DATA STRUCTURES

****

Data Structures - Definition, Classification of Data Structures, Operations on Data Structures, Abstract Data Type (ADT), Preliminaries of algorithms. Time and Space Complexity.

****

DATA STRUCTURES : Data may be organised in many different ways. The logical or mathematical model of a particular organisation of data is called a data structure.

Example : Arrays,
Linked lists,
Trees,
Structures,
Stacks,
Queues,
Graphs,
Algebraic expressions

What is an Algorithm: An algorithm is any well defined computational procedure that takes some values as input and produces some values as output. It is thus a sequence of computational steps that transform input into output.

An algorithm is composed of finite steps each of which may require one or more operations. Each operation may be characterized as either simple or complex.
An algorithm is a step by step procedure for performing some tasks in a finite amount of time.

According to **D.E.Knuth** a pioneer in the computer science discipline an algorithm must have the following 5 basic properties

1. Input
2. Output
3. Finiteness
4. Definiteness
5. Effectiveness

Input: An algorithm has 0 or more inputs that are given to it initially before it begins (or) dynamically as it runs.

Output: An algorithm has 1 or more outputs that have a specified relation with the inputs. An algorithm produces at least one or more outputs.

Finiteness: An algorithm must always terminate after a finite number of steps.

Definiteness: Each operation specified in an algorithm must have definite meaning. Each step of the algorithm must be precisely defined. Instructions such as "compute x/0 " or "subtract 7 or 6 to x" are not permitted because it is not clear which of the two possibilities should be done or what the result is.

Effectiveness: Each operation of an algorithm should be effective. i.e. The operation must be able to be carried out in a finite amount of time. Tracing of each step should be possible.

Algorithm Vs Program: In computational theory, we distinguish between an algorithm and a program. Program does not have to satisfy the finiteness condition. For example, we can think of an operating system that continues in a "wait" loop until more jobs are entered. Such a program does not terminate unless the system crashes. Since our programs always terminate, we use "algorithm" and "program" interchangeably.

Pseudo Code Conventions

Algorithm specification: Pseudo code conventions:

1) Comments begin with // and continue until the end of line.
2) Blocks are indicated with matching braces: { }. A compound statement can be represented as a block.
3) All statements are separated or terminated by a semicolon(;).
4) An identifier begins with a letter followed by letters or digits or underscore.
5) Assignment of values to the variables is done using the assignment operator :=
6) There are two Boolean values true and false
7) Elements of single dimensional and multidimensional arrays.
8) Loops : while, do-while, repeat until, for

```
Ex1)  while (condition)
      {
        statement1;
        statement2;
         -----
         -----
      }
```

```
Ex2)  do
      {
        statement1;
        statement2;
          -----
          -----
      } while (condition);
```

```
Ex3)  Repeat
      {
        -----
        ------
      }until(condition);
```

```
Ex4)  for var:=valu1 to value2 do
      {
        -----
        ------
      }
```

9) Conditional statements : if then else are written in the following form

```
Ex1)  if (condition) then
                statement;
Ex2) if (condition) then
              statement1
        else
              statement2
```

10) Case statement

```
            case
            {
              condition1: statement1
              condition2: statement2
               ------
            }
```

11) Input & output
12) Structure: algorithm is a procedure consisting of a heading and body.

```
Algorithm name_of_alg(parameters)
{
        ----
        ----
}
```

Ex1) Algorithm to find sum of elements of a single dimensional array

```
Algorithm addition(a,n)
// a is an array of size n
{
  sum:=0;
  for i:= 1 to n do
     sum:=sum+a[i];
  return sum;
}
```

Ex 2) Algorithm to find largest number from the elements of a single dimensional array

```
Algorithm find_large(a,n)
// a is an array of size n
{
  large:=a[1];
  for i:= 2 to n do
     if (a[i]>large) then large:=a[i];
  return large;
}
```

Ex 3) Algorithm to find smallest number from the elements of a single dimensional array

```
Algorithm find_small(a,n)
// a is an array of size n
{
  small := a[1];
  for i:= 2 to n do
     if (a[i]< small) then small := a[i];
  return small;
}
```

Ex 4) Algorithm to find Factorial of a given integer number

```
Algorithm factorial(n)
{
  fact:=1;
   for i:= 1 to n do
     fact:= fact*i;
  return fact;
}
```

Ex 5) Algorithm to find ncr

```
Algorithm find_ncr(n,r)
{
   fn:=1;
   for i:= 1 to n do
     fn:= fn*i;
   fr:=1;
   for i:= 1 to r do
     fr:= fr*i;
   k := n-r;
   fk:=1;
   for i:= 1 to k do
     fk:= fk*i;
   ncr:= fn / (  fr * fk);
  return ncr;
}
```

Ex 6) Algorithm to search for a target element using linear search method

```
Algorithm linear_search(a,n,tar)
// a is an array of size n
{
      pos:=0;
      for i:= 1 to n do
         if (a[i]=tar) then pos:=i;
      return pos;
}
```

Ex 7) Algorithm to search for a target element using Binary search method

```
Algorithm binary_search(a,n,tar)
// a is an array of size n
{
      low := 1;
      high := n;
      while ( low<=high)
      {
          mid:=(low+high)/2;
          if (tar = a[mid])  then pos:=mid;
          else
          if (tar>a[mid]) then low:= mid+1;
          else high := mid – 1;
       }
      return pos;
}
```

Ex 8) Algorithm to sort elements of a single dimensional array

```
Algorithm buble_sort(a,n)
// a is an array of size n
{
     for i:=1 to n-1 do
       for j:= 1 to n-i do
          if ( a[j]>a[j+1]) then
             {
                 temp:=a[j];
                  a[j]:= a[j+1];
                  a[j+1]:=temp;
             }}
```

Ex 9) Algorithm to sort elements of a single dimensional array using selection sort technique.

```
Algorithm selection_sort(a,n)
// a is an array of size n
{
     for I := n downto 2 do
       {
            k:= 1;
            for j:= 2 to i do
               if (a[j]>a[k]) then k:=j;
            temp:=a[i];
            a[i]:= a[k];
            a[k]:=temp;
        }
}
```

PERFORMANCE ANALYSIS

1. Space Complexity
2. Time Complexity

Algorithm analysis refers to the task of determining the computing time and storage requirements of an algorithm. It is also known as performance analysis which enables us to select an efficient algorithm.

We can analyze an algorithm by two ways.
1)By checking the correctness of an algorithm.
2)By measuring the time and space complexity of an algorithm.

To Compute the analysis of algorithm, two phases are required. They are
1)Priori analysis
2)Posteriori analysis

1)Priori analysis: This is one of the creative analysis of algorithms. In this we obtain a function which bounds the algorithms computing time.
Suppose there is a statement in the middle of a program. We wish to determine the total time that the statement will spend for the execution. This requires

i)The statements frequency count i.e. the number of times the statement will be executed

ii)The time taken for one execution

The product of these two numbers is the total time.

The priori analysis is mainly concerned with the order of determining the magnitude.
The notation used in priori analysis are Big-oh(O), Omega(Ω), theta(Θ) and small-oh(o). Priori analysis of computing time ignores all of the factors, which are machine or programming language dependent and only concentrates on determining the order of the magnitude of the frequency of execution of the statements.

2)Posteriori Analysis: in this we will collect the actual statistics about the algorithm, in conjunction of the time and space while executing. Once the algorithm is written it has to be tested. Testing a program consists of two major phases.

> **i) Debugging** : It is the process of executing programs on sample data sets that determine whether we get proper results. If faulty results occurs it has to be corrected.
>
> **ii) Profiling** : it is the process of executing a correct program on a actual data sets and measuring the time and space it takes to compute the results during execution. The actual time taken by the algorithm to process the data is called profiling.

Space Complexity: The space complexity of an algorithm is the amount of memory it needs to run to completion.

Time Complexity: The time complexity of an algorithm is the amount of computer time it needs to run to completion.

Space Complexity: The space needed by the algorithms is seen to be the sum of the following components:

1) A fixed part that is independent of the characteristics of the inputs and outputs. This part typically includes the instruction space, space for simple variables, space for constants etc..

2) A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved; the space needed by reference variables and the recursion stack space.

   The space requirement S(P) of any algorithm P may therefore be written as S(P)=c+Sp(instance characteristics).

```
Algorithm abc(a,b,c)
 {
      Return a+b+b*c + (a+b-c)/(a+b)+4.0;
 }
```

In this algorithm the problem instance is characterized by the specific values of a,b and c. making the assumption that one word is adequate to store the values of each of a,b and c, and the result, we see that space needed by abc is independent of the instance characteristics,
S_p (instance characteristics)=0
Space requirement s[p] = 3+0 =0
One space for each a,b,c
Space complexity is O(1)

```
Algorithm addition(a,n)
// a is an array of size n
{
  sum:=0.0;                                  ---------1
  for i:= 1 to n do                          ---------1          1
     sum:=sum+a[i];                          ---------n
  return sum;
}
```

Space needed by n is one word; space needed by a is n words; space needed by i and sum one word each.
S(addition)=3+n
Space complexity O(n)

```
Algorithm add(a,n)
// a is an array of size n
{
    if (n=0) return 0.0;
    else
    return a[n]+add(a,n-1);
}
```

The instances are characterized by n. The recursion stack space includes space for the formal parameters, the local variables, and the return address. Assume that the return address requires only one word of memory. Each call to add requires at least three words(n, return address, pointer to a[]). Since the depth of recursion is n+1, the recursion stack space needed is >=(3(n+1).
Recursion –stack space = 3(n+1) = 3n + 3 = O(n)

Time Complexity: The time taken by a program P is the sum of the compile time and the run time. The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will run several times without recompilation. Consequently we concern ourselves with just the run time of a program. This runtime is denoted by tp(instance characteristics).

A program step is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics. For example : Let us consider the statement in the following algorithm.

```
Algorithm abc(a,b,c)
{
     Return a+b+b*c + (a+b-c)/(a+b)+4.0;
}
```

The entire statement could be regarded as a step since its execution time is independent of the instance characteristics. The number of steps any program statement is assigned depends on the kind of statement. For example comments count as 0 steps. Assignment statement which does not involve any calls to other algorithms is counted as one step. In an iterative statement such as the for, while and repeat-until statements, we consider the step counts only for the control part of the statement.

Statement	s/e	frequency	Total Steps
Algorithm addition(a,n)	0	-	0
// a is an array of size n	0	-	0
{	0	-	0
sum:=0.0;	1	1	1
for i:= 1 to n do	1	n+1	n+1
sum:=sum+a[i];	1	n	n
return sum;	1	1	1
}	0	-	0
Total			2n+3

The number of steps per execution and the frequency of each of the statements in addition algorithm have been listed. The total number of steps required by the algorithm is determined to be 2n+3. It is important to note that the frequency of for statement is n+1 and not n. this is so because i has to be incremented to n+1 before the for loop can terminate.

Statement	s/e	Frequency		Total Steps	
		n=0	n>0	n=0	n>0
Algorithm add(a,n)	0	-	-	0	0
// a is an array of size n	0	-	-	0	0
{	0	-	-	0	0
if (n=0) then	1	1	1	1	1
return 0.0;	1	1	0	1	0
else	0	-	-	0	0
return a[n]+add(a,n-1);	1+x	0	1	0	1+x
}	0	-	-	0	0
Total				2	2+x

$x=t_{add(a,n-1)}$

Notice that under the s/e column, the else clause has been given a count of $1 + t_{add(a,n-1)}$ it includes all steps that get executed as a result of the invocation of add() from the else clause. The frequeny and total steps columns have been split into two parts: one for the case n=0 and the other for the case n>0.

Statement	s/e	frequency	Total Steps
Algorithm addmat(a,b,c,m,n)	0	-	0
// a,b,c are two dimensional arrays	0	-	0
// m,n are no of rows and cols	0	-	0
{	0	-	1
for i:= 1 to m do	1	m+1	m+1
for j:= 1 to n do	1	m(n+1)	mn+m
c[i,j]:=a[i,j]+b[i,j]	1	mn	mn
}	0	-	0
Total			2mn+2m+1

The frequency of first for loop is m+1. Similarly the frequency of frequency for the second for loop is m(n+1).

When you have obtained sufficient experience in computing step counts, you can avoid constructing the frequency table and obtain the step count as in the following example.

Example: The Fibonacci sequence of numbers starts as 0,1,1,2,3,5,8,13,21,34,55…..
Each new term is obtatined by taking the sum of the two previous terms.

```
Algorithm Fibonacci(n)    // compute the nth fibonacci number
{
   if  ( n<= 1) then  write (n);
      else
           {
                  first=0;
                  second=1;
                  for i:= 2 to n do
                    {
                        next = first+second;
                        first=second;
                        second=next;
                    }
              write(next);
           }
 }
```

To analyse the time complexity of this algorithm, we need to consider the two cases (1) n=0 or 1 and (2) n is > 1. The total steps for the case n>1 is 4n+1.

The order of growth:

The size of n is either increased or decreased. The time analysis may not be always linearly proportional, because it depends upon what kind of basic operations we have in the algorithm. Therefore, while analyzing an algorithm's time efficiency the order of growth of n is also important. We expect the algorithms must execute faster, but for as n varies the execution time varies. To understand the meaning of the order of growth, let us list the most common computing time functions.

s.no	Function	Name
1	1	Constant
2	log n	Logarithmic
3	n	Linear
4	n log n	n log n
5	n^2	Quadratic
6	n^3	Cubic
7	2^n	Exponential
8	n!	Factorial

The functions shown are quite common in the analysis of algorithms. These functions have strong significance in terms of their relative values. The growth of these common functions typical values for n can be considered

n	**log n**	**n log n**	**n^2**	**n^3**	**2^n**	**n!**
1	0	0	1	1	2	1
2	1	2	4	8	4	2
4	2	8	16	64	16	24
8	3	24	64	512	256	40320
16	4	64	256	4096	65536	20922789888000
32	5	160	1024	32768	$4.2x10^9$	$2.6x10^{35}$
10^5	17	$1.7x10^6$	10^{10}	10^{15}	-	
10^6	20	$2.0x10^7$	10^{12}	10^{18}	-	

The function log n grows very slowly compared to any other function with respect to n. But 2^n and n! grows very fast.

ASYMPTOTIC NOTATION (O, Ω, Θ)

Asymptotic means study of function of parameter 'n' and 'n' becomes larger and larger without bound. Here we are concerned about how the running time of an algorithm increases with the size of the input. Generally an algorithm that is asymptotically more efficient will be the best choice.

Big-Oh Notation (O): Big-oh notation gives an upper bound and a function f(n). The upper bound of f(n) indicates that the function f(n) will be the worst case that it does not consume more than this computing time.
Def: The function f(n) = O(g(n)) (read as f of n is big-oh of g of n) if and only if There exists positive constants c and n_0 such that f(n) <= c* g(n) for all n, n>= n_0.

Ex 1) Given f(n) = 3n+2, then prove that f(n) = O(n).
f(n) = 3n+2
= 3n+2 <= 5n for all n>=1
The above inequality can be satisfied using the definition of Big-oh-notation by setting c=5, g(n)=n and n_0 = 1.
Therefore f(n)= O(n).

Ex 2) Given f(n) = $20n^3$-3 then prove that f(n) = $O(n^3)$
f(n) = $20n^3$-3
$20n^3 - 3 <= 20n^3$ for all n>=1
The above inequality can be satisfied using the definition of Big-oh notation by setting c=20 and $g(n)=n^3$, n_0=1
Therefore f(n) = $O(n^3)$

Ex 3) Given f(n) = $10n^2 + 4n + 3$ then prove that f(n) = $O(n^2)$.
f(n) = $10n^2 + 4n + 3$
$10n^2 + 4n + 3 <= 10n^2 + 5n$ for all n>= 3
$5n <= n^2$ for all n>= 5
$10n^2 + 4n + 3 <= 10n^2 + n^2$ for all n>= 5
$10n^2 + 4n + 3 <= 11n^2$ for all n>= 5
The above inequality can be satisfied using the definition of Big-oh-notation by setting c=11 $g(n)=n^2$ and n_0=5

Therefore f(n)=$O(n^2)$.

We write O(1) to mean a computing time that is a constant
O(n) is called linear
$O(n^2)$ is called quadratic
$O(n^3)$ is called cubic
$O(2^n)$ is called exponential
O(log n) is called logarithmic
O(n log n) is called n log n

The relation among these computing time is
$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

Ex 4) f(n) = $n^3 \log n + n^2 \log n + n \log n$
Time complexity is O (n^3 log n).
Ex 5) f(n) = $n^2 + n^3 + 4^n = O(4^n)$
The time complexity is $O(4^n)$

Ex 6) for =1 to n div 2 do
for j=1 to n*n do
x=x+1
Time complexity = (n/2) n^2 = $n^3/2$ = O(n^3)

Omega Notation Ω : This notation is used to find the lower bound behavior of f(n). The lower bound implies that below this time the algorithm cannot perform better. It is represented mathematically by notation Ω.

Def: f(n)=Ω(g(n)) iff there exists two positive constants C and n_0 .
f(n)>=c.g(n) for all n>=n_0

Ex 1) Given f(n) = 3n + 2 then prove that f(n) = Ω(n).
3n+2 >= 3n for all n>=1
The above inequality is satisfied according to Ω definition by setting
C=3 g(n)=n n_0=1
Therefore f(n)=Ω(n)

Ex 2) Given f(n) = $50n^2$+10n -5 then prove that f(n) = Ω(n^2).
$50n^2$+10n -5 >= $50n^2$ For all n>=1
The above inequality is satisfied according to Ω definition by setting
C=50 g(n)=n^2 and n_0=1 Therefore f(n) = Ω(n^2)

Theta Notation θ
For some functions the lower bound and the upper bound may be same. i.e. big Oh and omega will have the same function. For example to find minimum or maximum of an array of elements the computing time is O(n) and Ω(n).

There exists a special notation to denote for functions having the same time complexity for lower and upper bounds and this notation is called Theta θ Notation.
Def: f(n)=θ (g(n)) iff there exists three positive constants c_1 c_2 and n_0 with the constraint that
c_1 g(n) <= f(n) <= c_2 (g(n) for all n >= n_0
Ex 1) f(n)=$^1/_2n^2$-3n θ(n^2)
Ex2 f(n)=3n+2 θ(n)
Ex3 f(n)=10n2+4n+2 θ(n^2)
Ex 4 f(n)=10xlogn+4 θ(log n)

Little-Oh Notation (o) : The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound $2n^2$=O(n^2) is asymptotically tight, but the bound 2n = O(n^2) is not. We use o-notation to denote an upper bound that is not asymptotically tight.

Definition : f(n) = o(g(n), iff f(n) < c.g(n) for any constants c > 0, n_0>0 and n>n_0.
Or

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

Examples
Ex1) Given f(n)=3n+2 The time complexity = o(n^2)
Ex2) Given f(n)=2^n, then prove that f(n)=o(n!).

$$f(n) = \lim_{n\to\infty} \frac{2^n}{n!}$$

=0
Therefore f(n)=o(n!)

BEST CASE, WORST CASE AND AVERAGE CASE EFFICIENCIES

1) Best Case : It is the minimum number of steps that can be executed for a given parameter.
2) Worst case : It is the maximum number of steps that can be executed for a given parameter.
3) Average case: It is the average number of steps executed for a given parameter.

Ex 1) Linear search (Sequential search)

Best Case:

2	4	5	6	8	9	10	11	13
1	2	3	4	5	6	7	8	9

If we want to search an element 2, whether it is present in the array or not, first A[1] is compared with 2. Match occurs. So the number of comparisons is only one. It is observed that search takes minimum number of comparisons, so it come under best case.
Time complexity is O(1).

Average Case: If we want to search an element 8, whether it is present in the array or not . first A[1] is compared with 8, no match occurs. Compare A[2],A[3] and A[4] with 8, no match occurs. Up to now 4 comparisons taken place. Now compare A[5] and 8 so match occurs. The number of comparisons are 5. It is observed that search takes average number of comparisons. So it comes under average case. If there are n elements, then we require n/2 comparisons. Time complexity is O(n/2) which is O(n). we can neglect constant.

Worst Case : If we want to search an element 13, whether it is present in the array or not. First A[1] is compared with 13. No match occurs. Continue this process until element is found or the list exhausted. The element is found at 9^{th} comparison. So number of comparisons are 9. It is observed that search takes maximum number of comparisons. So it comes under worst case.
Time complexity is O(n)

Note : If the element is not found in the list then we have to search entire list, so it comes under worst case.

Ex 2) Binary Search: The array elements are in ascending/descending order. The target is compared with middle element. If it is equal we stop our search otherwise we repeat the same process either in the first half or second half of the array depending on the value of target.
Best case time complexity O(1) : If the target found in the first comparison then it is called best case
Average case time complexity O(log n) :Average number of comparisons
Worst case time complexity O(log n) : maximum number of comparisons
Assume that the number of elements is considered as 2^m as every time the list is divided into two halfs.
$n = 2^m$
$\log (n) = \log(2^m)$
$\log (n) = m \log(2)$
$m = \log(n)/\log(2)$
$m = \log_2 n$ i.e. log n base 2
$m = \log (n)$
Maximum number of comparisons will be **m** in a binary search.