1

INTRODUCTION

1.1 ALGORITHM

Algorithm originates from the name of Latin translation of a book written by al-khwarizmi a Persian mathematician. The book was entitled: “Algoritmi de numero Indorum”. The term “Algoritmi” in the title of the book led to the term “Algorithm”. In mathematics and computer science, an algorithm is a step-by-step procedure for calculations.

An algorithm is an effective method for finding out the solution for a given problem. It is a sequence of instructions that conveys the method to address a problem. Algorithm is the basis for any program in computer science. Algorithm helps in the organization of the program. Every program is structured and follows the specified guidelines and algorithm gives steps for the program. A problem can be solved by means of logical induction but in order to implement it in the form of a program, we require an appropriate algorithm.

Formal Definitions

***Definition* 1:** Step by step procedure to solve a problem is called Algorithm.

***Definition* 2:** A countable set of instructions followed to complete the required task is called an algorithm.

***Definition* 3:** An algorithm is a finite set of instructions to do a particular task.

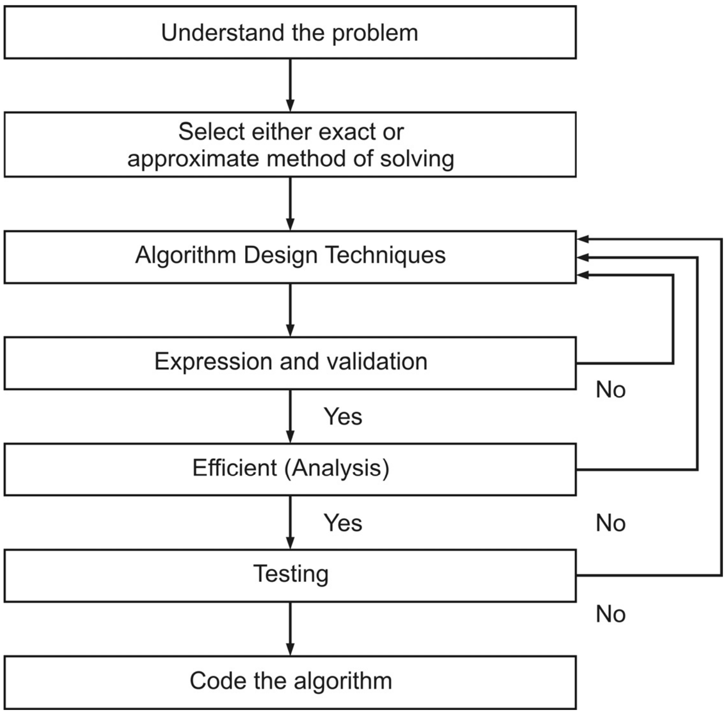
Various criteria are considered to evaluate an algorithm among which the essential ones are as follows:

1. **Input:** The algorithm should be given zero (or) more inputs explicitly. (≥ 0)

2. **Output:** One or more quantities are produced as an outcome. (> 0)
3. **Definiteness:** Every instruction must be free from ambiguity. (A = B + 3 or 5)
4. **Finiteness:** The algorithm must halt after finite number of steps for all the cases.
5. **Effectiveness:** Every step in the algorithm should be easy to understand and must be implementable through any of the programming languages.

Each step of algorithm must be subjected to one or more operations. For a computer to operate on the algorithm, certain constraints must be imposed on the operations performed on it. The time required to terminate an algorithm must be short sensibly. All the operations performed on an algorithm must be solvable manually within a limited amount of time. Computational procedures are those algorithms which are definite and effective. When an algorithm is implemented in a programming language, it gains this criterion.

1.2 LIFE CYCLE OF DESIGN AND ANALYSIS OF ALGORITHM


Understand the problem
Select either exact or approximate method of solving
Algorithm Design Techniques
Expression and validation
No
Yes
Efficient (Analysis)
Yes
No
Testing
No
Code the algorithm

Understand the Problem: Before designing the algorithm, we need to understand the problem completely. This is a critical phase. If we do any mistake in this phase, the entire algorithm becomes wrong. So, we need to gather all the requirements from the user regarding problem. After that we have to find out the necessary inputs for solving that problem. The input to the algorithm is called "instance" of the problem.

Exact vs. Approximate Solving: Solve the problem exactly if possible, otherwise use approximation methods. Though some problems are solvable by exact method, they are solved faster using approximation method. So in such situations, we will use approximation method.

Algorithm Design Techniques: An important aspect of this book is to study different design techniques which yielded good algorithms. Designing new algorithms becomes easy once these design strategies are clearly understood. Designing an algorithm needs human intervention and cannot be done by a machine alone. Some of the techniques used in designing algorithms are:

1. Brute force
2. Divide and Conquer
3. Greedy Method
4. Dynamic Programming
5. Back Tracking
6. Branch and Bound

Depending on the nature of the problem suitable design technique is adopted.

Expressing an algorithm: The algorithms are written using structured programming principles. This includes writing comments where ever necessary, providing indentation, adhering to standards etc. Algorithms can be described in the following three ways.

1. Natural language like English: When this way is used, care should be taken and we should ensure that each and every statement is definite.
2. Graphic representation called flowchart: This method will work well when the algorithm is small and simple.
3. Pseudo-code Method: In this method, we should typically describe algorithms as program, which resembles language like Pascal and algol.

Algorithm validation: Checking whether the algorithm is producing appropriate outputs for all the valid inputs is called algorithm validation. The algorithm must be independent of limitations in a programming language in which it is implemented. It must not be effected by any programming language issues.

The solution can be expressed as a set of assertions about the input and output variables and also as expression in predicate calculus. If these two forms are proved to be equivalent, then it is a correct solution. Once the algorithm is built, next we have to prove its correctness. Usually validation is used for proving correctness of the algorithm i.e., it should be tested by proving all possible combinations of the inputs. Some of the techniques used for generating the inputs are:

1. ***Boundary Value Technique*:** In Boundary Value Technique, if the correct value of a Variable is 10, it is tested by giving values in and around 10. (ex. 9,9.5, 10.5,11 etc.,)
2. ***Equivalence Partitioning*:** In Equivalence Partitioning, the i/p is divided into different groups and tests the algorithm to work satisfactorily by giving values from these groups.
3. ***Random Generation*:** Random Generation will test the algorithm by generating set of input values in random.

Analysis of an algorithm: Analyzing an algorithm involves study of data storage and processing of data which are performed by a computer. It measures the evaluation time of an algorithm and space required by it. Analysis is required to compare the performance of algorithms.

Algorithm analysis involves synthesizing a formula or guessing the fastness of algorithm depending upon the size of the problem it operates. Size of the problem can be

(a) Number of inputs/outputs in an algorithm.

E.g., For a multiplication algorithm, the numbers to be multiplied are the inputs and the product of the numbers is the output.

(b) Total number of operations involved in algorithm:

E.g., To find the minimum of all the elements in an array the number of comparisons made among the elements is the total number of operations.

Algorithm testing: Testing of an algorithm involves debugging and profiling. Debugging refers to finding out errors in the results and correcting the problem. Profiling refers to the measurement of performance of a correct program when executed on a data set. This includes calculation of time and space required for computation.

Coding an algorithm: After successful completion of all the phases, then an algorithm is converted into program by identifying a suitable computer language.

1.3 PSEUDO-CODE FOR EXPRESSING ALGORITHMS

1. Comments are denoted by '//'
2. Block of statements are enclosed within braces '{ }'.
3. No need of explicit declaration of a variables data type. Identifier starts with a character.
4. Records are used for the formation of complex data types. Here is an example,

```
Node = Record
{
  data type – 1   data – 1;
        :
        :
  data type – n  data – n;
  node * link;
  }
```

   Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period operators.

5. Values can be assigned to variables as follows:

   <Variable>: = <expression>;

6. TRUE and FALSE are two Boolean values present.

   → Logical Operators   AND, OR, NOT
   → Relational Operators   <, <=, >, >=, =, !=

7. Loop control statements used are for, while and repeat-until.

   while Loop:

```
        while < condition > do
        {
                <statement-1>
                      .
                      .
                      .
                <statement-n>
         }
```

   for Loop:

```
for variable: = value-1 to value-2 step step do
{
<statement-1>
      .
      .
      .
<statement-n>
}
```

repeat-until:

```
repeat
          <statement-1>
               .
               .
               .
          <statement-n>
          until<condition>
```

8. A conditional statement has the following forms.

```
If <condition> then <statement>
If <condition> then <statement-1>
Else <statement-2>
```

Case statement:

```
case
{
: <condition-1> : <statement-1>
                .
                .
                .
: <condition-n> : <statement-n>
: else : <statement-n+1>
}
```

9. Input and output are done using the instructions read and write.
10. A single procedure exists which is of the form: Algorithm.

    the heading takes the form,

    Algorithm Name ___ of __ Algorithm (List of Parameter)

As an example, the following algorithm calculates first 'n' natural numbers and returns the result:

Algorithm

```
Algorithm Sum(n)  // n is number of natural numbers
{
NumSum:=0
for i:= 1 to n do
NumSum:= Numsum + i
return NumSum;
}
```

In the above example, Sum is the name of algorithm, 'n' is parameter. NumSum and 'i' are local variables.

Case study: Selection Sort

Here is an example that demonstrates the method of transforming a problem into an algorithm.

- Let us consider a problem of sorting 'n' arbitrary elements in non decreasing order for which algorithm must be devised.
- A Simple solution given by the following.
- Find the smallest among the unsorted elements and place it first in the sorted list.

Algorithm

```
1:  For i := 1 to n do
2:  {
3:  Examine x[i] to x[n] and suppose the smallest
    element is x[min];
4:  swap x[i] and x[min];
5:  }
```

The above algorithm has two tasks:

(i) Finding the smallest element

(ii) Swap

The first task can be solved by assuming the minimum is x[i]; checking x[i] with x[I + 1], x[I + 2].......x[n], and whenever a smaller element is found, regarding it as the new minimum which is x[min]. x[n] is compared with the current minimum. The second task swap can be solved using the following code

```
temp:= x[i];
x[i]:=x[min];
x[min]:=temp;
```

Putting all these observations together, we get the algorithm Selection sort.

In the first step of this algorithm, the smallest element in the given list is identified. It is placed in the beginning of the list in the next step. A similar procedure is followed with the remaining elements where each time the least element among the elements is found and placed in appropriate position until the whole list is in sorted order.

Example: List L = 15, 25, 20, 5, 10

5 is identified it is swapped with 15, $\rightarrow$ 5, 25, 20, 15, 10

10 is identified it is swapped with 25, $\rightarrow$ 5, 10, 20, 15, 25

15 is identified it is swapped with 20, $\rightarrow$ 5, 10, 15, 20, 25

20 is identified it is swapped with 20, $\rightarrow$ 5, 10, 15, 20, 25

Algorithm

```
Algorithm selection sort (x, n)
// Sort the array x[1:n] into non-decreasing
//order.
{
for i:=1 to n do
{
min:=i;
for k:=i+1 to n do
if (x[k]<x[min]) then min:=k;
temp:=x[i];
x[i]:=x[min];
x[min]:=temp;
}
}
```

1.4 RECURSIVE ALGORITHMS

Ability of a function to call itself is called recursion. Recursive algorithms are potential mechanisms and can express a convoluted code in a lucid manner. That is why, in certain cases, recursive algorithms are preferable. Recursion is of two types (i) Direct recursion (ii) Indirect recursion

A direct recursive algorithm straight away calls itself. If 'P' is an algorithm that calls an algorithm 'Q' in its body. If 'Q' in turn calls 'P', then it is said to be indirect recursive algorithm.

The following two examples show how to develop recursive algorithms.

In the first, we consider the Towers of Hanoi problem, and in the second, we generate all possible permutations of a list of characters.

1.4.1 Towers of Hanoi

Towers of Hanoi is a classical example of recursion problem. It is easier, efficient and do not make use of complex data structure. The task is to move all the three disks from source to destination without violating the following rules.

1. At a time, only one disk may be moved.
2. Larger disk should not be placed on top.
3. Each time a move is made, upper disk from one tower must be moved to another tower.
4. For the purpose of intermediate storage of disks, only one auxiliary tower should be used.

There is an interesting story behind the concept of Towers of Hanoi. It believed that when the world was brought into existence there was a tower accommodated with 64 golden disks. It is also called Tower of Brahma or Luca’s Tower. The disks were of different sizes arranged in a stack in ascending order. There were two more empty towers beside the actual tower. Let them be auxiliary, destination and actual tower be source. The disks must be transferred from source to destination using the middle tower auxiliary.

It is said that Brahmin presents have been moving these disks, according to the unchangeable rules of the Brahma, since that time and when once this puzzle is completed, the world would come to an end.

Note: If there are ‘n’ disks then the minimum number of moves to solve the Towers of Hanoi problem is $2^n - 1$

Algorithm for Towers of Hanoi problem: Name of the algorithm is TOH and n disks are to be moved from source s to destination d using auxiliary tower a.

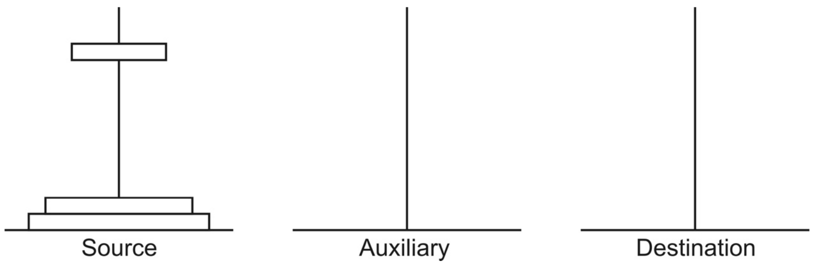
Algorithm

```
Algorithm TOH(n, s, d, a)
{
If (n>=1) then
{
TOH(n-1,s,a,d);
Write("move top disk from tower", s, "to top of
tower", d);
TOH(n-1,a,d,s);
}
}
```

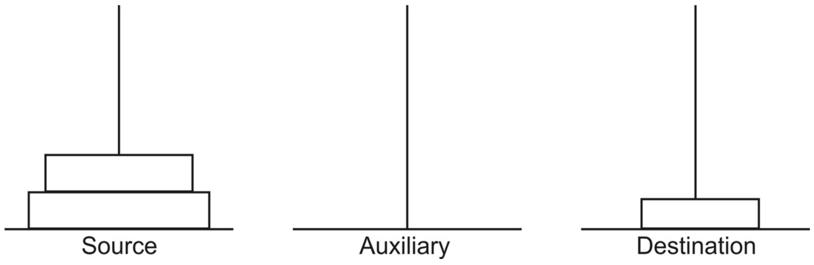
1.4.1.1 Solution to Towers of Hanoi for Three Disks

In this problem, there are three towers named source, auxiliary and destination. The source tower consists of three disks of different sizes, where each disk resting on the one just larger than it.

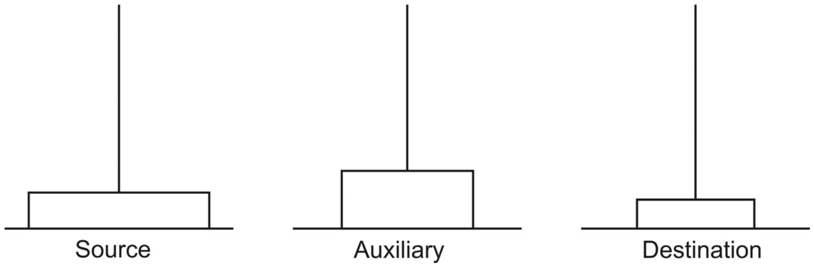
Start: The source tower consisting of 3 disks.



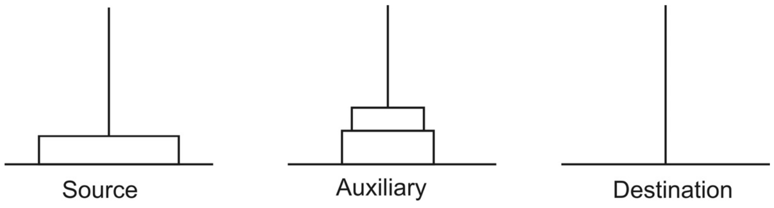
Step 1: Move disk 1 from source to destination.



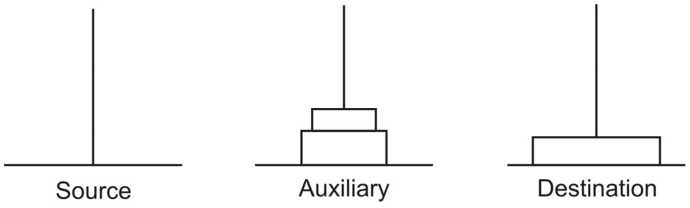
Step 2: Move disk 2 from source to auxiliary.



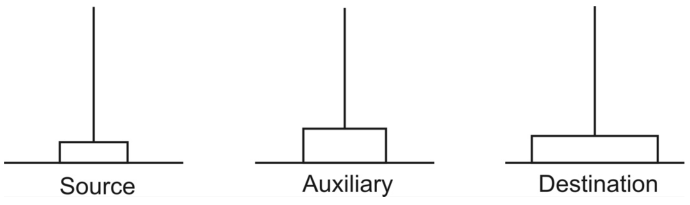
Step 3: Move disk 1 from destination to auxiliary.



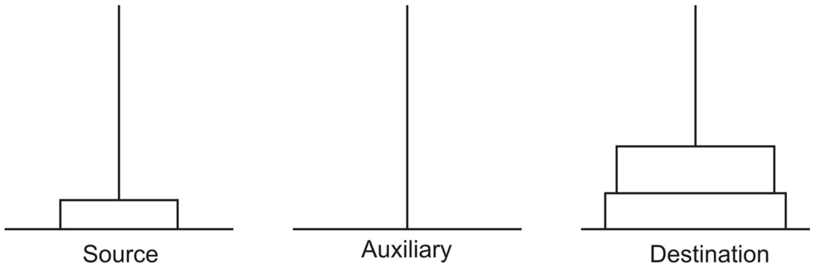
Step 4: Move disk 3 from source to destination.



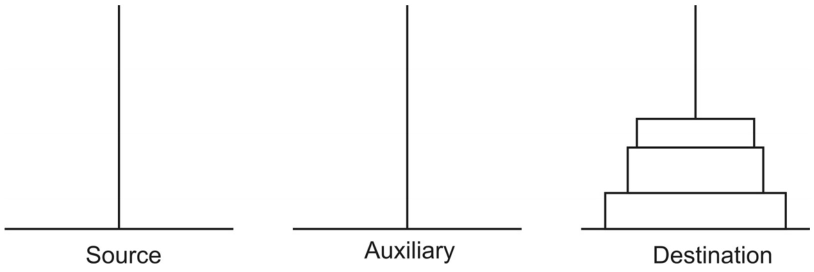
Step 5: Move disk 1 from auxiliary to source.



Step 6: Move disk 2 from auxiliary to destination.



Step 7: Move disk 1 from source to destination.



Thus, the destination consists of three disks satisfying the above rule.

Analysis:

Number of disks = 3

Number of times disk 1 moved = $2^2 = 4$

Number of times disk 2 moved = $2^1 = 2$

Number of times disk 3 moved = $2^0 = 1$

Total number of moves = $2^3 - 1 = 7$

Note: Number of movements of each disk is in power of 2

1.4.1.2 Solution to Towers of Hanoi for Four disks

***Step* 1:** Move disk 1 from Source to Auxiliary.

***Step* 2:** Move disk 2 from Source to Destination

***Step* 3:** Move disk 1 from Auxiliary to Destination.

***Step* 4:** Move disk 3 from Source to Auxiliary.

***Step* 5:** Move disk 1 from Destination to Source

***Step* 6:** Move disk 2 from Destination to Auxiliary

***Step* 7:** Move disk 1 from Source to Auxiliary

***Step* 8:** Move disk 4 from Source to Destination

***Step* 9:** Move disk 1 from Auxiliary to Destination

***Step* 10:** Move disk 2 from Auxiliary to Source

***Step* 11:** Move disk 1 from Destination to Source

***Step* 12:** Move disk 3 from Auxiliary to Destination

***Step* 13:** Move disk 1 from Source to Auxiliary

***Step* 14:** Move disk 2 from Source to Destination

***Step* 15:** Move disk 1 from Auxiliary to Destination

Analysis:

Number of disks = 4

Number of times disk 1 moved = $2^3 = 8$

Number of times disk 2 moved = $2^2 = 4$

Number of times disk 3 moved = $2^1 = 2$

Number of times disk 4 moved = $2^0 = 1$

Total number of moves = $2^4 - 1 = 15$

NOTE: Number of movements of each disk is in power of 2

1.4.1.3 C Program to Solve Towers of Hanoi Problem

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void Towers(int n, char from, char to, char aux)
{
  if(a>=1)
  {
    Towers(n-1,from,aux,to);
    printf("Move disk %d from %c to %c \n",n, from,to);
    Towers(n-1,aux,to,from);
  }
return;
}
void main()
{
  int disk, moves;
  clrscr();
  printf("Enter number of disks  : ");
scanf("%d",&disk);
moves=(pow(2,disk)-1);
printf("Number of moves needed are  :  %d\n",moves);
Towers(disk,'s','d','a'); getch();
  }
```

1.4.1.4 Permutation Generator

The problem is to print all possible permutations of a set containing 'k' elements (k>=1).

For example, if the set is {x, y, z}, then the set of permutation is,

{ (x, y, z),(y, z, x),(z, x, y),(x, z, y),(y, x, z),(z, y, x)}

- Hence for 'k' elements there are k! permutations
- A simple algorithm can be obtained by looking at the case of 4 statement(x, y, z, w)
- The Answer can be constructed by writing

1. x followed by all the permutations of (y, z, w)
2. y followed by all the permutations of (x, z, w)
3. z followed by all the permutations of (x, y, w)
4. w followed by all the permutations of (x, y, z)

Algorithm

```
Algorithm perm(a,k,n)
{
if(k=n) then write (a[1:n]); // output permutation
else   //a[k:n] ahs more than one permutation
// Generate this recursively.
for i:=k to n do
{
t:=a[k];
a[k]:=a[i];
a[i]:=t;
perm(a,k+1,n);
//all permutation of a[k+1:n]
t:=a[k];
a[k]:=a[i];
a[i]:=t;
}
}
```

1.5 PERFORMANCE ANALYSIS OF ALGORITHM

Study of algorithms is required for the following reasons:

- To estimate the computational time and space required while operating a problem.
- To prove that your method for obtaining solution ends after finite steps yielding a correct solution.
- To pick-up an efficient algorithm to solve a problem among the available algorithms.

1.5.1 Efficiency

Efficiency of an algorithm depends upon the amount of resources utilized by the algorithm. A maximum efficient algorithm will exhibit the property of minimal resource utilization and vice versa. Though two algorithms are designed to solve same problem, they may have different efficiencies.

For example, consider a problem of sorting ‘n’ elements. If an Insertion sort technique is adopted to solve this problem, it takes an approximate time equal to C_1n^2 (C_1 is constant). If the same problem is solved using Merge sort technique, it takes C_2 n $\log_2$ n units of time (C_2 is constant). For small inputs Insertion sort is better than merge sort. But for larger inputs Merge sort is better than Insertion sort because its running time grows more slowly with increase in input size compared to that of Insertion sort (i.e., n log n grows more slowly than n^2)

Analysis of algorithm is the study of algorithm i.e., calculating the time and space complexity. Analysis of algorithms is of two types Priori analysis and posteriori analysis.

1.5.2 Priori Analysis

It is also known as performance analysis. Analysis of algorithm is done before running the algorithm on any computer machine i.e., before executing algorithm, we will study the behavior of the algorithm. It gives an estimate about the running time of the algorithm.

We find the order of magnitude of an algorithm before a program is written and executed. This analysis is machine and platform independent. The first objective of priori analysis is to associate a mathematical function in terms of input size ‘n’ representing the rate of growth of the time of algorithm as a function of input size. Some algorithm may follow a constant growth that is irrespective of the input size, it takes constant growth. While other algorithm may have logarithmic growth like O(log n) and some may have polynomial growth like n, n^2 while some may have exponential growth like 2n, 3n. It is desirable to have algorithm for problems that take polynomial growth as they take less time than in comparison to exponential time algorithm. It is less accurate when compared to posteriori analysis, but cost of analysis is low.

Advantage of priori analysis:

1. Analysis is done before implementing or running the algorithm on machine.
2. Simple and uniform. Hence easier for making performance.

Drawback: We get only estimated value i.e., not real values.

1.5.3 Posteriori Analysis

It is also called performance measurement. During the stage of analysis, the target must be identified. Algorithm is converted to a program and run on a machine. While algorithm is executed, then the information collected regarding execution time and primary memory requirements is called as posteriori analysis. It gives accurate values and is very costly.

Note: Priori algorithm is always better than posteriori analysis.

Advantage of posteriori: Values we get are real or exact in actual units of time.

Drawback

1. Difficulty in conducting experiments.
2. Difficulty in making performance comparison, because of non-uniform values of a single algorithm.

Analysis of algorithm means developing a formula or prediction of how fast the algorithm works on the problem size.

Problem size of an algorithm: Size of the problem is based on kind of the problem dealt with.

Example 1: If an element is to be searched a p-element array, size of problem = Size of array = p.

Example 2: If the elements of two arrays of sizes ‘p’ and ‘q’ are to be merged then size of problem = p + q.

Example 3: If factorial of a number ‘p’ is to be computed, then size of the problem = ‘p’.

1.5.4 Complexity

Complexity is the time taken and the space required for an algorithm for its completion. It is a measurement through which one can judge the quality of an algorithm and can be used for finding and sorting out better algorithms. Complexity can be classified into two types. Normally:

1. Space complexity
2. Time complexity

1.5.4.1 Space Complexity

The complexity can be defined as the amount of space an algorithm requires.

The space needed by each of the algorithm is the sum of the following components.

1. A fixed part
2. A variable part

1. **Fixed part:** It depends on the characteristics of input and output. That consists of space needed by component variable whose size is dependent on the particular problem instance being solved.
2. **Variable part:** The space requirement denoted by S(p) of any algorithm 'p' can be given as $S(p) = C + S_p$ (Instance characteristics) where, C is constant.

1.5.4.2 Time Complexity

Time Complexity of a program 'p' can be defined as the sum of compile time and execution time (runtime). Let us suppose that we compile a program for once and we run it several times. Then runtime of program will be considered.

Time complexity is the amount of time required by the algorithm for the completion of the problem. In this we have three cases (i) best case (ii) worst case (iii) Average case.

1. **Best case:** If an algorithm takes minimum amount of time to run to completion for a specific set of input, then it is called best case time complexity.

   E.g., While searching a particular element by using sequential search we get the desired element at first place itself then it is called best case time complexity.
2. **Worst case:** If an algorithm takes maximum amount of time to run to completion for a specific set of input, then it is called worst case time complexity. Defining the other way, it is maximum amount of time taken by an algorithm to give an output is called as worst case.

   E.g., While searching an element by using linear searching method if desired element is placed at the end of the list then we get worst case time complexity.
3. **Average case:** The average time taken by an algorithm to run to completion is called average case.

This classification gives the complexity information about the behavior of the algorithm at a particular instance and of an algorithm on specific in random algorithm.

Case Study – Linear Search:

1	2	3	4	5	6	7	8	9
10	20	15	08	20	30	50	40	25

Output in this problem is of two types: 1. Successful Search 2. Unsuccessful Search

1. **Successful Search:** Element to be searched is present in the array. Minimum numbers of comparisons are required when we are searching for 1st element i.e., 10 in this problem and number of comparisons are 1, if numbers of elements are increased from 9 to 100 numbers of comparisons are not changed. This is best case for linear search.

   Here algorithm is same only we are changing the input, we get the output in least amount of time and it is O(1) or O(c) or O(k) where c & k are constants, O(1) does not mean 1 comparison, it means algorithm takes constant time independent of number of inputs.

   Maximum number of comparisons are required if we search for last element i.e., 25 in this problem, number of comparison equal to number of elements i.e., 9 in this problem. As number of elements is increasing, number of comparisons also increases. So time taken is dependent on number of elements and it is linear. This case falls under Worst case.

   Worst case for Linear Search = O(n).

2. **Unsuccessful Search:** If the element to be searched is not present in the array, then search is unsuccessful. Number of comparison required is ‘n’. So in this case Best case, Worst case and Average case, all are same and it is equal to O(n).

   Best case = Worst case = Average case = O(n)

   Time complexity of an algorithm can be found in three ways:

   (i) Brute force method (ii) Step count method (iii) Asymptotic Notation

1.5.4.2.1 Brute Force Method

Time Complexity of a program ‘p’ can be defined as the sum of compile time and execution time (runtime). Let us suppose that we compile a program for once and we run it several times. Then runtime of program will be considered. Let the runtime be t_p.

t_p = time taken to perform all the operations present in the program(addition, subtraction, comparison etc).

As t_p is influenced by many other factors and these factors are unknown at the time of conceiving of the program, only estimation of t_p is possible.

Let us assume that we know the characteristics of the compiler to be used. Let us determine the number of additions, subtractions, multiplications, divisions, comparisons, loads, and stores etc., that are performed by code for p.

Hence, an expression for t_p is as follows:

$$\mathbf{t_p (k) = C_a\, ADD(k) + C_s\, SUB(k) + C_m\, MUL(k) + C_d\, DIV(k) +} \ldots\ldots$$

'k' denotes instance characteristics and C_a, C_s, C_m, C_d etc., represent the time required for additions, subtractions, multiplications, divisions respectively. ADD, SUB, MUL, DIV are the functions that represent the number of additions, subtractions, multiplications, divisions etc that are performed when the code for 'p' is used on an instance with characteristic 'k'.

But it is difficult to obtain such an exact formula because the time of execution depends upon the numbers being operated upon. Also in the case of a multi-user system the execution time depends upon numerous factors such as system load, number of other programs being run on the computer at that particular instance.

1.5.4.2.2 Step Count Method

A program step is defined as a meaningful statement of a program that has an execution time that is independent of the instance characteristics. It is assumed that all statements have same cost (i.e., execution time). We can determine the total number of steps needed by the program by counting the total number of steps.

Order of Magnitude of an algorithm: Each algorithm contains a finite number of statements. Each statement occurs one or more times. The sum of number of occurrences of all the statements contained in an algorithm is called order of magnitude of an algorithm.

Example:

```
For (i = 0; i < n; i++)
     {
---------   ⎫
---------   ⎬     'k' statements
----------  ⎭
     }
```

Let us assume in above program, there are 'k' statements enclosed in the for loop and statement taken one unit of time for execution. Hence the execution of 'k' statements requires 'k' units of time. If these 'k' statements are executed 'n' times, then execution time is n*k units.

Hence order of magnitude of above algorithm = n*k units.

1. Few examples for finding out number of steps using step count method:

   Example:

```
Algorithm Sum()
{
read (a,b,c,d);               ← 1 unit
x=a+b+c+d;                    ← 1 unit
write(c)                      ← 1 unit
}                             -------------
                                3 units
                              -------------
```

The above algorithm has Four inputs, output and number of steps are three.

Step count =3;

2. Finding the sum of n numbers stored in the array a "n" is known as instance characteristics (input size):

```
Algorithm Sum(a,n)
{
    sum:=0.0;                 ← 1 unit
    for i:=1 to n do          ← n+1 unit
    sum:=sum + a[i];          ← n unit
    write(sum);               ← 1 unit
}                             -------------
                               2n+3 units
                              -------------
Step count = 2n + 3
```

3. Read n values from keyboard find their sum and average:

```
Algorithm Avg( )
{
sum:=0.0;               ← 1 unit
Read n;                 ← 1 unit
for i:=1 to n do        ← n+1 unit
 {
   Read num;            ← n unit
   Sum=sum + num;       ← n unit
 }
 Avg=sum/n              ← 1 unit
 Write(sum, avg);       ← 1 unit
 }
                        -------------
                          3n+5 units
                        -------------
```

Step count = 3n + 5

4. Addition of matrix A and matrix B of dimension m × n storing result in matrix C:

```
Algorithm matadd(a,b,c,m,n)
{
  for i:=1 to m do            ← m+1 unit
   for j:=1 to n do           ← m(n+1) unit
  c[i,j]=a[i,j]+b[i,j];       ← mn unit
}                             --------------
                              2mn+2m+1 units
                              --------------
```

Step count = 2mn + 2m + 1

5. Finding nth fibonacci number:

```
Algorithm Fibonacci(n)
{
   if (n<=1) then                   ← 1 unit
         write(n);
   else
{
    fib1 := 0;                      ← 1 unit
    fib2 := 1;                      ← 1 unit
    for i := 2 to n do              ← n+1 units
    {
      fib := fib1 + fib2;           ← n units
      fib1 := fib2;                 ← n units
      fib2 := fib;                  ← n units
    }
      Write (fn);                   ← 1 unit
   }
}                                   --------------
                                      4n+5 units
                                    --------------
```

Step count = 4n + 5

6. Finding the sum of n numbers stored in the array a using recursion:

```
Algorithm RecSum(a,n)               ← T(n) unit
{
  if(n<=0) then                     ← 1 unit
  {
     Return 0;
}
else
{
return RecSum(a, n-1) + a[n]; ← T(n-1) + b units
}
}
```

Solution: $T(n) = 1 \quad n = 0$

$T(n) = T(n-1) + b \quad n > 0$

Step count method for C programs:

1.
```
main( )
{
int a,b,c;
scanf("%d%d",&a,&b);              ← 1 unit
c=a+b;                            ← 1 unit
printf("%d",c)                    ← 1 unit
}                                   ------------
                                       3 units
                                    ------------
```
Step count = 3;

2.
```
main( )
{
int n;
printf("enter a number");         ← 1 unit
scanf("%d", &n);                  ← 1 unit
for(int i=0;i≤n;i++)              ← 2n+2 unit
printf("shyam");                  ← n unit
}                                -------------
                                    3n+4 units
                                 -------------
```
Step count = 3n + 4;

1.5.5 Asymptoic Notation: [Formal Definitions]

We use same simple abstractions to simplify algorithm analysis. First we ignored the actual cost of each statement, assumed that all statements have same cost (i.e., execution time). Then we observed that handling these constants are also complex, the running time of any sorting algorithm is $an^2 + bn + c$ for some +ve constants a, b & c that depend on the statement cost.

As the Size of 'n' is very large we are interested only in growth function of the running time of the algorithm. We are interested in finding out as 'n' is increased running time is also increased or not. If running time is not increased as 'n' is increased then the running time is independent of 'n' and time complexity is O(1).

If the running time is increased as 'n' is increased, then running time is dependent on 'n'. If the running time of an algorithm is $an^2 + bn + c$. We consider only the leading term of the formula (eg. an^2) since the lower order terms are relatively insignificant for large n.

We also ignore the coefficient of the leading term, Since they are less significant then the rate of growth.

This study of growth functions are called Asymptotic analysis of algorithm, and they are denoted by Asymptotic notations.

This asymptotic notation contains five types they are as follows.

1. Big-Oh Notation(O-Notation)
2. Omega Notation(**Ω**-Notation)
3. Theta Notation (Θ Notation)
4. Little –oh Notation(o-Notation)
5. Little omega Notation (ω-Notation)

1.5.5.1 Big-Oh Notation (O- Notation)

Big –Oh notation denoted by 'O' is a method of representing the upper bound or worst case of algorithm's run time. Using big-oh notation, we can give longest amount of time taken by the algorithm to complete.

Definition: Let f(n) and g(n) are two non-negative functions. The function **f(n) = O(g(n))** if there exist positive constants n_0 and c such that **f(n) ≤ c*g(n)** for all n, $n>n_0$.

→ Means order at most

→ Used to measure the worst case time complexity.

Find Big oh for the following functions:

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

$f(n) = O(g(n))$	
$f(n) <= cg(n)$	$n>=n_0$
$2n^2 + 3n + 1 <= 1$	false
$2n^2 + 3n + 1 <= 3n$	false
$2n^2 + 3n + 1 <= n^2$	false
$2n^2 + 3n + 1 <= 2n^2$	false
$2n^2 + 3n + 1 <= 3n^2$	true for n > = 4

   $f(n)<=3n^2$

   $f(n) = O(n^2)$ where $c = 3$ and $n_0= 4$

2. | | |
   |---|---|
   | $f(n) = 5n + 4$ | $n > = n_0$ |
   | $5n + 4 < =4$ | false |
   | $5n + 4 < = 5n$ | false |
   | $5n + 4 < = 6n$ | true for n > = 4 |

   $f(n)= O(n)$ where $c = 6$ and $n_0 = 4$

3. $f(n) = 10n^3 + 6n^2 + 6n + 2$

   $10n^3 + 6n^2 + 6n + 2 <= 11n^3 \quad n >= 7$

   $f(n) = O(n^3)$ where $c = 11$ $n_0 = 7$

Consider the function $f(n) = 2n + 2$ and $g(n) = n^2$ we have to find constant c so that $f(n) \leq g(n)$, in other words $2n + 2 \leq n^2$ then we find that for n = 1 or 2, f(n) is greater than g(n) that means for c = 1 when n = 1, f(n) = 4 and g(n) = 1, for n = 2, f(n) = 6 and g(n) = 4. When $n \geq 3$ we obtain $f(n) \leq g(n)$.

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

$5n + 2 = O(n)$ as $5n + 2 <= 6n$ for all $n >= 2$.

$109n + 6 = O(n)$ as $109n + 6 <= 110n$ for all $n >= 6$.

$13n^2 + 6n + 2 = O(n^2)$ as $10n^2 + 4n + 2 <= 14n^2$ for all $n >= 6$.

$4*2^n + n^2 = O(2^n)$ as $4*2^n + n^2 <= 5*2^n$ for $n >= 4$.

Various meanings associated with Big-Oh are

O(1) - Constant computing time

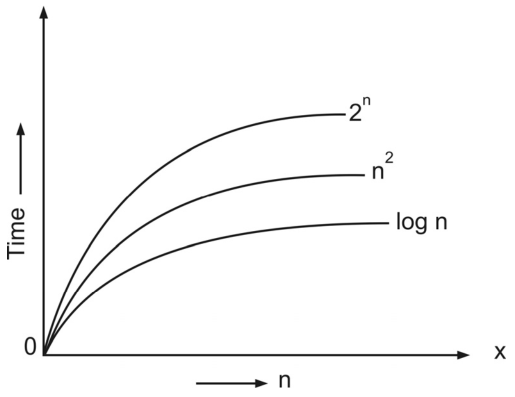
O(n) – Linear

$O(n^2)$ – Quadratic

$O(n^3)$ – Cubic

$O(2^n)$ – Exponential

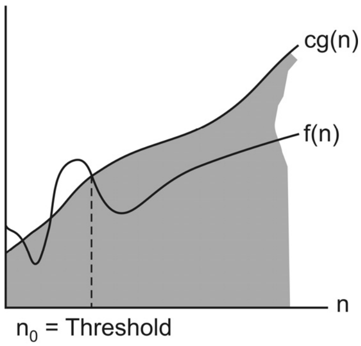
O(logn) – Logarithmic



The relationship among those computing time is 0 (1) < 0 (logn) < 0 (n) < 0 (n^2) < 0 (2^n)

$O(2^n)$ – rate of growth is very high (algorithm takes slower)

O(log n) – rate of growth is very less (algorithm takes faster)

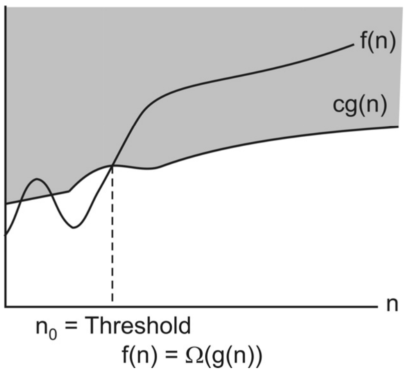


f(n) = O(g(n))

g(n) is an asymptotic upper bound for f(n).

1.5.5.2 Omega Notation (Ω-Notation)

Omega notation denoted as 'Ω' is a method of representing the bound of algorithm running time using omega notation we can denote shortest amount of time taken by algorithm to complete.



f(n) = Ω(g(n))

***Definition*:** Let f(n) and g(n) are two non-negative functions. The function f(n) = Ω(g(n)) if there exist positive constants C and n_0 such that f(n) > c*g(n) for all n, n ≥ n_0.

→ Means order at least.

→ Used to measure best case time complexity.

g(n) is an asymptotic lower bound for f(n).

Example: Consider f(n) = 2n + 5 and g(n) = 2n then 2n + 5 ≥ 2n. for n > 1 hence 2n + 5 = Ω(n) 5n + 2 = Ω(n) as 5n + 2 > = 5n for n > = 1. (the inequality holds for n > = 0, but the definition of Ω requires an n_0 > 0).

109n + 6 = Ω(n) as 109n + 6 > = 109n for all n > = 1.

$13n^2 + 6n + 2 = \Omega(n^2)$ as $13n^2 + 4n + 2 >= 13n^2$ for all n > = 1.

$4*2^n + n^2 = \Omega(2^n)$ as $4*2^n + n^2 >= 4*2^n$ for n > = 1.

Observe also that

5n + 2 = Ω(1).

$13n^2 + 6n + 2 = \Omega(n)$.

$13n^2 + 6n + 2 = \Omega(1)$.

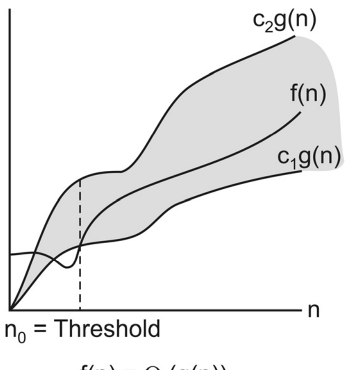
$4*2^n + n^2 = \Omega(n^2)$.

$4*2^n + n^2 = \Omega(n)$.

$4*2^n + n^2 = \Omega(1)$.

1.5.5.3 Theta Notation (Θ -Notation)

Theta notation denoted as 'Θ' is a method of representing running time between upper bound and lower bound.



f(n) = Θ (g(n))

Let f(n) and g(n) be two non negative functions. The function f(n) = ~ (g(n)) if there exist positive constants C_1, C_2 and n_0 such that $C_1g(n) \leq f(n) \leq C_2 g(n)$ for all n, n ≥ n_0.

→ Means order exactly.

→ Used to find average case time complexity

g(n) is an asymptotically tight bound for f(n).

E.g., if $f(n) = 2n + 8 > 5n$ where $n \geq 2$; $2n + 8 \geq 6n$ where $n \geq 2$ and $2n + 8 < 7n$ when $n \geq 2$. Hence $2n + 8 = \Theta(n)$ such that constants $C_1 = 5$, $C_2 = 7$ and $n_0 = 2$.

The theta notation is more precise than both big-oh and omega notation.

1.5.5.4 Little "oh" Notation (o-Notation)

Little oh notation is denoted as o. It is used to denote proper upper bound that is not asymptotically tight.

Let f(n) and g(n) are two non-negative functions. The function $f(n) = o(g(n))$ if there exist positive constants n_0 and c such that $f(n) < c*g(n)$ for all n, $n > n_o$.

$$\lim_{n\to\alpha} \frac{f(n)}{g(n)} = 0$$

E.g., The function $3n + 2 = o(n^2)$ since $\lim_{n\to\alpha} \frac{3n+2}{n^2} = 0$

$$f(n) = 3n + 2$$

$$f(n) < c\, g(n)$$

$$3n + 2 < n^2 \qquad n >= 4$$

$$\lim_{n\to\alpha} \frac{3n+2}{n^2} = 0$$

$$3n + 2 = o(n^2)$$

1.5.5.5 Little Omega Notation (ω-Notation)

Little omega notation is used to denote proper lower bound that is not asymptotically tight.

Let f(n) and g(n) are two non-negative functions. The function **$f(n) = w(g(n))$** if there exist positive constants n_0 and c such that **$f(n) > c*g(n)$** for all n, $n > n_0$.

$$\lim_{n\to\alpha} \frac{g(n)}{f(n)} = 0$$

Ex: The function $n^2 + 6$ since $\lim_{n\to\alpha} \frac{n}{n^2+6} = 0$

$$f(n) = n^2 + 6$$

$$f(n) > cg(n)$$

$$n^2 + 6 > n \quad n >= 1$$

$$\lim_{n \to \alpha} \frac{n}{n^2 + 6} = 0$$

$$n^2 + 6 > \omega(n)$$

1.5.6 Properties of Asymptotic Notations

Many of the relational properties of real numbers apply to asymptotic comparisons as well. For the following, assume that f(n) and g(n) are asymptotically positive.

Transitivity

$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$,

$f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O(h(n))$,

$f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$,

$f(n) = o(g(n))$ and $g(n) = o(h(n))$ imply $f(n) = o(h(n))$,

$f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$ imply $f(n) = \omega(h(n))$.

Reflexivity

$f(n) = \Theta(f(n))$,

$f(n) = O(f(n))$,

$f(n) = \Omega(f(n))$.

Symmetry

$f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.

Transpose symmetry

$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$,

$f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

Comparison of Asymptotic notations with real numbers: Let f(n) and g(n) be two non negative functions and a and b be two real numbers.

$f(n) = O(g(n)) \approx a \leq b$,

$f(n) = \Omega(g(n)) \approx a \geq b$,

$f(n) = \Theta(g(n)) \approx a = b$,

$f(n) = o(g(n)) \approx a < b$,

$f(n) = \omega(g(n)) \approx a > b$.

We say that f(n) is asymptotically smaller than g(n) if f(n) = o(g(n)), and f(n) is asymptotically larger than g(n) if f(n) = ω(g(n)).

One property of real numbers, however, does not carry over to asymptotic notation:

Trichotomy: For any two real numbers a and b, exactly one of the following must hold: a < b, a = b, or a > b, but all functions cannot be asymptotically comparable.

Any two real numbers can be compared, but not all functions are asymptotically comparable. Let f(n) and g(n) are two non-negative functions, a special case may exist for which neither f(n) = O(g(n)) nor f(n) = Ω(g(n)) are satisfied. For example, the functions n and $n^{1+\cos n}$ cannot be compared using asymptotic notation, since the value of 1 + cos (n) rotates between 0 and 2, taking on all values in between.

if f(n) = O(g(n)) and h(n) = O(g(n)) then f(n) + h(n) = O(max(g(n), d(n)))

if f(n) = O(g(n)) and h(n) = O(g(n)) then f(n) * h(n) = O(g(n) * d(n))

if f(n) = O(g(n)) Then a*f(n) = O(g(n)) where ‘a’ is constant.

1.5.6.1 Time Complexity of Program Segments with Loops

1.
   ```
   for i :=1 to n
   s;
   ```
   complexity: O(n)

2.
   ```
   for i:=1 to n
   for j:=1 to n
   s;
   ```
   complexity: $O(n^2)$

3.
   ```
   i=1, k=1;
   While(k<=n)
   {
   i++;
   k=k+i;
   }
   ```
   complexity: $O(\sqrt{2})$

4.
   ```
   for (i=1; i*i <=n;++i)
   S;
   ```
   complexity: $O(\sqrt{2})$

5.
   ```
   j=1;
   while(j<=n)
   {
   j =j*2;
   }
   ```
   complexity: O(log n)

6.
```
for i:=1 to n/2
for j:=1 to n/3
for k:=1 to n/4
s;
```
complexity: $O(n^3)$

7.
```
for i:=1 to n
for(j=1; j<=n; j=j*2)
s;
```
complexity: O(n log n)

8.
```
for i:=1 to n
for j:=1 to n
for k:=1 to n
{ s;
break;
}
```
complexity: $O(n^2)$

1.6 RECURRENCE RELATIONS

Definition 1: A recurrence relation is an equation or inequality that illustrates a function in terms of its value on smaller inputs. Special techniques are required to analyze the space and time required.

Definition 2: Recurrence relation is an equation that recursively defines a sequence, once one or more initial terms are given.

Each further term of the sequence is defined as a function of the preceeding terms.

A recurrence relation is the arrangement of a series of values in terms of previous values in the sequence and base values.

Solving Recurrence Relations: Solution to recurrence relation can be obtained using two methods: 1. substitution method 2. Masters method.

1.6.1 Substitution Method

In this method we consistently guess an asymptotic bound (upper or lower) on the solution, and trying to prove it by induction.

Example Problems:

1. $T(n) = T(n-1) + n \qquad n > 1$

   $T(n) = 1 \qquad n = 1$

Solution:

$$T(n) = T(n-1) + n \qquad \text{.....(1)}$$

$T(n-1) = T(n-2) + n - 1$ .....(2)

Substituting (2) in (1)

$T(n) = T(n-2) + n - 1 + n$ .....(3)

$T(n-2) = T(n-3) + n - 2$ .....(4)

Substituting (4) in (3)

$T(n) = T(n-3) + n - 2 + n - 1 + n$ .....(5)

General equation

$T(n) = T(n-k) + (n-(k-1)) + n - (k-2) + n - (k-3) + n - 1 + n$ .....(6)

$T(1) = 1$

$n - k = 1$

$k = n - 1$ .....(7)

Substituting (7) in (6)

$T(n) = T(1) + 2 + 3 + \ldots\ldots n - 1 + n$

$= 1 + 2 + 3 + \ldots\ldots + n$

$= n(n+1)/2$

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

2. $T(n) = T(n-1) + b \quad n > 1$

$T(n) = 1 \quad n = 1$

***Solution*:**

$T(n) = T(n-1) + b$ .....(1)

$T(n-1) = T(n-2) + b$ .....(2)

Substituting (2) in (1)

$T(n) = T(n-2) + b + b$

$T(n) = T(n-2) + 2b$ .....(3)

$T(n-2) = T(n-3) + b$ .....(4)

Substituting (4) in (3)

$T(n) = T(n-3) + 3b$

General equation

$T(n) = T(n-k) + k.b$

$T(1) = 1$

$n - k = 1$

$k = n - 1$

$T(n) = T(1) + (n-1)b$

$= 1 + bn - b$

$T(n) = O(n)$

3. $T(n) = 2\,T(n-1) + b \qquad n > 1$

$T(n) = 1 \qquad n = 1$

***Solution*:**

$T(1) = 1$

$T(2) = 2.T(1) + b$

$= 2 + b$

$= 2^1 + b$

$T(3) = 2T(2) + b$

$= 2(2 + b) + b$

$= 4 + 2b + b$

$= 4 + 3b$

$= 2^2 + (2^2 - 1)b$

$T(4) = 2.T(3) + b$

$= 2(4 + 3b) + b$

$= 8 + 7b$

$= 2^3 + (2^3 - 1)b$

General equation

$T(k) = 2^{k-1} + (2^{k-1} - 1)b$

.

.

.

$T(n) = 2^{n-1} + (2^{n-1} - 1)b$

$T(n) = 2^{n-1}(b + 1) - b$

$= 2^n(b + 1)/2 - b$

Let $c = (b + 1)/2$

$T(n) = c\,2^n - b$

$= O(2^n)$

4. $T(n) = T(n/2) + b \qquad n > 1$

$T(1) = 1 \qquad n = 1$

***Solution*:**

$T(n) = T(n/2) + b$ …..(1)

$T(n/2) = T(n/4) + b$ …..(2)

Substituting (2) in (1)

$T(n) = T(n/4) + b + b$

$= T(n/4) + 2b$ …..(3)

$T(n/4) = T(n/8) + b$ …..(4)

Substituting (4) in (3)

$T(n) = T(n/8) + 3b$

$= T(n/2^3) + 3b$

General equation

$T(n) = T(n/2^k) + kb$ …..(5)

$T(1) = 1$

$n/2^k = 1$

$2^k = n$

$K = \log n$ …..(6)

Substituting (6) in (5)

$T(n) = T(1) + b.\log n$

$= 1 + b \log n$

$T(n) = O(\log n)$

1.6.2 The Master Method

This is a cookbook method for determining asymptotic solutions to recurrences of a specific form. Master method provides solution to recurrence relation of the function.

$T(n) = aT(n/b) + f(n)$ $n > d$

$= c$ $n = d$

Where a, b, c and d are positive constants.

$a >= 1, b > 1, c >= 1, d >= 1.$

$T(n) = a\,T(n/b) + f(n),$

Where we interpret n/b to mean either $\lceil n/b \rceil$ $\lfloor n/b \rfloor$. Then T(n) can be bounded asymptotically as follows:

Case 1: If $f(n) = O((n^{\log_b a - €})$ for some constant $€ > 0$, then $T(n) = \theta(n^{\log_b a})$.

Case 2: If $f(n) = \theta(n^{\log_b a} \log^k n)$, then $T(n) = \theta\,(n^{\log_b a} \log^{k+1} n)$.

Case 3: $f(n) = \Omega(n^{\log_b a+€)}$ for some constant $€ > 0$, and if $af(n/b) <= cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \theta(f(n))$.

Use the master method to give tight asymptotic bounds for the following recurrences:

1. $T(n) = 4T(n/2) + n$ $n > 1$

   $T(n) = 1$ $n = 1$

   From the above recurrence relation we obtain

   $a = 4, b = 2, c = 1, d = 1, f(n) = n$

   $\log_b a = \log_2 4 = \log_2 2^2 = 2 \log_2 2 = 2$

   $n^{\log_b a} = n^2$

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

$n = O(n^2)$ It will fall in Case 1. So that

$T(n) = \theta(n^2)$

2. $T(n) = 4T(n/2) + n^2$ $n > 1$

   $T(n) = 1$ $n = 1$

   From the above recurrence relation we obtain

   $a = 4, b = 2, c = 1, d = 1, f(n) = n^2$

   $n^{\log_b a} = n^2$

   $f(n) = \theta(n^2)$

   $n^2 = \theta(n^2)$

   It will fall in case 2.

   $T(n) = \theta(n^2 \log n)$

3. $T(n) = 4T(n/2) + n^3$ $n > 1$

   $T(n) = 1$ $n = 1$

   From the above recurrence relation we obtain

   $a = 4, b = 2, c = 1, d = 1, f(n) = n^3$

   $n^{\log_b a} = n^3$

   $f(n) = \Omega(n^{\log_b a + €)}$

   $= \Omega(n^{2 + €})$

   $n^3 = \Omega(n^{2 + €})$

   This will fall in case 3.

   $T(n) = \theta(n^3)$

4. $T(n) = 2T(n/2) + n$ $n > 1$

   $T(n) = 1$ $n = 1$

   From the above recurrence relation we obtain

   $a = 2, b = 2, c = 1, d = 1, f(n) = n$

   $n^{\log_b a} = n^{\log_2 2} = n$

   $f(n) = \theta\, n^{\log_b a})$

   $= \theta(n)$

   It will fall in case 2.

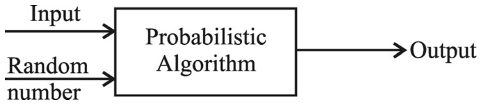
   $T(n) = \theta(n \log n)$

1.7 PROBABILISTIC ANALYSIS

To analyze the running time of an algorithm we use theory of probability. Probabilistic analysis is used to find out average running over all possible inputs. It is required to know about the distribution of the inputs to perform probabilistic analysis to compute an expected running time.

In some problems, the set of all possible inputs can be assumed, for other problems set of all possible inputs cannot be determined. Probabilistic analysis can be used to all problems where the set of all possible inputs can be assumed for other problems we cannot apply probabilistic analysis.

Input distribution should be known to apply probabilistic analysis. Some part of the algorithm behavior is randomized, for analysis and design of algorithms probability and randomness is used very frequently and these algorithms are called probabilistic or randomized algorithms.



In these algorithms a random bits generated through pseudo random generator are used as an auxiliary input to get the good performance in the average case. The performance of the algorithm will be determined by the random input and it is called the expected run time. The worst case is ignored as the probability of its occurrence is very less.

Example: An array of n elements having n/2 ‘1’s and n/2 ‘0’s, the problem is to search a ‘1’ in this array. If we use any deterministic algorithm it will take n/2 comparisons, which is very long, and it is not guaranteed for all possible inputs that the algorithm will complete quickly.

With high probability we can search quickly ‘1’ if we check elements at random for all possible inputs.

Randomized Algorithm is used in Quick sort where the complexity of the problem is reduced from $O(n^2)$ to $O(n \log n)$ for some set of input such as when the elements are already in sorted order.

1.8 AMORTIZED ANALYSIS

An important analysis tool useful for understanding the run times of algorithms that have steps with widely varying performance is “amortization”. The term “amortization” itself comes from the field of accounting, which provides an instance monetary metaphor for algorithm analysis.

Example: The above example provides a motivation for the amortization technique, which gives us a worst case way of performing an average case analysis. Formally, we define the amortized running time of an operation within a series of operations as the worst case running time of the series of operations divided by number of operations. When the series of operations is not specified, it is usually assumed to be a series of operations from the repertoire of a certain data structure starting from an empty structure. Thus amortized running time of each operation in the clearable table ADT is

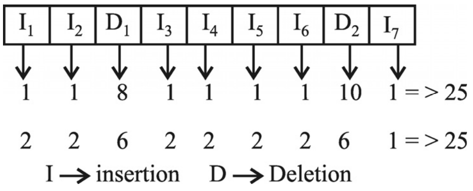
‘O(n)’. When we implement that clearable table with an array. Note that the actual running time of operation may be much higher than the amortized running time.

For example, a particular clear operation may take ‘O(n)’ time.

The advantage of using amortization is that it gives us a way to a robust average case analysis without using any probability.

Amortized complexity: In amortized complexity we change sum of the actual cost of an operation to other operation. The amortized cost of each insertion is no more than 2 and that of each deletion is no more than 6.

The actual cost of any insertion or deletion is no more than ‘2 * I + 6 * D’.



I_1 I_2 D_1 I_3 I_4 I_5 I_6 D_2 I_7
1 1 8 1 1 1 1 10 1 => 25
2 2 6 2 2 2 2 6 1 => 25
I → insertion D → Deletion

Amortized means finalizing the average run time per operation over a worst case sequence of operations.

Note: The only requirement is that the sum of the actual complexity of all operations in any sequence of operations be greater than or equal to their sum of actual complexity.

$$\sum_{1<=i<=n} \text{amortized}(i) \geq \sum_{1<=i<=n} \text{actual}(i)$$

Case Study 1: Linear Search

1	2	3	4	5	6	7	8	9
10	20	15	08	20	30	50	40	25

Number of comparisons required to search: 10 → 1 comparison
Number of comparisons required to search: 20 → 2 comparisons
Number of comparisons required to search: 15 → 3 comparisons
Number of comparisons required to search: 08 → 4 comparisons
Number of comparisons required to search: 20 → 5 comparisons
Number of comparisons required to search: 30 → 6 comparisons
Number of comparisons required to search: 50 → 7 comparisons
Number of comparisons required to search: 40 → 8 comparisons
Number of comparisons required to search: 25 → 9 comparisons

Total number of comparisons required to search all the elements in the array are 45.

Then amortized cost = total number of comparison/ total number of elements

= 45/9 = 5

Case Study 2: Binary Search

1	2	3	4	5	6	7
10	15	20	21	40	45	70

Number of comparisons required to search: 10 → 3 comparisons

Number of comparisons required to search: 15 → 2 comparisons

Number of comparisons required to search: 20 → 3 comparisons

Number of comparisons required to search: 21 → 1 comparison

Number of comparisons required to search: 40 → 3 comparisons

Number of comparisons required to search: 45 → 2 comparisons

Number of comparisons required to search: 70 → 3 comparisons

Total number of comparisons required to search all the elements in the array are 17.

Then amortized cost = total number of comparison/ total number of elements

= 17/7 = 2.43

1. low = 1, high = 7, mid = (1+7)/2 = 4 → 1 comparison
2. low = mid +1 = 5, high = 7, mid = (5 + 7)/2 = 6 → 2 comparisons
3. low = 1, high = mid – 1 = 3, mid = (1 + 3)/2 = 2 → 2 comparisons
4. low = mid +1 = 7, high = 7, mid = (7 + 7)/2 = 7 → 3 comparisons
5. low = 5, high = mid – 1 = 5, mid = (5 + 5)/2 = 5 → 3 comparisons
6. low = mid +1 =3, high = 3, mid = (3 + 3)/2 = 3 → 3 comparisons
7. low = 1, high = mid – 1 = 1, mid = (1 + 1)/2 =2 → 3 comparisons

Previous Gate Questions and Solutions

1. Consider the following three claims

1. $(n + k)^m = \Theta(n^m)$, where k and m are constants
2. $2^{n+1} = O(2^n)$
3. $2^{2n+1} = O(2^n)$

Which of these claims are correct?

(a) 1 and 2 (b) 1 and 3

(c) 2 and 3 (d) 1, 2 and 3

Answer: (a)

***Solution*:**

Consider each statement separately

I. $f(n) = (n + k)^m$

So, $f(n) = (1 + n)^m$

Assume k = 1 constant

$f(n) = 1 + {}^mC_{1n} + {}^mC_{2n} +$ --------- ${}^mC_m\, n^m$

$f(n) = O(n^m)$ which is correct

II. $f(n) = 2^{n+1}$

$f(n) = 2^n . 2^1$

$f(n) = 2.2^n$

$f(n) = O(2^n)$ which is correct

III. $f(n) = 2^{n+1}$

$f(n) = 2^{2n} .2^1$

$f(n) = 2.2^{2n}$

$f(n) = O(2^{2n})$ which is false

Then I and II are correct.

2. Let A [1,…., n] be an array storing a bit(1 or 0) at each location, and f(m) is a function whose time complexity is θ(m). Consider the following program fragment written in a C like language:

```
Counter = 0;
for(i=0;i<n;i++)
{if (A[i]==1)counter++;
else {f(counter);counter=0;
}}
```

The Complexity of the program fragment is:

(a) $\Omega(n^2)$ (b) $\Omega(n \log n)$ and $O(n^2)$

(c) $\theta(n)$ (d) $o(n)$

Answer: (c)

***Solution*:**

The given code is:

```
1. Counter = 0;
```

```
2.  for(i=0;i<n;i++)
3.  {if (A[i]==1)counter++;
4.  else {f(counter);counter=0};
5.  }
```

The time complexity of the program fragment depends on the frequency (Number of steps) of line 3 and 4. In line 4 the frequency depends on the variable counter and there is no increment in the counter variable which is initialize to 0, so f(0) then counter = 0 means there is no cell in an array which having a bit 0, so all cells in the array contain 1. Consider the line 3 if (A[i] = 1)counter++; the value of i will be increases up to n so the value of counter will be n. Since n is the frequency of the line 3 and the frequency of the line 4 is 0. So the time complexity of the line 3 is O(n) on average n and f(0) = O(1) is the time complexity of line 4. So the time complexity of the program fragment is maximum of line 3 and 4 which is O(n) on average.

3. The time complexity of the following C function is (assume n>0)

```
int recursive(int n) {
if(n==1)
return(1);
else
return(recursive(n-1) + recursive(n-1));
}
```

(a) O(n) (b) O(n log n) (c) $O(n^2)$ (d) $O(2^n)$

Answer: (d)

***Solution*:**

The given C function is recursive. The best way to find the time complexity of recursive function is that convert the code (algorithm) into recursion equation and solution of the recursion equation is time complexity of the given algorithm.

```
1.int recursive(int n) {
2.if(n==1) return(1);
3.else
4.return(recursive(n - 1) + recursive(n - 1));
5.}
```

The name of the function is recursive

Let recursive(n) = T(n)

According to line 2 if(n = 1) return(1)

Then the recursion equation is

$T(n) = 1 \quad n = 1$

According to line 4 recursion equation is

$T(n) = T(n-1) + T(n-1) \; n > 1$

Or $\quad T(n) = 2T(n-1) \; n > 1$

So the complete recursion equation is

$T(n) = 1 \;\; n = 1$

$T(n) = T(n-1) + T(n-1) \; n > 1$

Or $T(n) = 2T(n-1) \; n > 1$

$T(1) = 1 = 2^0$

$T(2) = 2T(1) = 2.1 = 2^1$

$T(3) = 2T(2) = 2.2 = 2^2$

$T(4) = 2T(3) = 2.3 = 2^3$

. . . .

. . . .

. . . .

$T(n) = 2^{n-1}$

Or $T(n) = 2^n.1/2$

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

4. Suppose $T(n) = 2T(n/2) + n$, $T(0) = T(1) = 1$ Which one of the following is FALSE?

(a) $T(n) = O(n^2)$ (b) $T(n) = \theta(n \log n)$

(c) $T(n) = \Omega(n^2)$ (d) $T(n) = O(n \log n)$

Answer: (a)

***Solution*:**

$T(0) = T(1) = 1$

$T(n) = 2T(n/2) + n$

$T(n) = 2T(n/2) + n \qquad n > 1$

$T(n) = 1 \qquad n = 1$

From the above recurrence relation we obtain

$a = 2, \; b = 2, c = 1, \; d = 1, f(n) = n$

$n \log_b a = n\log_2 2 = n$

$f(n) = \theta(n \log {}_b a)$

$\quad = \theta\,(n)$

It will fall in case 2.

$T(n) = \theta(n \log n)$

Implies $T(n) = O(n \log n)$

Implies $T(n) = \Omega(n \log n) \Rightarrow \Omega(n^2)$

$T(n)$ is not $O(n^2)$.

5. Consider the following is true?

$T(n) = 2T([\sqrt{2}]) + 1, T(1)=1$

Which one of the following is true?

(a) $T(n) = \theta(\log \log n)$ (b) $T(n) = \theta(\log n)$

(c) $T(n) = \theta(\sqrt{2})$ (d) $T(n) = \theta(n)$

Answer: (a)

***Solution*:**

$T(1) = 1$

$T(n) = 2T([\sqrt{2}]) + 1$

We know that $\log^2{}_2 = 1$

So, all the level sums are equal to $\log^2{}_2$

The problem size at level k of the recursion tree is n^{2-k} and we stop recursing this value is a constant. Setting $n^{2-k} = 2$ and solving for k gives us $2^{-k} \log_2 n = 1 \Rightarrow 2^k = \log_2 n \Rightarrow k = \log\log_2 n)$

So $T(n) = \theta(\log \log n)$

6. Consider the following functions:

$f(n) = 2^n$

$g(n) = n!$

$h(n) = n \log n$

Which of the following statements about the asymptotic behavior of f(n), g(n) and h(n) is true ?

(a) $f(n) = O(g(n)); g(n) = O(h(n))$

(b) $f(n) = \Omega(g(n)); g(n) = O(h(n))$

(c) $g(n) = O(f(n)); h(n) = O(f(n))$

(d) $h(n) = O(f(n)); g(n) = \Omega(f(n))$

Answer: (d)

Solution:

$f(n) = 2^n$

$\Rightarrow f(n) = O(2^n)$

$g(n) = n!$

$\Rightarrow g(n) = O(n!)$

$h(n) = n^{\log n}$

$\Rightarrow h(n) = O(n^{\log n})$

The Asymptotic order of the function is as follows $1 < \log\log n < \log n < n^e < n^c < n^{\log n} < c^n < n^n < c^{cn} < n!$

Where $0 < \in < 1 < c$ [$n < n^2$ means n grows more slowly than n^2]

So $n^{\log n} < c^n < n!$

$n^{\log n} < 2^n < n!$

Assume C = 2

$h(n) < f(n) < g(n)$

From the above relation we can arrive at $h(n) \in O(f(n))$ and $g(n) = O(f(n))$

7. Consider the Quick sort algorithm. Suppose there is a procedure for finding a pivot element which splits the list into sub-lists each of which contains at least one-fifth of the elements. Let T(n) be the number of comparisons required to sort n elements. Then

(a) $T(n) \le 2T(n/5) + n$ (b) $T(n) \le T(n/5) + T(4n/5) + n$

(c) $T(n) \le 2T(4n/5) + n$ (d) $T(n) \le 2T(n/2) + n$

Answer: (b)

Solution:

If we want to sort n elements with the help of quick sort algorithm. If pivot elements which split the lists into two sub lists each in which one list contains one-fifth element or n/5 and other list contains 4n/5 and balancing takes n so

$T(n) \le T(n/5) \le T(4n/5) + n$

[**Note:** $n - n/5 = 5n - n/5 = 4n/5$]

8. The running time of an algorithm is represented by the following recurrence relation

$$T(n) = \begin{cases} n & n \ge 3 \\ T\left(\frac{n}{3}\right) + cn & \text{otherwise} \end{cases}$$

Which one of the following represents the time complexity of the algorithm?

(a) $\theta(n)$ (b) $\theta(n \log n)$

(c) $\theta(n^2)$ (d) $\theta(n^2 \log n)$

Answer: (a)

***Solution*:**

Complexity is decided for large values of n only,

So, $T(n) = T(n/3) + cn$ for $n > 3$

Using masters theorem

Here $a = 1$, $b = 3$, $\log_b a = \log_3 1 = 0$

$f(n) = cn = \theta(n^1)$

Since $n^{\log a} = n^0$ is below $f(n) = \theta(n^1)$

This belongs to case III of masters theorem,

Where the solution is

$T(n) = \theta(f(n)) = \theta(n)$

9. Two alternative packages A and B are available for processing a database having 10^k records. Package A requires $0.0001\ n^2$ time units and Package B requires $10n\log_{10}n$ time units to process n records. What is the smallest value of k for which package B will be preferred over A ?

(a) 12 (b) 10 (c) 6 (d) 5

Answer: (c)

***Solution*:**

A requires $0.0001\ n^2$ time units. A and B require $10\ n\log_{10}n$ time unit to process n records for package B will be preferred over A

So $0.0001\ n^2 < 10\ n\log_{10}n$

$n^2/10^5 < 10\ n\log_{10}n$

$n^2 = 10^6\ n\log_{10}n$

compare this constant with 10^K

So minimum value of $K = 6$ for which package B will be preferred over A.

10. Procedure A(n)

```
{
if (n<=2) return(1)
else
   return(A $\sqrt{2}$ ))
}
```

Solution:

$T(n) = c, n <= 2$

$T(n) = T(\sqrt{2}\ ) + 6, n > 2$

$T(n) = T(\sqrt{2}\ ) + 6$

$T(n) = T(n^{1/2}) + b \Rightarrow T(n^{1/2}) = T(n^{1/4}) + b$

$T(n) = T(n^{1/4}) + 2b$

$T(n) = T(n^{1/8}) + 3b$

$T(n) = T(n^{1/}2^{k}) + kb \Rightarrow T(n) = T(2^{1/}2^{k}) + kb$

$T(n) = T(2) + b \log_2{}^{l} \Rightarrow T(n) = c + b \log_2{}^{l} \Rightarrow T(n) = c + b \log \log_2 x \Rightarrow$

$O(\log \log n)$

Objective Question Bank

1. The step-by-step instructions that solve a problem are called ______ . [ ]
   - A. An algorithm B. A plan
   - C. A sequential structure D. None of the above
2. The primary tool used in structured design is a: [ ]
   - A. Structure chart B. Data-flow diagram
   - C. Program flowchart D. None of the above
3. System Study involves [ ]
   - A. Study of an existing system
   - B. Documenting the existing system.
   - C. Identifying current deficiencies and establishing new goals
   - D. All of the above
4. A problem's _____ will answer the question, "What information will the computer need to know in order to either print or display the output times?" [ ]
   - A. Input B. Output
   - C. Purpose D. None of the above
5. Documentation is prepared [ ]
   - A. At every stage B. At system design
   - C. At system analysis D. At system development
6. Problem analysis is done during [ ]
   - A. System design phase B. Systems analysis phase
   - C. Before system test D. All of the above

7. Two main measures for the efficiency of an algorithm are [ ]
   A. Processor and memory B. Complexity and capacity
   C. Time and space D. Data and space
8. The time factor when determining the efficiency of algorithm is measured by [ ]
   A. Counting microseconds
   B. Counting the number of key operations
   C. Counting the number of statements
   D. Counting the kilobytes of algorithm
9. The space factor when determining the efficiency of algorithm is measured by [ ]
   A. Counting the maximum memory needed by the algorithm
   B. Counting the minimum memory needed by the algorithm
   C. Counting the average memory needed by the algorithm
   D. Counting the maximum disk space needed by the algorithm
10. Which of the following case does not exist in complexity theory? [ ]
    A. Best case B. Worst case
    C. Average case D. Null case
11. Which of the following data structure store the homogeneous data elements? [ ]
    A. Arrays B. Records
    C. Pointers D. None
12. An algorithm that calls itself directly or indirectly is known as [ ]
    A. Sub algorithm B. Recursion
    C. Polish notation D. Traversal algorithm
13. ________ is a method of expressing algorithms by a collection of geometric shapes with imbedded descriptions of algorithmic steps [ ]
    A. Flow chart B. Chart
    C. Numerical method D. Differential method
14. ________ is the maximum no. of steps that can be executed for a given parameter [ ]
    A. Best case B. Worst case
    C. Average case D. None

15. The number of times a statement is executed is usually referred as [ ]

    A. Complexity B. Frequency count
    C. Both A & B D. None

16. Theta notation expresses [ ]

    A. Tight bounds B. Upper bounds
    C. Lower bounds D. Worst cases

17. Function g is an upper bound on function f if for all x, [ ]

    A. $g(x) \leq f(x)$ B. $g(x) \geq f(x)$
    C. $g(x) = f(x)$ D. $f(x) < g(x)$

18. An Algorithm must be always [ ]

    A. Terminate B. Executable
    C. Well-ordered D. All

19. Which of the following needed to Design an Algorithm [ ]

    A. Pseudo code B. Methodology
    C. Both A&B D. None

20. Transferring the values from the user to a variable, or vice-versa is [ ]

    A. Function B. Algorithm
    C. Expression D. All

21. Which one of the following search is not very efficient for large lists [ ]

    A. Binary B. Sequential
    C. Both A&B D. None

22. What are the major phases of performance evaluation [ ]

    A. A prior estimates B. A posterior testing
    C. Both A&B D. None

23. Theta notation expresses [ ]

    A. Tight bounds B. Upper bounds
    C. Lower bounds D. Worst cases

24. Consider the following functions:

    $f(n) = 2n$

    $g(n) = n!$

    $h(n) = n \log n$

Which of the following statements about the asymptotic behavior of f(n), g(n) and h(n) is true? [ ]

A. f(n) = O(g(n)); g(n) = O(h(n))

B. f(n) = Ω(g(n)); g(n) = O(h(n))

C. g(n) = O(f(n)); h(n) = O(f(n))

D. h(n) = O(f(n)); g(n) = Ω(f(n))

Fill in the Blanks

1. _______________ is composed of a finite set of steps, each of which may require 1 or more operations.
2. A _______________ is the expression of an algorithm in a programming language.
3. Algorithms that are definite and effective are also called _______________
4. Program proving is also called _______________
5. _______________ refers to the task of determining how much computing time and storage an algorithm requires.
6. _______________ is the process of executing programs on sample data sets to determine whether faulty results occur and, if so, to correct them.
7. Compound data types can be formed with _______________
8. Input and Output instructions are done using the instructions _______________ and _______________
9. The _______________ of an algorithm is the amount of memory it needs to run to completion.
10. The _______________ of an algorithm is the amount of time it needs to run to completion.
11. The time T(P) taken by a program P is the sum of the _______________ and _______________
12. The function _______________ if there exist positive constants c and n_0 such that $f(n) \leq c*g(n)$ for all n, $n \geq n_0$.
13. The function _______________ if there exist positive constants c and n_0 such that $f(n) \geq c*g(n)$ for all n, $n \geq n_0$.
14. _______________notation provides an asymptotic lower bound on a function.
15. We use _________________to denote an upper bound that is NOT asymptotically tight.

16. Big oh provides an _____________upper bound on a function.
17. $\sum(1 \le k \le n)[O(n)]$ where O(n) stands for order n is:__________

Review Questions

1. (a) Compare Big-oh notation and Little-oh notation. Illustrate with an example.

   (b) Find Big-oh notation and Little-oh notation for $f(n) = 7n^3 + 50n^2 + 200$.

   R09 April-May 2012

2. Solve the following recurrence relations and give a bound for each of them.

   (a) $T(n) = 2\,T(n/3) + 1$ (b) $T(n) = 5\,T(n/4) + n$

   (c) $T(n) = 9\,T(n/3) + n2$ (d) $T(n) = 49\,T(n/25) + n^{3/2} \log n$

   (e) $T(n) = T(n-1) + nc$, where $c >= 1$, a constant.

   R09 set no 4 December-January, 2011-2012

3. (a) Explain the asymptotic notations used in algorithm analysis.

   (b) What is big "oh" notation? Show that if $f(n) = a_m n^m + .. + a_1 n + a_0$ then $f(n) = O(n^m)$.

   R09 June-2014

4. Solve the following recurrence relations and give a Ω bound for each of them:

   (a) $T(n) = 7\,T(n/7) + n$

   (b) $T(n) = 8\,T(n/2) + n3$

   (c) $T(n) = 8\,T(n-1) + 2$

   (d) $T(n) = T(n-1) + cn$, where $c > 1$, a constant.

   R09 set no 1 December-January, 2011-2012

5. (a) Write a recursive algorithm that converts a string of numerals to an integer, for example "34567" to 34567.

   (b) Write a recursive algorithm that calculates and returns the length of a list.

   R09 set no 3 December-January, 2011-2012

6. Define time complexity, Describe different asymptotic notations used to represent the time complexities with suitable examples.

   R09 May 2013

7. (a) Write an algorithm to find the largest Element in an array of n elements. Find its time complexity.

   (b) Explain about amortized analysis and probabilistic analysis.

   R09 November-December 2012

8. The Fibonacci numbers are defined as $f_0 = 0$ and $f_1 = 1$ and $f_i = f_i - 1 + f_i - 2$ for $i > 1$. Write both recursive and iterative algorithm to compute f_i. Also find their time complexities using step count method?

R09 December 2011

9. (a) Present an algorithm that searches for the element x in unsorted array a[1:n]. If x occurs, then return a position in the array; else return zero. Evaluate its time complexity.

   (b) Obtain a non-deterministic algorithm of complexity O(n) to determine whether there is a subset of n numbers ai, $1 \leq i \leq n$, that sums to n.

R09 December 2011

10. Define Time complexity. Explain the following Asymptotic notations:

   (a) Big Oh notation (b) Theta notation

   (c) Little oh notation (d) Amortized analysis.

R09 November/December-2013

11. (a) Define the terms Time and Space complexities. Explain about space analysis of an algorithm with example.

   (b) Explain various approaches to time complexity.

12. (a) Define the terms "Time complexity" and "Space complexity" of algorithms. Give a notation for expressing such a complexity and explain the features of such a notation.

   (b) Explain in detail about Little oh notation.

13. (a) Define Omega notation. Explain the terms involved in it. Give an example.

   (b) Explain in detail about Amortized analysis.

14. (a) For the Travelling sales person algorithm show that the time complexity is $0(n^2 2^n)$ and space complexity is $O(n2^n)$.

   (b) Write an algorithm of matrix chain multiplication.