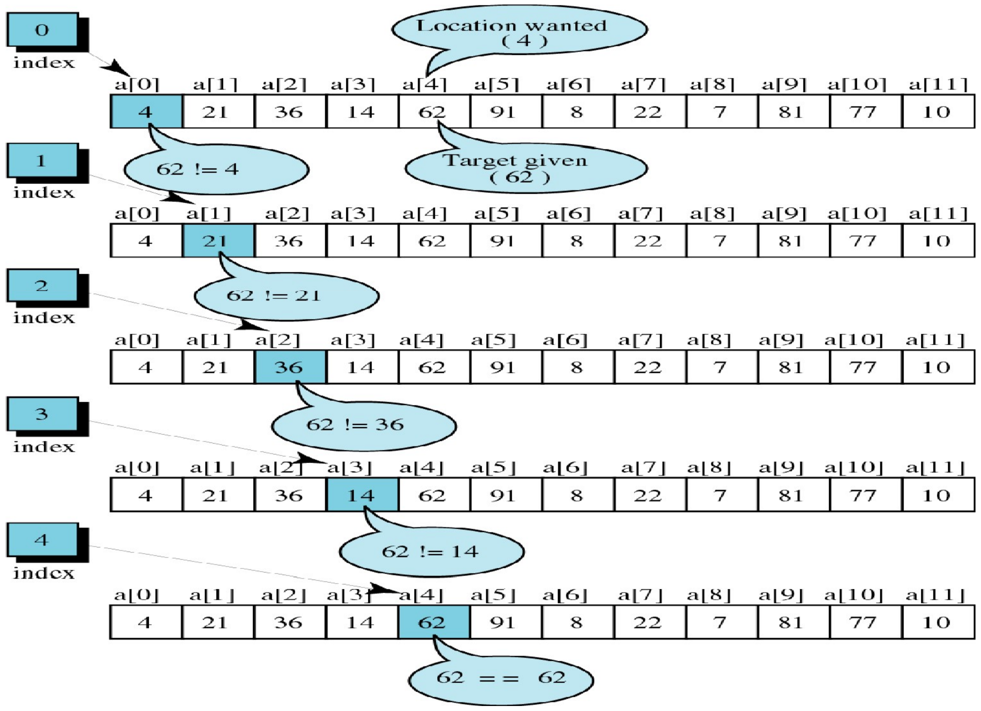
  array_name[x][y];
- The array size is = First index x second index = xy.
- This also true for other array dimension, for example three dimensional array,
- array_name[x][y][z]; => First index x second index x third index = xyz
- For example,

  ThreeDimArray[2][4][7] = 2 x 4 x 7 = 56.

  And if you want to illustrate the 3D array, it could be a cube with wide, long and height dimensions.

4. Explain linear search with sample program?

Linear search in C is to find whether a number is present in an array. If it's present, then at what location it occurs. It is also known as a sequential search. It is straightforward and works as follows: we compare each element with the element to search until we find it or the list ends.



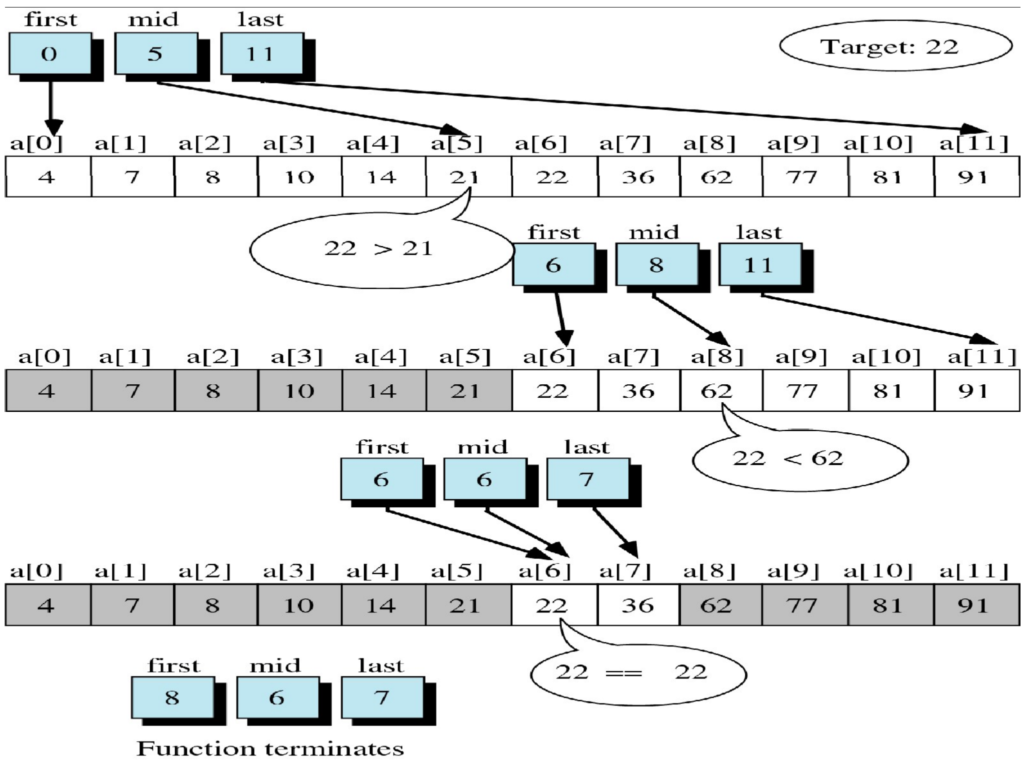
Above diagram shows the simple example to find the 62 in the given array using linear search.

Program:

```
#include <stdio.h>
int main()
{
  int array[100], search, c, n;
  printf("Enter number of elements in array\n");
  scanf("%d", &n);
  printf("Enter %d integer(s)\n", n);
  for (c = 0; c < n; c++)
  scanf("%d", &array[c]);
  printf("Enter a number to search\n");
  scanf("%d", &search);
  for (c = 0; c < n; c++)
  {
   if (array[c] == search)    /* If required element is found */
 {
   printf("%d is present at location %d.\n", search, c+1);
  break;
  }
 }
if (c == n)
 printf("%d isn't present in the array.\n", search);
return 0;
}
```

5. **Explain binary search with sample program?**

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.



The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.
5. Search an ordered array of integers for a value and return its index if the value is found. Otherwise, return -1.

The above example shows the steps to find the element 62 in a given array using binary search.

Binary search is based on the “divide-and-conquer” strategy which works as follows:

n Start by looking at the middle element of the array

– 1. If the value it holds is lower than the search element, eliminate the first half of the array from further consideration.

– 2. If the value it holds is higher than the search element, eliminate the second half of the array from further consideration.

Repeat this process until the element is found, or until the entire array has been eliminated

Algorithm:

```
Set first and last boundary of array to be searched
Repeat the following:
            Find middle element between first and last boundaries;
            if (middle element contains the search value)
        return middle_element position;
            else if (first >= last )
        return –1;
            else if (value < the value of middle_element)
                        set last to middle_element position – 1;
            else
        set first to middle_element position + 1;
```

Program : Binary Search

```
#include <stdio.h>
int main()
{
  int i, first, last, middle, n, search, array[100];
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (i = 0; i < n; i++)
                        scanf("%d", &array[i]);
            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;
}
```

6. Explain time complexity and space complexity?

Algorithm Efficiency

□ Some algorithms are more efficient than others. We would prefer to choose an efficient algorithm, so it would be nice to have metrics for comparing algorithm efficiency.

□ The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process.

□ Usually there are natural units for the domain and range of this function. There are two main complexity measures of the efficiency of an algorithm

□ Time complexity

□ Time Complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.

□ "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.

□ Space complexity

□ Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.

□ We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this.

□ We can use bytes, but it's easier to use, say, number of integers used, number of fixed-sized structures, etc. In the end, the function we come up with will be independent of the actual number of bytes needed to represent the unit.

□ Space complexity is sometimes ignored because the space used is minimal and/or obvious, but sometimes it becomes as important an issue as time.

7. Explain single dimension array?

□ An array is a collection of elements of the same type that are referenced by a common name.

□ Compared to the basic data type (int, float & char) it is an aggregate or derived data type.

□ All the elements of an array occupy a set of contiguous memory locations.

One Dimensional Array: Declaration

Dimension refers to the array's size, which is how big the array is. A single or one dimensional array declaration has the following form,

array_element_data_type   array_name[array_size];

Here, array_element_data_type define the base type of the array, which is the type of each element in the array. array_name is any valid C / C++ identifier name that obeys the same rule for the identifier naming. array_size defines how many elements the array will hold.
For example, to declare an array of 30 characters, that construct a people name, we could declare,
char cName[30];
Which can be depicted as follows,
In this statement, the array character can store up to 30 characters with the first character occupying location cName[0] and the last character occupying cName[29]. Note that the index runs from 0 to 29. In C, an index always starts from 0 and ends with array's (size-1). So, take note the difference between the array size and subscript/index terms.

Examples of the one-dimensional array declarations,
```
int      xNum[20], yNum[50];
float    fPrice[10], fYield;
char     chLetter[70];
```
The first example declares two arrays named xNum and yNum of type int. Array xNum can store up to 20 integer numbers while yNum can store up to 50 numbers. The second line declares the array fPrice of type float. It can store up to 10 floating-point values. fYield is basic variable which shows array type can be declared together with basic type provided the type is similar. The third line declares the array chLetter of type char. It can store a string up to 69 characters. Why 69 instead of 70? Remember, a string has a null terminating character (\0) at the end, so we must reserve for it.

Array Initialization

- An array may be initialized at the time of declaration.
- Giving initial values to an array.
- Initialization of an array may take the following form,
- type array_name[size] = {a_list_of_value};
- For example:
- int idNum[7] = {1, 2, 3, 4, 5, 6, 7};
- float fFloatNum[5] = {5.6, 5.7, 5.8, 5.9, 6.1};
- char chVowel[6] = {'a', 'e', 'i', 'o', 'u', '\0'};
- The first line declares an integer array idNum and it immediately assigns the values 1, 2, 3, ..., 7 to idNum[0], idNum[1], idNum[2],..., idNum[6] respectively.
- The second line assigns the values 5.6 to fFloatNum[0], 5.7 to fFloatNum[1], and so on.
- Similarly the third line assigns the characters 'a' to chVowel[0], 'e' to chVowel[1], and so on. Note again, for characters we must use the single apostrophe/quote (') to enclose them.
- Also, the last character in chVowel is NULL character ('\0').

Initialization of an array of type char for holding strings may take the following form,
char array_name[size] = "string_lateral_constant";
□ For example, the array chVowel in the previous example could have been written more compactly as follows,
char chVowel[6] = "aeiou";
□ When the value assigned to a character array is a string (which must be enclosed in double quotes), the compiler automatically supplies the NULL character but we still have to reserve one extra place for the NULL.
□ For unsized array (variable sized), we can declare as follow,
char chName[ ] = "Mr. Dracula";
□ C compiler automatically creates an array which is big enough to hold all the initializer.

8. Explain little O and little ω notations?

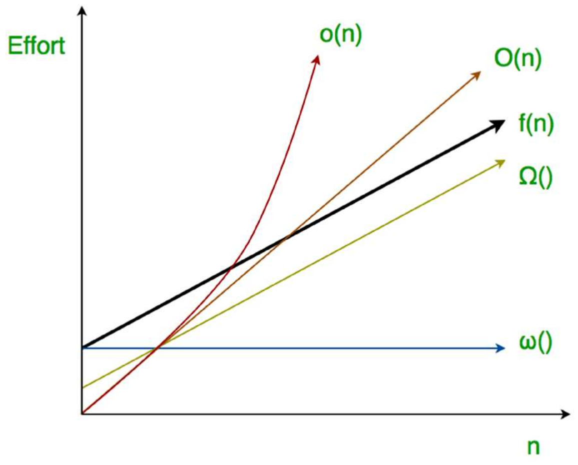
The following 2 more asymptotic notations are used to represent time complexity of algorithms.

Little o asymptotic notation

Big-O is used as a tight upper-bound on the growth of an algorithm's effort (this effort is described by the function f(n)), even though, as written, it can also be a loose upper-bound. "Little-o" (o()) notation is used to describe an upper-bound that cannot be tight.

Definition : Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is o(g(n)) (or f(n) E o(g(n))) if for any real constant $c > 0$, there exists an integer constant $n0 \geq 1$ such that $0 \leq f(n) < c*g(n)$.

Thus, little o() means loose upper-bound of f(n). Little o is a rough estimate of the maximum order of growth whereas Big-O may be the actual order of growth.



In mathematical relation,

f(n) = o(g(n)) means

lim f(n)/g(n) = 0

n→∞

Examples:

Is 7n + 8 ∈ o(n2)?

In order for that to be true, for any c, we have to be able to find an n0 that makes

f(n) < c * g(n) asymptotically true.

lets took some example,

If c = 100,we check the inequality is clearly true. If c = 1/100 , we'll have to use a little more imagination, but we'll be able to find an n0. (Try n0 = 1000.) From these examples, the conjecture appears to be correct.

then check limits,

lim f(n)/g(n) = lim (7n + 8)/(n2) = lim 7/2n = 0 (l'hospital)

n→∞ n→∞ n→∞

hence 7n + 8 ∈ o(n2)

Little ω asymptotic notation

Definition : Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is ω(g(n)) (or f(n) ∈ ω(g(n))) if for any real constant $c > 0$, there exists an integer constant $n0 \geq 1$ such that $f(n) > c * g(n) \geq 0$ for every integer $n \geq n0$.

f(n) has a higher growth rate than g(n) so main difference between Big Omega (Ω) and little omega (ω) lies in their definitions.In the case of Big Omega f(n)=Ω(g(n)) and the bound is 0<=cg(n)<=f(n), but in case of little omega, it is true for 0<=c*g(n)<f(n).

The relationship between Big Omega (Ω) and Little Omega (ω) is similar to that of Big-O and Little o except that now we are looking at the lower bounds. Little Omega (ω) is a rough estimate of the order of the growth whereas Big Omega (Ω) may represent exact order of growth. We use ω notation to denote a lower bound that is not asymptotically tight.
And, f(n) ∈ ω(g(n)) if and only if g(n) ∈ o((f(n)).

In mathematical relation,
if f(n) ∈ ω(g(n)) then,

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

Example:
Prove that 4n + 6 ∈ ω(1);
the little omega(o) running time can be proven by applying limit formula given below.
if $\lim_{n\to\infty} f(n)/g(n) = \infty$ then functions f(n) is ω(g(n))
here,we have functions f(n)=4n+6 and g(n)=1

$$\lim_{n\to\infty} (4n+6)/(1) = \infty$$

and,also for any c we can get n0 for this inequality 0 <= c*g(n) < f(n), 0 <= c*1 < 4n+6
Hence proved.

9. Explain lower bound and upper bound theory of algorithm?

The Lower and Upper Bound Theory provides a way to find the lowest complexity algorithm to solve a problem. Before understanding the theory, first lets have a brief look on what actually Lower and Upper bounds are.

Lower Bound –
Let L(n) be the running time of an algorithm A(say), then g(n) is the Lower Bound of A if there exist two constants C and N such that L(n) >= C*g(n) for n > N. Lower bound of an algorithm is shown by the asymptotic notation called Big Omega (or just Omega).
Upper Bound –
Let U(n) be the running time of an algorithm A(say), then g(n) is the Upper Bound of A if there exist two constants C and N such that U(n) <= C*g(n) for n > N. Upper bound of an algorithm is shown by the asymptotic notation called Big Oh(O) (or just Oh).
1. Lower Bound Theory:
According to the lower bound theory, for a lower bound L(n) of an algorithm, it is not possible to have any other algorithm (for a common problem) whose time complexity is less than L(n) for random input. Also every algorithm must take at least L(n) time in worst case. Note that L(n) here is the minimum of all the possible algorithm, of maximum complexity.

The Lower Bound is a very important for any algorithm. Once we calculated it, then we can compare it with the actual complexity of the algorithm and if their order are same then we can declare our algorithm as optimal. So in this section we will be discussing about techniques for finding the lower bound of an algorithm.

Note that our main motive is to get an optimal algorithm, which is the one having its Upper Bound Same as its Lower Bound (U(n)=L(n)). Merge Sort is a common example of an optimal algorithm.

Trivial Lower Bound –
It is the easiest method to find the lower bound. The Lower bounds which can be easily observed on the basis of the number of input taken and the number of output produces are called Trivial Lower Bound.

Example: Multiplication of n x n matrix, where,

Input: For 2 matrix we will have 2n2 inputs
Output: 1 matrix of order n x n, i.e., n2 outputs
In the above example its easily predictable that the lower bound is O(n2).

Computational Model –
The method is for all those algorithms that are comparison based. For example in sorting we have to compare the elements of the list among themselves and then sort them accordingly. Similar is the case with searching and thus we can implement the same in this case. Now we will look at some examples to understand its usage.

Ordered Searching –
It is a type of searching in which the list is already sorted.

Example-1: Linear search
Explanation –
In linear search we compare the key with first element if it does not match we compare with second element and so on till we check against the nth element. Else we will end up with a failure.
Using Lower bond theory to solve algebraic problem:

Straight Line Program –
The type of programs build without any loops or control structures is called Straight Line Program. For example,

```
//summing to nos
Sum(a, b)
{
        //no loops and no control structures
        c:= a+b;
        return c;
}
```

Algebraic Problem –
Problems related to algebra like solving equations inequalities etc., comes under algebraic problems. For example, solving equation ax2+bx+c with simple programming.

```
Algo_Sol(a, b, c, x)
{
        //1 assignment
        v:=a*x;

        //1 assignment
        v:=v+b;

        //1 assignment
        v:=v*x;

        //1 assignment
```

```
          ans:=v+c;
          return ans;
}
```

The above example shows us a simple way to solve an equation for 2 degree polynomial i.e., 4 thus for nth degree polynomial we will have complexity of O(n2).

Let us demonstrate via an algorithm.

Example: anxn+an-1xn-1+an-2xn-2+…+a1x+a0 is a polynomial of degree n.

```
pow(x, n)
{
          p := 1;

          //loop from 1 to n
          for i:=1 to n
                    p := p*x;

          return p;
}

polynomial(A, x, n)
{
          int p, v:=0;
          for i := 0 to n

                    //loop within a loop from 0 to n
                    v := v + A[i]*pow(x, i);
          return v;
}
```

Loop within a loop => complexity = O(n2);

Now to find an optimal algorithm we need to find the lower bound here (as per lower bound theory). As per Lower Bound Theory, The optimal algorithm to solve the above problem is the one having complexity O(n). Lets prove this theorem using lower bounds.

Theorem: To prove that optimal algo of solving a n degree polynomial is O(n)
Proof: The best solution for reducing the algo is to make this problem less complex by dividing the polynomial into several straight line problems.

=> anxn+an-1xn-1+an-2xn-2+...+a1x+a0
can be written as,
((..(anx+an-1)x+..+a2)x+a1)x+a0
Now, algorithm will be as,

```
v=0
v=v+an
v=v*x
v=v+an-1
v=v*x
...
v=v+a1
v=v*x
v=v+a0
polynomial(A, x, n)
```

```
{
int p, v=0;

// loop executed n times
for i = n to 0
            v = (v + A[i])*x;

      return v;
}
```

Clearly, the complexity of this code is O(n). This way of solving such equations is called Horner's method. Here is were lower bound theory works and give the optimum algorithm's complexity as O(n).

2. Upper Bound Theory:
According to the upper bound theory, for an upper bound U(n) of an algorithm, we can always solve the problem in at most U(n) time.Time taken by a known algorithm to solve a problem with worse case input gives us the upper bound.

10. Describe primitive data structure?

A particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it.

Primitive Data Structure

- Primitive data structures are basic structures and are directly operated upon by machine instructions.
- Primitive data structures have different representations on different computers.
- Integers, floats, character and pointers are examples of primitive data structures.
- These data types are available in most programming languages as built in type.
- Integer: It is a data type which allows all values without fraction part. We can use it for whole numbers.
- Float: It is a data type which use for storing fractional numbers.
- Character: It is a data type which is used for character values.
- Pointer: A variable that holds memory address of another variable are called pointer.

11. Describe non-primitive data structure?

Non primitive Data Type

- These are more sophisticated data structures.
- These are derived from primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items.
- Examples of Non-primitive data type are Array, List, and File etc.
- A Non-primitive data type is further divided into Linear and Non-Linear data structure
- Array: An array is a fixed-size sequenced collection of elements of the same data type.
- List: An ordered set containing variable number of elements is called as Lists.
- File: A file is a collection of logically related information. It can be viewed as a large list of records consisting of various fields.

Linear data structures

- A data structure is said to be Linear, if its elements are connected in linear fashion by means of logically or in sequence memory locations.
- There are two ways to represent a linear data structure in memory,
- Static memory allocation
- Dynamic memory allocation
- The possible operations on the linear data structure are: Traversal, Insertion, Deletion, Searching, Sorting and Merging.
- Examples of Linear Data Structure are Stack and Queue.

- Stack: Stack is a data structure in which insertion and deletion operations are performed at one end only.
- The insertion operation is referred to as 'PUSH' and deletion operation is referred to as 'POP' operation.
- Stack is also called as Last in First out (LIFO) data structure.
- Queue: The data structure which permits the insertion at one end and Deletion at another end, known as Queue.
- End at which deletion is occurs is known as FRONT end and another end at which insertion occurs is known as REAR end.
- Queue is also called as First in First out (FIFO) data structure.

Nonlinear data structures
Nonlinear data structures are those data structure in which data items are not arranged in a sequence. Examples of Non-linear Data Structure are Tree and Graph.

Tree: A tree can be defined as finite set of data items (nodes) in which data items are arranged in branches and sub branches according to requirement. Trees represent the hierarchical relationship between various elements. Tree consist of nodes connected by edge, the node represented by circle and edge lives connecting to circle.

Graph: Graph is a collection of nodes (Information) and connecting edges (Logical relation) between nodes.

o A tree can be viewed as restricted graph.
o Graphs have many types:
  - Un-directed Graph
  - Directed Graph
  - Mixed Graph
  - Multi Graph
  - Simple GraphNull Graph
  - Weighted Graph
  -

12. Write a program to sum all elements of array?

```
#include <stdio.h>
int main()
{
   int a[1000],i,n,sum=0;

   printf("Enter size of the array : ");
   scanf("%d",&n);

   printf("Enter elements in array : ");
   for(i=0; i<n; i++)
   {
      scanf("%d",&a[i]);
   }


   for(i=0; i<n; i++)
   {

      sum+=a[i];
   }
    printf("sum of array is : %d",sum);
    return 0;
}
```

13. Write a program to obtain transpose of a matrix?

```
#include <stdio.h>
int main() {
    int a[10][10], transpose[10][10], r, c, i, j;
    printf("Enter rows and columns: ");
    scanf("%d %d", &r, &c);

    // Assigning elements to the matrix
    printf("\nEnter matrix elements:\n");
    for (i = 0; i < r; ++i)
        for (j = 0; j < c; ++j) {
            printf("Enter element a%d%d: ", i + 1, j + 1);
            scanf("%d", &a[i][j]);
        }

    // Displaying the matrix a[][]
    printf("\nEntered matrix: \n");
    for (i = 0; i < r; ++i)
        for (j = 0; j < c; ++j) {
            printf("%d  ", a[i][j]);
            if (j == c - 1)
                printf("\n");
        }

    // Finding the transpose of matrix a
    for (i = 0; i < r; ++i)
        for (j = 0; j < c; ++j) {
            transpose[j][i] = a[i][j];
        }

    // Displaying the transpose of matrix a
    printf("\nTranspose of the matrix:\n");
    for (i = 0; i < c; ++i)
        for (j = 0; j < r; ++j) {
            printf("%d  ", transpose[i][j]);
            if (j == r - 1)
                printf("\n");
        }
    return 0;
}
```

14. Write a program that will convert 2D matrix representation to Sparse representation?

```
#include <stdio.h>
#define MAX 20

void read_matrix(int a[10][10], int row, int column);
void print_sparse(int b[MAX][3]);
void create_sparse(int a[10][10], int row, int column, int b[MAX][3]);
```

```
int main()
{
   int a[10][10], b[MAX][3], row, column;
   printf("\nEnter the size of matrix (rows, columns): ");
   scanf("%d%d", &row, &column);

   read_matrix(a, row, column);
   create_sparse(a, row, column, b);
   print_sparse(b);
   return 0;
}

void read_matrix(int a[10][10], int row, int column)
{
   int i, j;
   printf("\nEnter elements of matrix\n");
   for (i = 0; i < row; i++)
   {
      for (j = 0; j < column; j++)
      {
         printf("[%d][%d]: ", i, j);
         scanf("%d", &a[i][j]);
      }
   }
}

void create_sparse(int a[10][10], int row, int column, int b[MAX][3])
{
   int i, j, k;
   k = 1;
   b[0][0] = row;
   b[0][1] = column;
   for (i = 0; i < row; i++)
   {
      for (j = 0; j < column; j++)
      {
         if (a[i][j] != 0)
         {
            b[k][0] = i;
            b[k][1] = j;
            b[k][2] = a[i][j];
            k++;
         }
      }
      b[0][2] = k - 1;
   }
}

void print_sparse(int b[MAX][3])
{
   int i, column;
   column = b[0][2];
   printf("\nSparse form - list of 3 triples\n\n");
```

```
        for (i = 0; i <= column; i++)
        {
            printf("%d\t%d\t%d\n", b[i][0], b[i][1], b[i][2]);
        }
    }
```

15. Write a program for matrix multiplication?

```
#include<stdio.h>
#include<stdlib.h>
int main(){
int a[10][10],b[10][10],mul[10][10],r,c,i,j,k;
system("cls");
printf("enter the number of row=");
scanf("%d",&r);
printf("enter the number of column=");
scanf("%d",&c);
printf("enter the first matrix element=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
scanf("%d",&a[i][j]);
}
}
printf("enter the second matrix element=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
scanf("%d",&b[i][j]);
}
}

printf("multiply of the matrix=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
mul[i][j]=0;
for(k=0;k<c;k++)
{
mul[i][j]+=a[i][k]*b[k][j];
}
}
}
//for printing result
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
printf("%d\t",mul[i][j]);
}
printf("\n");
}
return 0;
}
```

16. Explain insertion and deletion operations in array?

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data at the end of the array −

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>

main() {
   int LA[] = {1,3,5,7,8};
   int item = 10, k = 3, n = 5;
   int i = 0, j = n;

   printf("The original array elements are :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }

   n = n + 1;

   while( j >= k) {
      LA[j+1] = LA[j];
      j = j - 1;
   }

   LA[k] = item;

   printf("The array elements after insertion :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after insertion :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 10
LA[4] = 7
```

```
LA[5] = 8
```

For other variations of array insertion operation click here

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to delete an element available at the Kth position of LA.

```
1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

Example

Following is the implementation of the above algorithm –

```
#include <stdio.h>

void main() {
   int LA[] = {1,3,5,7,8};
   int k = 3, n = 5;
   int i, j;

   printf("The original array elements are :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }

   j = k;

   while( j < n) {
      LA[j-1] = LA[j];
      j = j + 1;
   }

   n = n -1;

   printf("The array elements after deletion :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When we compile and execute the above program, it produces the following result –

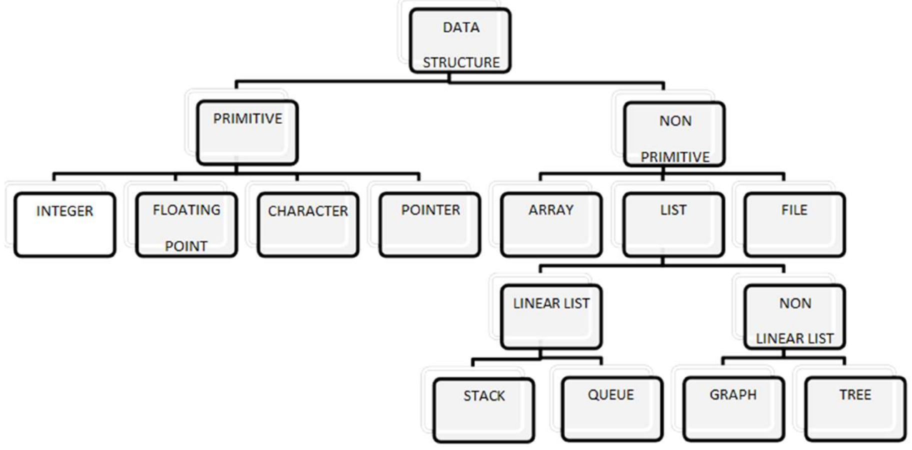
Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after deletion :
```

LA[0] = 1
LA[1] = 3
LA[2] = 7
LA[3] = 8

<u>10 marks:</u>

1. **Explain classification of datastructures with a neat diagram?**



Data Structures are normally classified into two broad categories

1. Primitive Data Structure

2. Non-primitive data Structure

Data types
A particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it.
A particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it.
Primitive Data Structure

- Primitive data structures are basic structures and are directly operated upon by machine instructions.
- Primitive data structures have different representations on different computers.
- Integers, floats, character and pointers are examples of primitive data structures.
- These data types are available in most programming languages as built in type.
- Integer: It is a data type which allows all values without fraction part. We can use it for whole numbers.
- Float: It is a data type which use for storing fractional numbers.
- Character: It is a data type which is used for character values.
- Pointer: A variable that holds memory address of another variable are called pointer.

Non primitive Data Type

- These are more sophisticated data structures.
- These are derived from primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items.
- Examples of Non-primitive data type are Array, List, and File etc.
- A Non-primitive data type is further divided into Linear and Non-Linear data structure
- Array: An array is a fixed-size sequenced collection of elements of the same data type.
- List: An ordered set containing variable number of elements is called as Lists.

- File: A file is a collection of logically related information. It can be viewed as a large list of records consisting of various fields.

Linear data structures

- A data structure is said to be Linear, if its elements are connected in linear fashion by means of logically or in sequence memory locations.
- There are two ways to represent a linear data structure in memory,
- Static memory allocation
- Dynamic memory allocation
- The possible operations on the linear data structure are: Traversal, Insertion, Deletion, Searching, Sorting and Merging.
- Examples of Linear Data Structure are Stack and Queue.

• Stack: Stack is a data structure in which insertion and deletion operations are performed at one end only.

- The insertion operation is referred to as 'PUSH' and deletion operation is referred to as 'POP' operation.
- Stack is also called as Last in First out (LIFO) data structure.
- Queue: The data structure which permits the insertion at one end and Deletion at another end, known as Queue.
- End at which deletion is occurs is known as FRONT end and another end at which insertion occurs is known as REAR end.
- Queue is also called as First in First out (FIFO) data structure.

Nonlinear data structures

Nonlinear data structures are those data structure in which data items are not arranged in a sequence. Examples of Non-linear Data Structure are Tree and Graph.

Tree: A tree can be defined as finite set of data items (nodes) in which data items are arranged in branches and sub branches according to requirement. Trees represent the hierarchical relationship between various elements. Tree consist of nodes connected by edge, the node represented by circle and edge lives connecting to circle.

Graph: Graph is a collection of nodes (Information) and connecting edges (Logical relation) between nodes.

o A tree can be viewed as restricted graph.

o Graphs have many types:
  - Un-directed Graph
  - Directed Graph
  - Mixed Graph
  - Multi Graph
  - Simple Graph Null Graph
  - Weighted Graph

2. Explain array and two dimensional array with diagrams?

- An array is a collection of elements of the same type that are referenced by a common name.
- Compared to the basic data type (int, float & char) it is an aggregate or derived data type.
- All the elements of an array occupy a set of contiguous memory locations.

Why need to use array type?Consider the following issue:

We have a list of 1000 students' marks of an integer type. If using the basic data type (int), we will declare something like the following…"

int studMark0, studMark1, studMark2, ..., studMark999;

Can you imagine how long we have to write the declaration part by using normal variable declaration?

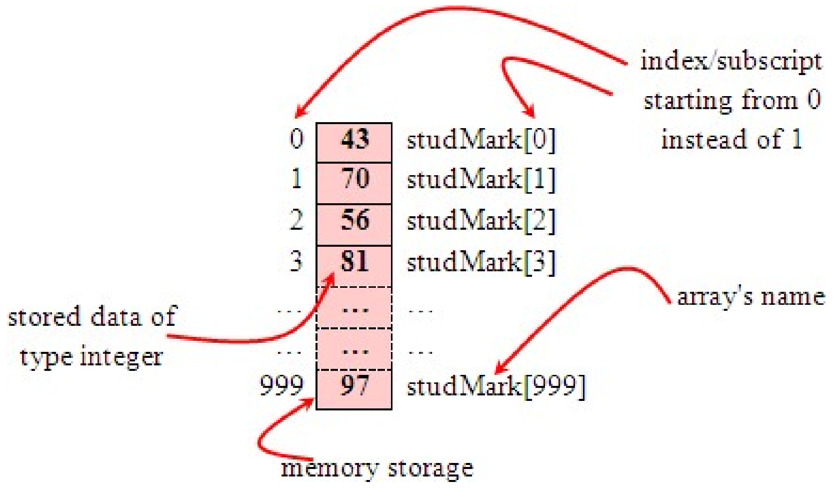
int main(void)

```
{
int studMark1, studMark2, studMark3, studMark4, …, …, studMark998, stuMark999, studMark1000;
…
…
return 0;
}
```

By using an array, we just declare like this,

int studMark[1000];

This will reserve 1000 contiguous memory locations for storing the students' marks. Graphically, this can be depicted as in the following figure.



This absolutely has simplified our declaration of the variables. We can use index or subscript to identify each element or location in the memory. Hence, if we have an index of jIndex, studMark[jIndex] would refer to the jIndexth element in the array of studMark. For example, studMark[0] will refer to the first element of the array. Thus by changing the value of jIndex, we could refer to any element in the array. So, array has simplified our declaration and of course, manipulation of the data.

One Dimensional Array: Declaration

Dimension refers to the array's size, which is how big the array is. A single or one dimensional array declaration has the following form,

array_element_data_type array_name[array_size];

Here, array_element_data_type define the base type of the array, which is the type of each element in the array. array_name is any valid C / C++ identifier name that obeys the same rule for the identifier naming. array_size defines how many elements the array will hold.

For example, to declare an array of 30 characters, that construct a people name, we could declare,

char cName[30];

Which can be depicted as follows,

In this statement, the array character can store up to 30 characters with the first character occupying location cName[0] and the last character occupying cName[29]. Note that the index runs from 0 to 29. In C, an index always starts from 0 and ends with array's (size-1). So, take note the difference between the array size and subscript/index terms.

Examples of the one-dimensional array declarations,

```
int      xNum[20], yNum[50];
float    fPrice[10], fYield;
char     chLetter[70];
```

The first example declares two arrays named xNum and yNum of type int. Array xNum can store up to 20 integer numbers while yNum can store up to 50 numbers. The second line declares the array fPrice of type float. It can store up to 10 floating-point values. fYield is basic variable which shows array type can be declared together with basic type provided the type is similar. The third line declares the array chLetter of type char. It can store a string up to 69 characters. Why 69 instead of 70? Remember, a string has a null terminating character (\0) at the end, so we must reserve for it.

Array Initialization

- An array may be initialized at the time of declaration.
- Giving initial values to an array.
- Initialization of an array may take the following form,
- type array_name[size] = {a_list_of_value};
- For example:
- int idNum[7] = {1, 2, 3, 4, 5, 6, 7};
- float fFloatNum[5] = {5.6, 5.7, 5.8, 5.9, 6.1};
- char chVowel[6] = {'a', 'e', 'i', 'o', 'u', '\0'};
- The first line declares an integer array idNum and it immediately assigns the values 1, 2, 3, ..., 7 to idNum[0], idNum[1], idNum[2],..., idNum[6] respectively.
- The second line assigns the values 5.6 to fFloatNum[0], 5.7 to fFloatNum[1], and so on.
- Similarly the third line assigns the characters 'a' to chVowel[0], 'e' to chVowel[1], and so on. Note again, for characters we must use the single apostrophe/quote (') to enclose them.
- Also, the last character in chVowel is NULL character ('\0').

Initialization of an array of type char for holding strings may take the following form,
char array_name[size] = "string_lateral_constant";

□ For example, the array chVowel in the previous example could have been written more compactly as follows,
char chVowel[6] = "aeiou";

□ When the value assigned to a character array is a string (which must be enclosed in double quotes), the compiler automatically supplies the NULL character but we still have to reserve one extra place for the NULL.

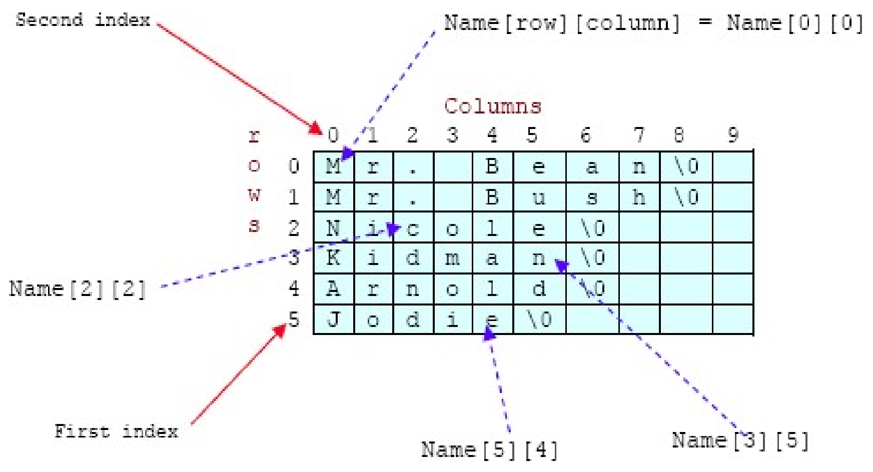
□ For unsized array (variable sized), we can declare as follow,
char chName[ ] = "Mr. Dracula";

□ C compiler automatically creates an array which is big enough to hold all the initializer.

Two Dimensional Array

A two dimensional array has two subscripts/indexes. The first subscript refers to the row, and the second, to the column. Its declaration has the following form,
data_type array_name[1st dimension size][2nd dimension size];
For examples,
int xInteger[3][4];
float matrixNum[20][25];
The first line declares xInteger as an integer array with 3 rows and 4 columns.
Second line declares a matrixNum as a floating-point array with 20 rows and 25 columns.

If we assign initial string values for the 2D array it will look something like the following,
char Name[6][10] = {"Mr. Bean", "Mr. Bush", "Nicole", "Kidman", "Arnold", "Jodie"};
Here, we can initialize the array with 6 strings, each with maximum 9 characters long. If depicted in rows and columns it will look something like the following and can be considered as contiguous arrangement in the memory.



	0	1	2	3	4	5	6	7	8	9
0	M	r	.		B	e	a	n	\0	
1	M	r	.		B	u	s	h	\0	
2	N	i	c	o	l	e	\0			
3	K	i	d	m	a	n	\0			
4	A	r	n	o	l	d	\0			
5	J	o	d	i	e	\0				

- Take note that for strings the null character (\0) still needed.
- From the shaded square area of the figure we can determine the size of the array.
- For an array Name[6][10], the array size is 6 x 10 = 60 and equal to the number of the colored square. In general, for
- array_name[x][y];
- The array size is = First index x second index = xy.
- This also true for other array dimension, for example three dimensional array,
- array_name[x][y][z]; => First index x second index x third index = xyz
- For example,
- ThreeDimArray[2][4][7] = 2 x 4 x 7 = 56.
- And if you want to illustrate the 3D array, it could be a cube with wide, long and height dimensions.

3. **Explain the various operation on array with sample programs?**

Following are the basic operations supported by an array.

- Traverse − print all the array elements one by one.
- Insertion − Adds an element at the given index.
- Deletion − Deletes an element at the given index.
- Search − Searches an element using the given index or by the value.
- Update − Updates an element at the given index.

Traverse Operation

This operation is to traverse through the elements of an array.

Example

Following program traverses and prints the elements of an array:

```
#include <stdio.h>
main() {
  int LA[] = {1,3,5,7,8};
  int item = 10, k = 3, n = 5;
  int i = 0, j = n;
  printf("The original array elements are :\n");
  for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
  }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
```

```
LA[3] = 7
LA[4] = 8
```

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data at the end of the array −

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>

main() {
   int LA[] = {1,3,5,7,8};
   int item = 10, k = 3, n = 5;
   int i = 0, j = n;

   printf("The original array elements are :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }

   n = n + 1;

   while( j >= k) {
      LA[j+1] = LA[j];
      j = j - 1;
   }

   LA[k] = item;

   printf("The array elements after insertion :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after insertion :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 10
LA[4] = 7
LA[5] = 8
```

For other variations of array insertion operation click here

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to delete an element available at the Kth position of LA.

```
1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>

void main() {
   int LA[] = {1,3,5,7,8};
   int k = 3, n = 5;
   int i, j;

   printf("The original array elements are :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }

   j = k;

   while( j < n) {
      LA[j-1] = LA[j];
      j = j + 1;
   }

   n = n -1;

   printf("The array elements after deletion :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after deletion :
LA[0] = 1
LA[1] = 3
LA[2] = 7
```

LA[3] = 8

Search Operation

You can perform a search for an array element based on its value or its index.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Example

Following is the implementation of the above algorithm −

```c
#include <stdio.h>

void main() {
   int LA[] = {1,3,5,7,8};
   int item = 5, n = 5;
   int i = 0, j = 0;

   printf("The original array elements are :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }

   while( j < n){
      if( LA[j] == item ) {
         break;
      }

      j = j + 1;
   }

   printf("Found element %d at position %d\n", item, j+1);
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
Found element 5 at position 3
```

Update Operation

Update operation refers to updating an existing element from the array at a given index.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to update an element available at the Kth position of LA.

```
1. Start
```

```
2. Set LA[K-1] = ITEM
3. Stop
```

Example

Following is the implementation of the above algorithm −

```c
#include <stdio.h>

void main() {
  int LA[] = {1,3,5,7,8};
  int k = 3, n = 5, item = 10;
  int i, j;

  printf("The original array elements are :\n");

  for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
  }

  LA[k-1] = item;

  printf("The array elements after updation :\n");

  for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
  }
}
```

When we compile and execute the above program, it produces the following result −

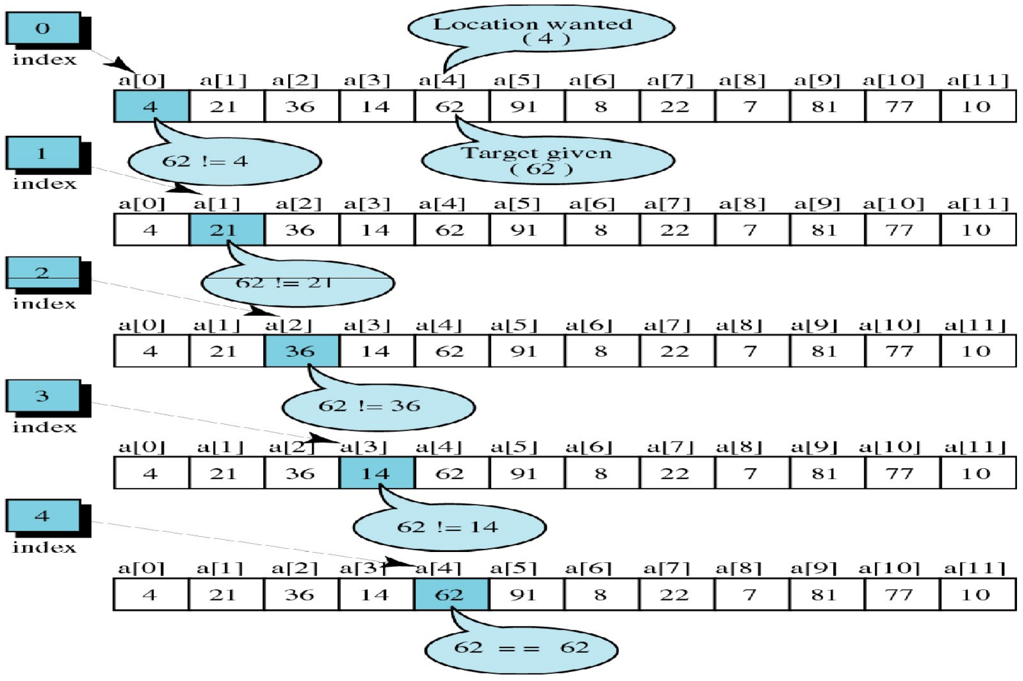
Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after updation :
LA[0] = 1
LA[1] = 3
LA[2] = 10
LA[3] = 7
LA[4] = 8
```

4. **Explain linear search and binary search?**

Linear Search

Linear search in C is to find whether a number is present in an array. If it's present, then at what location it occurs. It is also known as a sequential search. It is straightforward and works as follows: we compare each element with the element to search until we find it or the list ends.

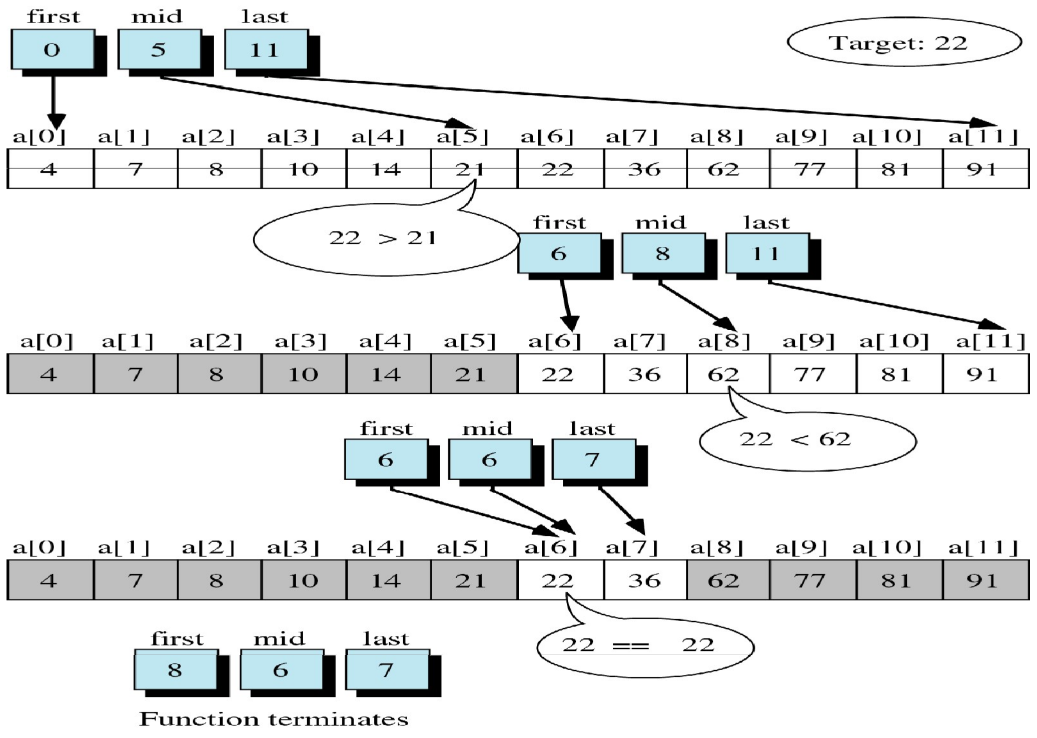


Above diagram shows the simple example to find the 62 in the given array using linear search.
Program:

```
    #include <stdio.h>
    int main()
    {
      int array[100], search, c, n;
      printf("Enter number of elements in array\n");
      scanf("%d", &n);
      printf("Enter %d integer(s)\n", n);
      for (c = 0; c < n; c++)
      scanf("%d", &array[c]);
      printf("Enter a number to search\n");
      scanf("%d", &search);
      for (c = 0; c < n; c++)
      {
       if (array[c] == search)    /* If required element is found */
      {
       printf("%d is present at location %d.\n", search, c+1);
      break;
      }
     }
  if (c == n)
   printf("%d isn't present in the array.\n", search);
 return 0;
}
```

Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.
The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).
We basically ignore half of the elements just after one comparison.
1. Compare x with the middle element.

2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.
5. Search an ordered array of integers for a value and return its index if the value is found. Otherwise, return -1.



The above example shows the steps to find the element 62 in a given array using binary search.
Binary search is based on the "divide-and-conquer" strategy which works as follows:
n Start by looking at the middle element of the array
– 1. If the value it holds is lower than the search element, eliminate the first half of the array from further consideration.
– 2. If the value it holds is higher than the search element, eliminate the second half of the array from further consideration.
Repeat this process until the element is found, or until the entire array has been eliminated
Algorithm:

```
Set first and last boundary of array to be searched
Repeat the following:
     Find middle element between first and last boundaries;
     if (middle element contains the search value)
    return middle_element position;
     else if (first >= last )
    return –1;
     else if (value < the value of middle_element)
            set last to middle_element position – 1;
     else
    set first to middle_element position + 1;
```

Program : Binary Search

```
#include <stdio.h>
int main()
{
  int i, first, last, middle, n, search, array[100];
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (i = 0; i < n; i++)
              scanf("%d", &array[i]);
       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;
 }
```

5. Explain complexity analysis (time and space) and best, average and worst case analysis?

□ Time complexity

□ Time Complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.

□ "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.

□ Space complexity

□ Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.

□ We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this.

□ We can use bytes, but it's easier to use, say, number of integers used, number of fixed-sized structures, etc. In the end, the function we come up with will be independent of the actual number of bytes needed to represent the unit.

□ Space complexity is sometimes ignored because the space used is minimal and/or obvious, but sometimes it becomes as important an issue as time.

Worst Case Analysis

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched is not present in the array. When x is not present, the search () functions compares it with all the elements of array [] one by one. Therefore, the worst case time complexity of linear search would be.

Average Case Analysis

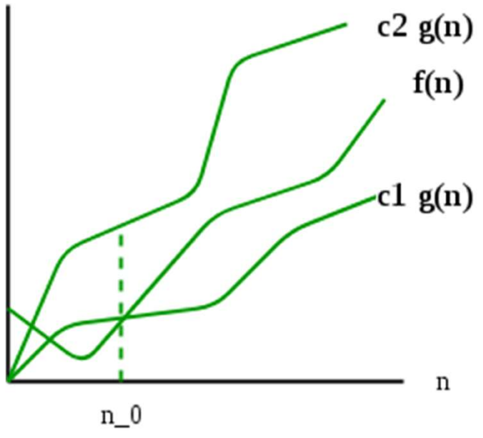
In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed. So we sum all the cases and divide the sum by (n+1).

Best Case Analysis

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in worst case is constant (not dependent on n). So time complexity in the best case would be.

6. Write about asymptotic notations and their properties?

Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.



f(n) = theta(g(n))

1) Θ Notation: The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.

A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression.

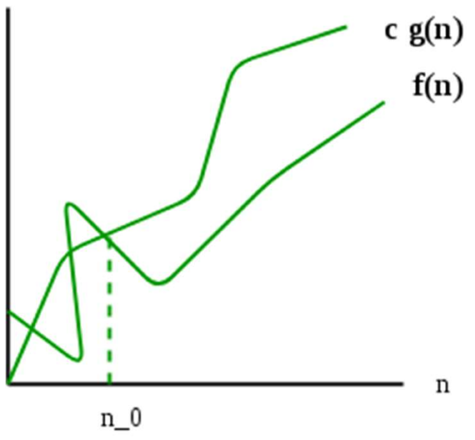
3n3 + 6n2 + 6000 = Θ(n3)

Dropping lower order terms is always fine because there will always be a n0 after which Θ(n3) has higher values than Θn2) irrespective of the constants involved.

For a given function g(n), we denote Θ(g(n)) is following set of functions.

Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such
that 0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}

The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.

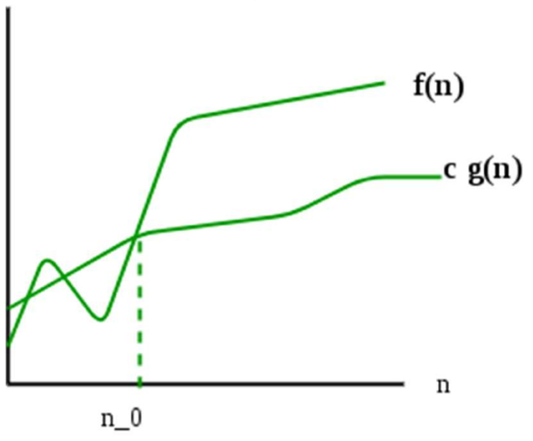


f(n) = O(g(n))

2) Big O Notation: The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.
If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:
1. The worst case time complexity of Insertion Sort is Θ(n^2).
2. The best case time complexity of Insertion Sort is Θ(n).

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

O(g(n)) = { f(n): there exist positive constants c and
n0 such that 0 <= f(n) <= c*g(n) for
all n >= n0}



f(n) = Omega(g(n))

3) Ω Notation: Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Ω Notation can be useful when we have lower bound on time complexity of an algorithm. As discussed in the previous post, the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.

For a given function g(n), we denote by Ω(g(n)) the set of functions.

Ω (g(n)) = {f(n): there exist positive constants c and
n0 such that 0 <= c*g(n) <= f(n) for
all n >= n0}.

Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as $\Omega(n)$, but it is not a very useful information about insertion sort, as we are generally interested in worst case and sometimes in average case.

Properties of Asymptotic Notations :

As we have gone through the definition of this three notations let's now discuss some important properties of those notations.

General Properties :

If f(n) is O(g(n)) then a*f(n) is also O(g(n)) ; where a is a constant.

Example: $f(n) = 2n^2+5$ is $O(n^2)$
then $7*f(n) = 7(2n^2+5)$
$= 14n^2+35$ is also $O(n^2)$

Similarly this property satisfies for both Θ and Ω notation.
We can say
If f(n) is Θ(g(n)) then a*f(n) is also Θ(g(n)) ; where a is a constant.
If f(n) is Ω (g(n)) then a*f(n) is also Ω (g(n)) ; where a is a constant.

Reflexive Properties :

If f(n) is given then f(n) is O(f(n)).

Example: $f(n) = n^2$; $O(n^2)$ i.e O(f(n))

Similarly this property satisfies for both Θ and Ω notation.
We can say
If f(n) is given then f(n) is Θ(f(n)).
If f(n) is given then f(n) is Ω (f(n)).

Transitive Properties :

If f(n) is O(g(n)) and g(n) is O(h(n)) then f(n) = O(h(n)) .

Example: if f(n) = n , $g(n) = n^2$ and $h(n)=n^3$
n is $O(n^2)$ and n^2 is $O(n^3)$
then n is $O(n^3)$

Similarly this property satisfies for both Θ and Ω notation.
We can say
If f(n) is Θ(g(n)) and g(n) is Θ(h(n)) then f(n) = Θ(h(n)) .
If f(n) is Ω (g(n)) and g(n) is Ω (h(n)) then f(n) = Ω (h(n))

Symmetric Properties :

If f(n) is Θ(g(n)) then g(n) is Θ(f(n)) .

Example: $f(n) = n^2$ and $g(n) = n^2$
then $f(n) = \Theta(n^2)$ and $g(n) = \Theta(n^2)$

This property only satisfies for Θ notation.

Transpose Symmetric Properties :
If $f(n)$ is $O(g(n))$ then $g(n)$ is $\Omega(f(n))$.

Example: $f(n) = n$, $g(n) = n^2$
then n is $O(n^2)$ and n^2 is $\Omega(n)$

This property only satisfies for O and Ω notations.

Some More Properties :
If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ then $f(n) = \Theta(g(n))$
If $f(n) = O(g(n))$ and $d(n)=O(e(n))$
then $f(n) + d(n) = O(\max(g(n), e(n)))$
Example: $f(n) = n$ i.e $O(n)$
$d(n) = n^2$ i.e $O(n^2)$
then $f(n) + d(n) = n + n^2$ i.e $O(n^2)$
If $f(n)=O(g(n))$ and $d(n)=O(e(n))$
then $f(n) * d(n) = O(g(n) * e(n))$
Example: $f(n) = n$ i.e $O(n)$
$d(n) = n^2$ i.e $O(n^2)$
then $f(n) * d(n) = n * n^2 = n^3$ i.e $O(n^3)$