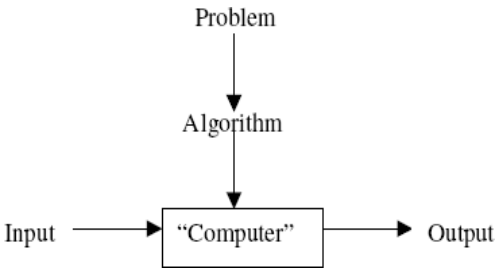
UNIT –I

Explain about algorithm with suitable example (Notion of algorithm).

An algorithm is a sequence of unambiguous instructions for solving a computational problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.



Algorithms – Computing the Greatest Common Divisor of Two Integers(gcd(m, n): the largest integer that divides both m and n.)

- **Euclid's algorithm:** gcd(m, n) = gcd(n, m mod n)

Step1: If n = 0, return the value of m as the answer and stop; otherwise, proceed to Step 2.

Step2: Divide m by n and assign the value of the remainder to r.

Step 3: Assign the value of n to m and the value of r to n. Go to Step 1.

Algorithm *Euclid(m, n)*

```
//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
while n ≠ 0 do
        r ← m mod n
        m ← n
        n ← r
return m
```

About This algorithm

- Finiteness: how do we know that Euclid's algorithm actually comes to a stop?
- Definiteness: nonambiguity
- Effectiveness: effectively computable.

- **Consecutive Integer Algorithm**

Step1: Assign the value of min{m, n} to t.

Step2: Divide m by t. If the remainder of this division is 0, go to Step3;otherwise, go to Step 4.

Step3: Divide n by t. If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, proceed to Step4.

Step4: Decrease the value of t by 1. Go to Step2.

About This algorithm

- Finiteness
- Definiteness
- Effectiveness

Middle-school procedure

Step1: Find the prime factors of m.

Step2: Find the prime factors of n.

Step3: Identify all the common factors in the two prime expansions found in Step1 and Step2. (If p is a common factor occurring Pm and Pn times in m and n, respectively, it should be repeated in min{Pm, Pn} times.)

Step4: Compute the product of all the common factors and return it as the gcd of the numbers given.

Explain the various Asymptotic Notations used in algorithm design? Or Discuss the properties of asymptotic notations. (Or) Explain the basic efficiency classes with notations.

Asymptotic notation is a notation, which is used to take meaningful statement about the efficiency of a program. The efficiency analysis framework concentrates on the order of growth of an algorithm‘s basic operation count as the principal indicator of the algorithm‘s efficiency.

To compare and rank such orders of growth, computer scientists use three notations, they are:

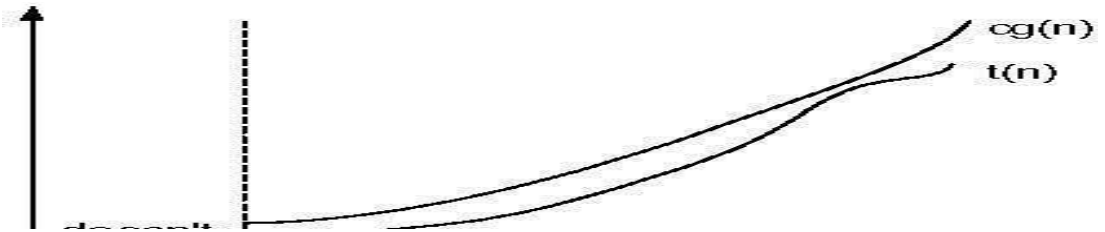
O - Big oh notation, Ω - Big omega notation, Θ - Big theta notation

Let t[n] and g[n] can be any nonnegative functions defined on the set of natural numbers. The algorithm‘s running time t[n] usually indicated by its basic operation count C[n], and g[n], some simple function to compare with the count.

There are 5 basic asymptotic notations used in the algorithm design.

- Big Oh: A function t[n] is said to be in O[g[n]], denoted by t[n] ε O[g[n]], if t[n] is bounded above by some constant multiple of g[n] for all large n, i.e., if there exists some positive constant c and some non-negative integer n0 such that T [n] <=cg [n] for all n >= n0
- Big Omega: A function t[n] is said to be in Ω [g[n]], denoted by t[n] ε Ω [g[n]], if t[n] is bounded below by some constant multiple of g[n] for all large n, i.e., if there exists some positive constant c and some non-negative integer n0 such that T [n] >=cg [n] for all n >=n0
- Big Theta: A function t[n] is said to be in θ [g[n]], denoted by t[n] ε θ [g[n]], if t[n] is bounded

  both above & below by some constant multiple of g[n] for all large n, i.e., if there exists some positive constants c1 & c2 and some nonnegative integer n0 such that c_2g [n] <= t [n] <= c_1g [n] for all n >= n0
- Little oh: The function f[n] = 0[g[n]] iff Lim f[n] = 0 n - a g[n]
- Little Omega. :The function f[n] = ω [g[n]] iff Lim f[n] = 0 n - a g[n] **t[n] □ O[g[n]] iff t[n] <=cg[n] for n > *n*0**



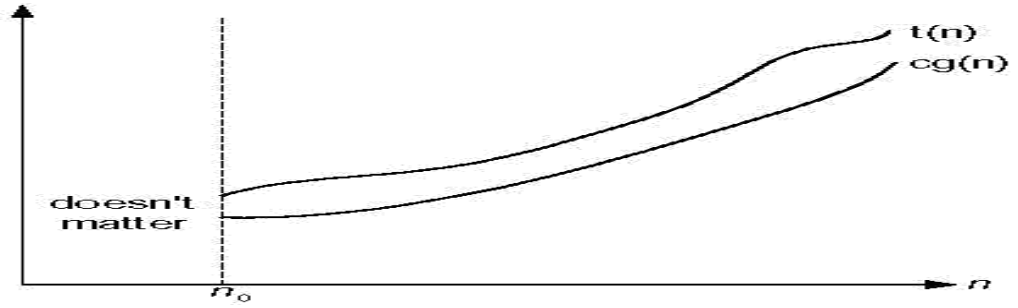


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

t[n] □ Ω[g[n]] iff t[n] >=cg[n] for n > *n*0

t[n]□ Θ[g[n]] iff t[n]□O[g[n]] and □Ω[g[n]]

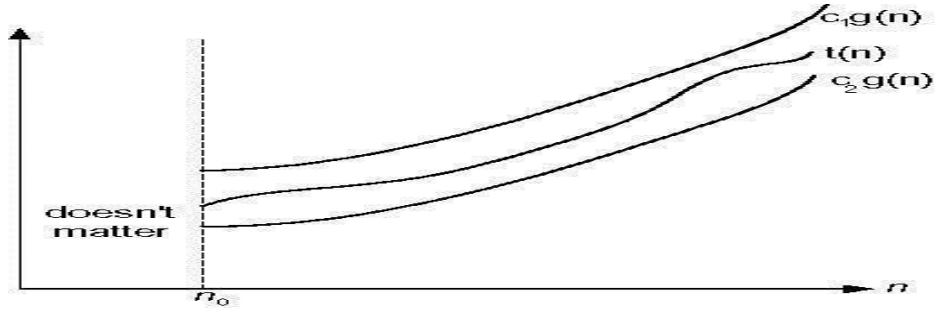


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Informal Definitions: Big O, Ω, Θ

Some properties of asymptotic order of growth

- *f*[*n*] □ O[*f*[*n*]]
- *f*[*n*] □ O[*g*[*n*]] iff *g*[*n*] □□[*f*[n]]

□□If *f* [*n*] □ O[*g* [*n*]] and □ O[*h*[*n*]] , the n *f*[*n]* □ O[*h*[*n*]]

Note similarity with $a \leq b$

- If $f1[n] \in O[g1[n]]$ and $f2[n] \in O[g2[n]]$, then $f1[n] + f2[n] \in O[\max\{g1[n], g2[n]\}]$

Basic Efficiency classes:

1	constant	Best case
log *n*	logarithmic	Divide ignore part
n	linear	Examine each
n log *n*	*n*-log-*n* or linear logarithmic	Divide use all parts
n^2	quadratic	Nested loops
n^3	cubic	Nested loops
2^n	exponential	All subsets
n!	factorial	All permutations

Explain recursive and non-recursive algorithms with example. (Or)

With an example, explain how recurrence equations are solved.

Mathematical Analysis of Recursive Algorithms

General Plan for Analysis

- Decide on a parameter indicating an input‘s size.
- Identify the algorithm‘s basic operation.
- Check whether the number of times the basic op. is executed may vary on different inputs of the same size. [If it may, the worst, average, and best cases must be investigated separately.]
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.
- Solve the recurrence by backward substitutions or another method.

EXAMPLE Compute the factorial function *F[n]* = *n*! for an arbitrary nonnegative integer *n* **ALGORITHM** *F[n]*

//Computes *n*! recursively
//Input: A nonnegative integer *n*
//Output: The value of *n*!
if *n* = 0 **return** 1
else return *F[n* − 1*]* * *n*
Thus, we succeeded in setting up the recurrence relation and initial condition for the algorithm′s number of multiplications *M(n)*:

M(n) = *M(n* − 1*)* + 1 for *n* > 0,
*M(*0*)* = 0.
From the several techniques available for solving recurrence relations, we use what can be called the ***method of backward substitutions***. The method′s idea (and the reason for the name) is immediately clear from the way it applies to solving our particular recurrence:

M(n) = *M(n* − 1*)* + 1 substitute *M(n* − 1*)* = *M(n* − 2*)* + 1
= [*M(n* − 2*)* + 1]+ 1= *M(n* − 2*)* + 2 substitute *M(n* − 2*)* = *M(n* − 3*)* + 1
= [*M(n* − 3*)* + 1]+ 2 = *M(n* − 3*)* + 3.

Since it is specified for *n* = 0, we have to substitute *i* = *n* in the pattern′s formula to get the ultimate result of our backward substitutions:

M(n) = *M(n* − 1*)* + 1= . . . = *M(n* − *i)* + *i* = . . . = *M(n* − *n)* + *n* = *n.*

EXAMPLE 2: consider educational workhorse of recursive algorithms: the Tower of Hanoi puzzle. We have n disks of different sizes that can slide onto any of three pegs. Consider A (source), B (auxiliary), and C (Destination). Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary.

ALGORITHM TOH(n, A, C, B)

//Move disks from source to destination recursively
//Input: n disks and 3 pegs A, B, and C
//Output: Disks moved to destination as in the source order.
if n=1
Move disk from A to C
else
Move top n-1 disks from A to B using C
TOH(n - 1, A, B, C)
Move top n-1 disks from B to C using A
TOH(n - 1, B, C, A)

Let us apply the general plan outlined above to the Tower of Hanoi problem.
The number of disks n is the obvious choice for the input′s size indicator, and so is moving one disk as the algorithm′s basic operation. Clearly, the number of moves $M(n)$ depends on n only, and we get the following recurrence equation for it:

$M(n) = M(n - 1) + 1 + M(n - 1)$ for $n > 1$.

With the obvious initial condition $M(1) = 1$, we have the following recurrence relation for the number of moves $M(n)$:

$M(n) = 2M(n - 1) + 1$ for $n > 1$, **(2.3)**

$M(1) = 1$.

We solve this recurrence by the same method of backward substitutions:

$M(n) = 2M(n - 1) + 1$ sub. $M(n - 1) = 2M(n - 2) + 1$

$= 2[2M(n - 2) + 1] + 1 = 22M(n - 2) + 2 + 1$ sub. $M(n - 2) = 2M(n - 3) + 1$

$= 22[2M(n - 3) + 1] + 2 + 1 = 23M(n - 3) + 22 + 2 + 1.$

The pattern of the first three sums on the left suggests that the next one will be $24M(n - 4) + 23 + 22 + 2 + 1$, and generally, after i substitutions, we get

$M(n) = 2iM(n - i) + 2i - 1 + 2i - 2 + \ldots + 2 + 1 = 2iM(n - i) + 2i - 1.$

Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$, we get the following formula for the solution to recurrence (2.3):

$M(n) = 2n - 1M(n - (n - 1)) + 2n - 1 - 1$

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

Mathematical Analysis of Non-Recursive Algorithms

General Plan for Analysis

Decide on parameter n indicating *input size*

- Identify algorithm‘s *basic operation*
- Determine *worst*, *average*, and *best* cases for input of size n
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules [see Appendix A]

EXAMPLE Consider the problem of finding the value of the largest element in a list of n numbers **ALGORITHM** MaxElement[A[0..n−1]]

```
//Determines the value of the largest element in a
given array //Input: An array A[0..n−1] of real
numbers
//Output: The value of the largest element in A
maxval ←A[0]
fori ←1 to n−1do
if A[i]> maxval
maxval←A[i]
```

return maxval

Let us denote *C(n)* the number of times this comparison is executed and try to find a formula expressing it as a function of size *n*. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop′s variable *i* within the bounds 1 and $n - 1$, inclusive. Therefore, we get the following sum for *C(n)*:

This is an easy sum to compute because it is nothing other than 1 repeated $n - 1$ times. Thus, t(n)=O(n)

EXAMPLE 2 Given two n×n matrices A and B, find the time efficiency of the definition-based algorithm for computing their produc tC =AB. By definition,C is an n×n matrix whose elements are computed as the scalar [dot] products of the rows of matrix A and the columns of matrixB: whereC[i, j]=A[i, 0]B[0, j]+...+A[i, k]B[k, j]+...+A[i, n−1]B[n−1, j] for every pair of indices 0 ≤i, j ≤n−1.

ALGORITHM MatrixMultiplication[A[0..n−1, 0..n−1], B[0..n−1, 0..n−1]]
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two n×n matrices A andB
//Output: MatrixC =AB
fori ←0 to n−1do
for j ←0 to n−1do
C[i, j]←0.0
for k←0 to n−1do
C[i, j]←C[i, j]+A[i, k]∗B[k, j]
return C

To measure an input‘s size by matrix order n. There are two arithmetical operations in the innermost loop here—multiplication and addition—that, in principle, can compete for designation as the algorithm‘s basic operation.

Obviously, there is just one multiplication executed on each repetition of the algorithm's innermost loop, which is governed by the variable k ranging from the lower bound 0 to the upper bound $n-1$. Therefore, the number of multiplications made for every pair of specific values of variables i and j is

$$\sum_{k=0}^{n-1} 1,$$

and the total number of multiplications $M(n)$ is expressed by the following triple sum:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1.$$

Now, we can compute this sum by using formula (S1) and rule (R1) given above. Starting with the innermost sum $\sum_{k=0}^{n-1} 1$, which is equal to n (why?), we get

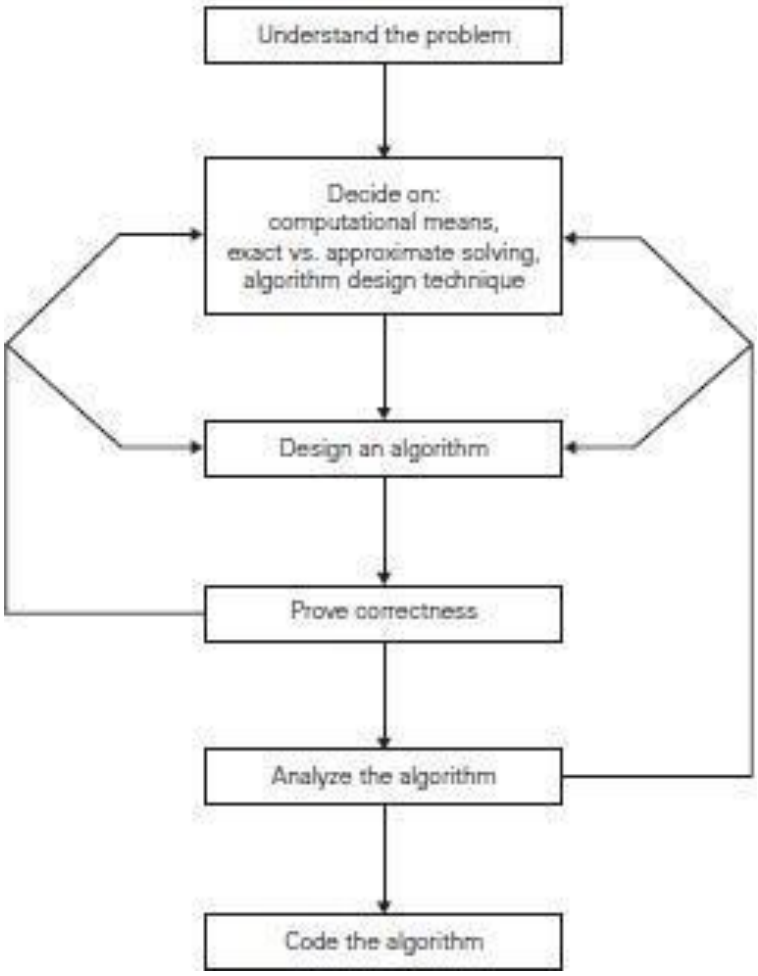
$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

What are the fundamental steps to solve an algorithm? Explain. Or Describe in detail about the steps in analyzing and coding an algorithm.

An algorithm is a sequence of unambiguous instructions for solving problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time. Algorithmic steps are

- Understand the problem
- Decision making
- Design an algorithm
- Proving correctness of an algorithm
- Analyze the algorithm
- Coding and implementation of an algorithm

Figure : Algorithm design and analysis process



3. **Understand the problem**
    a. Read the description carefully to understand the problem completely
    b. Identfiy the problem types and use existing algorithm to find solution
    c. Input (instance) to the problem and range of the input gets fixed.
4. **Decision making**
    a. <u>Ascertaining the capabilities of computational device</u>
        i. In Ram instructions are executed one after another, accordingly algorithms designed to be executed on such machines are executed sequential algorithms
        ii. In some computers operations are executed concurrently in parallel.
        iii. Choice of computational devices like processor and memory is mainly based on space and time efficiency.
    b. <u>Choosing between exact versus approximate problem solving</u>

i. An algorithm used to solve the problem exactly and produce correct result is called exact algorithm
ii. If the problem is to so complex and not able to get exact solution then it is called approximation algorithm.

c. Algorithm design strategies
   i. Algorithms + data structures = programs, though algorithms and data structures are independent then they combined to produce programs.
   ii. Implementation of an algorithm is possible with the help of algorithms and data structures.
   iii. Algorithm design strategy techniques are brute force, dynamic programming, greedy technique, divide and conquer and so on.

iv. Methods for specifying an algorithm

1. **Natural language**: It is very simple and easy to specify an algorithm using natural language. Example // addition of 2 nos
   Read a
   Read b
   Add
   c=a+b
   Store and display the result in c
2. **Flow chart** – Flowchart is a diagrammatic and graphical representation of an algorithm. It is a method of expressing an algorithm by a collection of connected graphical shapes containing description of the algorithm‘s steps.
3. **Pseudo code -** It is a mixture of natural language and programming language constructs. It is usually more precise than natural language.
   Example // sum of 2 nos
   //input a and b
   // output c
   c← a+ b

d. Proving an algorithm’s correctness

i. Once an algorithm has been specified then its correctness must be proved.
ii. An algorithm must yields a required result for every legitimate input in a finite amount of time.
iii. For example, the correctness of Euclid‘s algorithm for computing the greatest common divisor stems from the correctness of the equality gcd(m, n) = gcd(n, m mod n).
iv. A common technique for proving correctness is to use mathematical induction because an algorithm‘s iterations provide a natural sequence of steps needed for such proofs.
v. The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. The error produced by the algorithm should not exceed a predefined limit.

e. Analyzing an algorithm

For an algorithm the most important is algorithm efficiency .There are two types of algorithm efficiencies are

- *Time efficiency:* indicates how fast the algorithm runs
- *Space efficiency:* indicates how much extra memory the algorithm needs

So the efficiency of an algorithm through analysis is based on both time and space efficiency. There are some factors to analyze an algorithm are:

- Simplicity of an algorithm
- Generality of an algorithm
- Time efficiency of an algorithm
- Space efficiency of an

algorithm **f. Coding an algorithm**

i. The coding / implementation of an algorithm is done by a suitable programming language like C, C++, JAVA.

ii. The transition from an algorithm to a program can be done either incorrectly or very inefficiently. Implementing an algorithm correctly is necessary. The Algorithm power should not reduce by inefficient implementation.

iii. Standard tricks like computing a loop's invariant outside the loop, collecting common sub expressions, replacing expensive operations by cheap ones, selection of programming language and so on should be known to the programmer.

Discuss Fundamentals of the analysis of algorithm efficiency elaborately.

Analysis of algorithm is the process of investigation of an algorithm's efficiency with respect to two resources: running time and memory space.

- The simplicity and generality measures of an algorithm estimate the efficiency
- The speed and memory are the efficiency considerations of

modern computers. That there are two kinds of efficiency: time efficiency and space efficiency.

- Time efficiency, also called time complexity, indicates how fast an algorithm in question runs.
- Space efficiency, also called space complexity, refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output.

1. Measuring an input‘s size
    a. The efficiency measure of an algorithm is directly proportional to the input size or range.
    b. The input given may be a square or a non-square matrix.
    c. Some algorithms require more than one parameter to indicate the size of their inputs.
2. Units for measuring time
    a. We can simply use some standard unit of time measurement-a second, a millisecond, and so on-to measure the running time of a program implementing the algorithm.
    b. There are obvious drawbacks to such an approach. They are
        - Dependence on the speed of a particular computer
        - Dependence on the quality of a program implementing the algorithm
        - The compiler used in generating the machine code
        - The difficulty of clocking the actual running time of the program.
    c. Since we are in need to measure algorithm efficiency, we should have a metric that does not depend on these extraneous factors.
    d. One possible approach is to count the number of times each of the algorithm's operations is executed. This approach is both difficult and unnecessary.
    e. The main objective is to identify the most important operation of the algorithm, called the basic operation, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.
3. Efficiency classes
    a. It is reasonable to measure an algorithm's efficiency as a function of a parameter

indicating the size of the algorithm's input.

b. But there are many algorithms for which running time depends not only on an input size but also on the specifics of a particular input.

4. **Example, sequential search**. This is a straightforward algorithm that searches for a given item (some search key K) in a list of n elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted.

//Searches for a given value in a given array by
sequential search //Input: An array A[0..n -1] and a
search key K
//Output: Returns the index of the first element of A that matches K
//or -1 ifthere are no matching
elements i←0
while i < n and A[i] ≠
K do i←i+1
if i < n return i
else return -1

Worst case
efficiency

- The worst-case efficiency of an algorithm is its efficiency for the worst-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the longest among all possible inputs of that size.
- In the worst case, when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size n: **Cworst (n) = n.**

Best case Efficiency

- The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the fastest among
all possible inputs of that size.
- First, determine the kind of inputs for which the count C (n) will be the smallest among all possible inputs of size n. (Note that the best case does not mean the smallest input; it means the input of size n for which the algorithm runs the fastest.)
- Then ascertain the value of C (n) on these most convenient inputs. Example- for sequential search, best-case inputs will be lists of size n with their first elements equal to a search key; accordingly, **Cbest(n) = 1.**

- The average number of key comparisions Cavg(n) can be computed as follows, o let us consider again sequential search. The standard assumptions are,

In the case of a successful search, the probability of the first match occurring in the ith position of the list is pin for every i, and the number

of comparisons made by the algorithm in such a situation is obviously i.

Cavg(n) =(n+ 1)/2

5. Orders of growth

Big oh , Big omega and Big Theta notations described above 2 Question.

Discuss important problem types that you face during Algorithm Analysis.

- sorting
  - Rearrange the items of a given list in ascending order.
  - Input: A sequence of n numbers <a1, a2, …, an>
  - Output: A reordering <a´1, a´2, …, a´n> of the input sequence such that a´1≤a´2 ≤… ≤a´n.
  - A specially chosen piece of information used to guide sorting. I.e., sort student records by names.

    Examples of sorting algorithms
  - Selection sort
  - Bubble sort
  - Insertion sort
  - Merge sort
  - Heap sort …

    Evaluate sorting algorithm complexity: the number of key comparisons.
  - Two properties
  - Stability: A sorting algorithm is called stable if it preserves the relative order of any two equal elements in its input.
  - In place: A sorting algorithm is in place if it does not require extra memory, except, possibly for a few memory units.
- searching
  - Find a given value, called a search key, in a given set.
  - Examples of searching algorithms
    - Sequential searching
    - Binary searching…
- string processing
  - A string is a sequence of characters from an alphabet.
  - Text strings: letters, numbers, and special characters.
  - String matching: searching for a given word/pattern in a text.
- graph problems
  - Informal definition
    - A graph is a collection of points called vertices, some of which are connected by line segments called edges.
  - Modeling real-life problems
  - Modeling WWW
  - communication networks
  - Project scheduling …

    Examples of graph algorithms
  - Graph traversal algorithms
  - Shortest-path algorithms
  - Topological sorting
- combinatorial problems
- geometric problems,