Unit-I Topic-I Algorithm, ADT Topic-II Array

Algorithm:

"Algorithm is step by step description of how to solve any problem".
Thus, it is logical flow (Solution) of problem.
All algorithms must satisfy the following criteria: **(Characteristics of Algorithm)**

1) Each and every instruction should be precised (Accurate) and unambiguous (Unmistakable).
2) One or more instructions should not be repeated.
3) The desired result must be obtained.
4) Representation of algorithm is in very simple English language therefore it is easy to understand.
5) It gives the language independent layout of the program.

Analysis of Problem (Program):

Basically there are two criteria used to check the performance of problem or program viz:

1) Time Complexity.
2) Space Complexity.

Let us see in details:

1) Time Complexity:

The amount of time taken by the program for execution is the "time complexity" of that program.
The total number of times a statement gets executed is called as its "Frequency count".
Calculation of exact time taken by program for execution may be difficult. However a rough estimate can be made. To do this, we have to identify active operations (Which are central in program and execute often) in the program. The other operations like assignments, manipulation of loop etc. are called book keeping operations and will not execute as many times as the active operations. Thus, they need not be computed.
Therefore, we can say that execution time is proportional to the frequency count of active operations.
Example.

1)
```
for( i = 1 ; i <= 10 ; i + +)
{
    a = a + 1;
}
```

In above example, the statement 'a = a + 1' executes 10 times therefore its frequency count is 10.

2)
```
for( i = 1 ; i <= m ; i + +)
{
    for( j = 1; j <= n ; j+ +)
    {
            a = a*a ;
    }
}
```

In above example, the statement "a = a*a" executes m*n times i.e. for each value 'i', 'j' varies from 1 to n.
If m = n then frequency count is n^2. Thus, its time complexity is n^2.

2) Space Complexity:

The amount of memory space taken by the program for execution is the "space complexity" of that program
The space complexity can be computed by considering data and their memory sizes. Again, those items with maximum storage demands are of mostly considerable in space complexity.

- Note that: The program with less time complexity & less space complexity is the best among all other programs.

Big O notation: (Big-oh notation)

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

- Big O notation characterizes functions according to their growth rates.
- Different functions with the same growth rate may be represented using the same O notation.
- Generally, Big O notation is used to express the time and space complexity of program.

Time complexity by using Big O notation:

We can express the time complexity of program in terms of the order of magnitude of frequency count using the "Big O" notation.

Example:

If $f(n) = n^4+20n^3+n+50$ then the term with highest order of magnitude i.e. n^4 determines the time complexity of the function f(n) which is noted as $O(n^4)$

Definition of Big-O notation:

$f(n) = O(g(n))$ [read as f of n equals big-Oh of g of n] is true if there exist two constants c and n_0, such that

$|f(n)| <= c\ |g(n)|$ for all $n >= n_0$.

Different Problem solving techniques (Algorithms):

1) Divide and Conquer Algorithm:

Concept: A divide and conquer algorithm works by breaking down a main problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. Then solutions of sub-problems are combined to give a solution to the main problem.

Application of Divide and conquer algorithm:

This algorithm is more efficient for all kinds of problems, such as

- Sorting (e.g. quick sort, merge sort)
- Multiplying large numbers
- Syntactic analysis (e.g., top-down parsers)
- Computing the discrete Fourier transform.

2) Greedy Algorithm:

Concept: Consider there is wide variety of some problems. Most of these problems have 'n' inputs and requires obtaining a subset that satisfies some *constraint (Rule).*

- Any subset that satisfies these constraints is called as *"Feasible solution" or "Candidate solution"*.
- A feasible solution that either maximizes or minimizes a given objective of function is called as *"Optimal Solution"*. Thus, *optimal solution* is nothing but *best solutions among all other solutions*. And Greedy algorithm always thinks about optimal solution.
- Greedy algorithms mostly (but not always) fail to find the globally optimal solution, because they usually do not operate fully on all the data.

Objective or aim of Greedy algorithm:

- An optimization solution is one in which we want to find, not just *a* solution, but the *best* solution
- A "greedy algorithm" sometimes works well for optimization problems
- A greedy algorithm works in phases. At each phase:
  - It take the best among all phases, without considering future consequences

Application of Greedy algorithm:

This algorithm is more efficient for following problems,

- It is used in Graph coloring algorithm
- It is used in Kruskal's algorithm and Prim's algorithm.
- It is used in Dijkstra's algorithm for finding single-source shortest path.

Example: Suppose, you done shopping in a cloth shop. And your total bill is Rs. 1923/-.
And you give Rs. 2000/- for cashier. Then cashier think about change to give you. He has to give you Rs. 77/-.Also, cashier has some 50 rupee, some 20 rupee, some 10 rupee, some 5 rupee coins, some 2 rupee coins and some 1 rupee coins.
He returns Rs. 77/- to you in such a way that less number of coins or less number of rupees that makes total Rs. 77/-. For that he gives you one 50 rupee, one 20 rupee, one 5 rupee and one 2 rupee coins.

This is optimal solution for cashier for making change; such optimal solution is done by Greedy algorithm.

3) Dynamic Programming:

Concept: In computer science, dynamic programming is a method for solving complex problems by breaking them down into simpler sub problems. It is applicable to problems showing the properties of overlapping sub problems which are only slightly smaller and optimal substructure and store the result of each sub problems in such a way that that it will be used in future.

The key idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (sub problems), and then combine the solutions of the sub problems to reach an overall solution. Often, many of these sub problems are really the same.

- The dynamic programming is used to solve each sub problem only once, thus reducing the number of computations. Once the solution to a given sub problem has been computed, it is stored. And the next time the same solution is needed, it is simply looked up in stored data. This approach is especially useful when the number of repeating sub problems grows exponentially as a function of the size of the input.

Application of Dynamic programming:

This algorithm is more efficient for following problems,

- It is used in Viterbi algorithm (used for hidden Markov models)
- It is used in Early algorithm (a type of chart parser)
- It is used in Floyd's all-pairs shortest path algorithm
- It is used in dynamic time warping algorithm for computing the global distance between two time series
- It is used in Knuth's word wrapping algorithm

4) Branch and Bound:

The Branch and Bound method was first proposed by A. H. Land and A. G. Doig in 1960 for discrete programming.

Concept:

- Branch and bound (BB or B&B) is a general algorithm for finding optimal solutions of various problems, especially in discrete and combinatorial optimization.
- A branch-and-bound algorithm consists of a systematic enumeration of all *candidate solutions* or *feasible solutions*, where large subsets of useless feasible solutions are discarded, by using upper and lower estimated bounds.

Application of Branch and Bound:

This approach is used for a number of NP-hard problems, such as

- Knapsack problem
- Integer programming
- Nonlinear programming
- Traveling salesman problem (TSP)
- Quadratic assignment problem (QAP)
- Maximum satisfies ability problem (MAX-SAT)
- Nearest neighbor search (NNS)
- Cutting stock problem

4) Backtracking:

Concept:

Backtracking is a way of trying out various sequences of decisions, until you find "Correct" decision.

Suppose you have to make a series of decisions, among various choices, where

- You don't have enough information to know what to choose?
- Each decision leads to a new set of choices
- Some sequence of choices (possibly more than one) may be a solution to your problem!

Example:

1) Given a maze, find a path from start to finish

- At each intersection, you have to decide between three or fewer choices:
  - Go straight
  - Go left
  - Go right

- You don't have enough information to choose correctly
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution types of maze problem can be solved with backtracking

Applications of Backtracking Algorithm:

Backtracking is useful in Puzzles such as eight queens' puzzle, crosswords, verbal arithmetic, Sudoku, Peg Solitaire.

****

ADT(Abstract Data Type)

"ADT is a general tool which is used to define the data structure."

To define ADT of any data structure, following points are considered:

I) Data Object
II) Operation performed on to data structure
III) Rules to define operation

If we explain any data structure by using above three points then that explanation becomes ADT.

E.g.:

1) ADT for array:

We can define ADT for array by using following three points:

I) Data Object:

- Array is a linear data structure which is collection of elements having same data type referred by common name.
- In case of array, all elements are stored in sequential manner also all elements are stored at individual index (Since, index is integer value that represents position of element in array) which is shown in following figure:

```
int     p[5]= {10,20,30,40,50} ;
```

index	**0**	**1**	**2**	**3**	**4**
Array→ p	10	20	30	40	50
Address	90	92	94	96	98

II) Operation performed on Array:

I) Insert Operation:

This operation is used to insert new element at appropriate position in an array.

E.g.:

```
void    insert(int  pos, int      elt);
```

In above declaration, insert ( ) function inserts element 'elt' at position 'pos' in an array.

II) Retrieve Operation:

This operation is used to retrieve the element from array from appropriate position.

E.g.:

```
int     retrieve(int   pos);
```

In above declaration, retrieve ( ) function retrieve one integer element from position 'pos' in an array.

III) Rules for Operation:

To perform both above declared functions, we must have to pass valid index value to them.

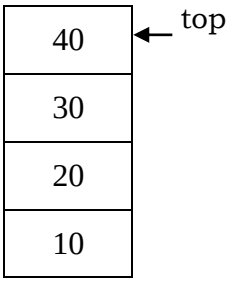
Thus, we explain array by using tree point's data object, operation performed on array and rules for operation and hence it becomes ADT for array.

2) ADT for STACK:

=> We can define ADT for stack by using following three points:

I) Data Object:

- Stack is a linear data structure which is collection of elements into which elements or items are inserted or removed from only one end called 'top'.
- Stack having only one end therefore it works in LIFO (Last In First Out) manner i.e last inserted element is come out at first attempt.
    - Following figure shows stack with elements:



Stack

II) Operation performed on Stack:

III) Push Operation:

This operation is used to insert new element into stack. Always, element is inserted at the 'top' of the stack.

E.g.:

```
void    push(stack   *p, int     x);
```

In above declaration, push ( ) function add or inserts element 'x' at into stack 'p'.

IV) Pop Operation:

This operation is used to remove the element from the stack. Always, topmost element is removed from stack.

E.g.:

```
int     pop(stack *p);
```

In above declaration, pop ( ) function removes the topmost element from stack 'p'.

III) Rules for Operation:

I) Rule for Push operation:

While performing push operation on to stack, first we have to check stack is full or not.
If stack is full then element cannot pushed into stack.

II) Rule for Pop Operation:

While performing pop operation on to stack, first we have to check stack is empty or not.
If stack is Empty then element cannot popped from stack.

Thus, we explain stack by using tree point's data object, operation performed on stack and rules for operation and hence it becomes ADT for stack.

Que. Explain ADT for queue. (Home work)

2. ARRAY

INTRODUCTION:

We know that the concept of variable is introduced **to store the data**. But single variable can store only one value at a time, this is the drawback of variable. To overcome the drawback of single variable the concept of Array is introduced. That is array is also single variable but it has an ability to store multiple (more than one) value at a time.

Definition:

"An Array is linear data structure which is collection of elements or items having **same data type** referred by common name"

OR

"An Array is linear data structure which is collection of homogeneous (having same type) elements or items referred by common name"

- In an array the individual element is accessed with the help of integer value (which is always integer and greater than 1) and is called **subscript or index.**
- Also all elements of array are stored in **continuous memory** allocation.

Declaration Syntax:

datatype array_name[size1][size2]……[sizeN];

here;

datatype = any valid datatype in C language i.e. int, char, float etc.
array_name = an identifier (name given by programmer)
size1, size2,……sizeN are **integer constants** and that denotes total number of elements stored in an array.

e.g. 1) int x[5];
here ;
'x' is array which can holds(stores) **five** integer values at a time.
Also, to store above entire (complete) array 'x', 10 bytes of memory is required. (Since to store one integer value, 2 bytes of memory is required)

2) float y[10];
here ;
'y' is array which can holds(stores) **ten** float values at a time.
Also, to store above entire (complete) array 'y', 40 bytes of memory is required. (Since to store one float value, 4 bytes of memory is required)

Types of Array:

Depending on number of subscripts used in array, array having three types:

1) One Dimensional array:

" The array having **only one subscript** is called as One dimensional array or vector"

Declaration Syntax:

datatype array_name[size];

here;
datatype = any valid datatype in C language i.e. int, char, float etc.
'size' is integer value that denotes total number of elements stored in array

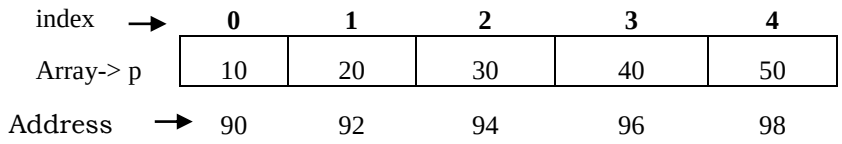
e.g.
1) char x[5];
here ;
'x' is array which can holds(stores) **five** characters at a time.
Also, to store above entire (complete) array 'x', 5 bytes of memory is required. (Since to store one character, 1 byte of memory is required)

Initialization of One dimensional array:

One dimensional array can be initialized as fallow;

e.g. int p[5] = { 10 , 20 , 30 , 40 , 50 } ;
here;

'p' is integer type array which stores five integers at a time. The storage of all elements in array 'p' is shown in following figure:



In above figure the elements 10, 20, 30, 40, 50 are stored in array 'p' at individual index. That is the element 10 is stored at index 0, 20 is stored at index 1 like that, 50 is stored at index 4. Also all elements are stored in continuous memory location.

Note:

Index is nothing but position of the element in an array.

2) Two Dimensional array:

" The array having **two subscripts** is called as Two dimensional array or matrix"

The two dimensional array is used to perform matrix operations.

Declaration Syntax:

datatype array_name[rowsize][columnsize];

here;

datatype = any valid datatype in C language i.e. int, char, float etc.

'rowsize' is integer value that denotes total number of rows in array

'columnsize' is integer value that denotes total number of columns in array

Note:

Matrix is nothing but collection of rows and columns.

e.g.

1) int x[3][2];

here ;

'x' is array which can holds total 6 integers at a time.

3 is rowsize i.e. there are 3 rows in matrix 'x'

2 is columnsize i.e. there are 2 columns in matrix 'x'

Initialization of Two dimensional array:

Two dimensional array can be initialized as fallow;

e.g. int z[3][3] = { 5 , 3 , 6 , 1 , 7 , 8, 9 , 4 , 2 } ;

OR

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

here;

'z' is integer type two dimensional array which stores nine integers at a time. The storage of all elements in array 'z' is shown in following figure:

row index ↓	**0**	**1**	**2**	← column index
0	5	3	6	
1	1	7	8	
2	9	4	2	

In above figure the individual element of array 'z' is also accessed by following way:

The element 8 can be accessed as z[1][2]

The element 7 can be accessed as z[1][1]

The element 9 can be accessed as z[2][0] etc.

like that we can access all individual elements of matrix 'z'

3) Multi Dimensional array:

"The array having **more than two subscripts** is called as multi-dimensional array"

Declaration Syntax:

datatype array_name[size1][size2]……[sizeN];

here;

datatype = any valid datatype in C language i.e. int, char, float etc.
array_name = an identifier (name given by programmer)
size1, size2,……,sizeN are **integer constants** and that denotes total number of elements stored in an array.

e.g. 1) int z[2][3][2];
The above array is the multidimensional array having three subscripts and which stores 12 integer values at a time.
2) float x[2][2][2][3];
The above array 'x' is also multidimensional array having four subscripts and which stores 24 float values at a time.

Operations of Array:

There are following operations of array:-

1) insert ( ) 2) traverse( ) or display ( ) 3) reverse( ) 4) delete( )

Note: To perform all these operations, we have to declare one array having name 'item' in global section of program and 'cnt' variable that store array element count. As follow-

```
int item[10];
int cnt =0;
```

Let's see all these functions in details

1) insert ( ) or add( ) –

This operation inserts new element into array. Always new element is inserted at next index.
Operation-

```
Void insert(int x )
{
    static int i = 0;
    item[i] = x;
    i++;
    cnt++;
}
```

2) traverse ( ) or display( ) –

Traverse means visiting or retrieving every element of array. By using this function we can traverse in an array from 0^{th} index to n-1^{th} index. (Here, 'n' is size of an array). While traversing, we display all elements of array.
Operation-

```
Void traverse( )
{
    int i ;
    for( i = 0; i<=cnt-1; i++)
    {
        cout<< "\t"<<item[i];
    }
}
```

3) reverse ( )–

This function is used to reverse the array. To reverse, we can traverse in an array from n-1^{th} to 0^{th} index (Here, 'n' is size of an array). While traversing, we display all elements of array.
Operation-

```
Void reverse( )
{
    int i ;
    for( i = cnt-1; i>= 0; i--)
    {
        cout<< "\t"<<item[i];
    }
}
```

4) remove( ) or delete ( )–

This function is used to delete the element from an array. To delete appropriate element in an array, we just have to shift array element to left side.
Operation-

```
int      remove( int   pos )
{
          int    i , z;
          if ( pos < 0 || pos > cnt-1 )
          {
                    cout<<" Delete operation not performed";
          }
          else
          {
           z= item[pos];
          for( i = pos; i<= cnt; i++)
          {
              item[i]=item[i+1];
          }
          cnt-- ;
          return(z);
}
```

Following program shows implementation of array:

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
#define n  10
int   item[n];
int   cnt=0;
void insert(int);
void traverse();
void reverse();
int remove(int);
void    main()
{

   int x,pos,z,ch;
   clrscr();
   do
   {
          cout<<"\nEnter your choice= ";
          cout<<"\n1:Insert\n2:Delete\n3:Display\n4:
Reverse\n5:Exit";
          cin>>ch;
          switch(ch)
          {
          case 1:
          cout<<"\nEnter val to insert=";
                              cin>>x;
                              insert(x);
                              break;
          case  2:
cout<<"\nEnter position of element to remove=";
          cin>>pos;
          z=remove(pos);
cout<<"\nRemoved Element="<<z;
          break;
case  3:
          traverse();
          break;
case  4:
          reverse();
          break;
case  5:
          exit(1);
```

```
void      insert(int     x )
{
                    static    int    i = 0;
                    item[i] = x;
                    i++;
                    cnt++;
}

void      traverse( )
{
          int    i ;
          for( i = 0; i<=cnt-1; i++)
          {
                    cout<<"\t"<<item[i];
          }
}
void      reverse( )
{
  int    i ;
  for( i = cnt-1; i>= 0; i--)
 {
   cout<<"\t"<<item[i];
 }
}

int      remove( int   pos )
{
          int    i , z;
if ( pos < 0 || pos > cnt-1 )
{
cout<<"Delete operation not performed";
return 0;
}
else
{
        z= item[pos];
          for( i = pos; i<= cnt; i++)
          {
              item[i]=item[i+1];
          }
          cnt-- ;
          return(z);
```

```
        }
   }while(ch!=5);
getch();
}
```

```
}
}
```

Theory Assignment-I

1) What is Array? Explain all types of array in details.
2) Explain following operations of array-
   1) insert ( ) 2) traverse( ) or display ( ) 3) reverse( ) 4) delete( )

Practical Assignment-I

1) Write a program to implement array data structure with its operations.
2) Write a program that print only even numbers in an array.
3) Write a program that print only odd numbers in an array.
4) Write a program that print maximum & minimum number in an array.
5) Write a program to find addition of two matrices.
6) Write a program to find subtraction of two matrices.
7) Write a program to find multiplication of two matrices.