CS 6402

DESIGN AND ANALYSIS OF ALGORITHMS

QUESTION BANK

UNIT I INTRODUCTION

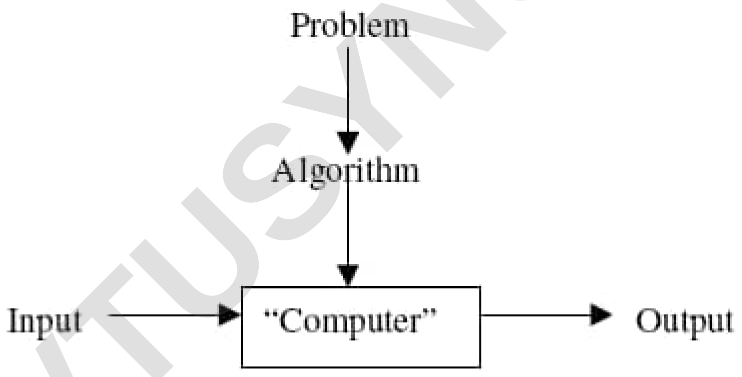
2 marks

1. Why is the need of studying algorithms?
From a practical standpoint, a standard set of algorithms from different areas of computing must be known, in addition to be able to design them and analyze their efficiencies. From a theoretical standpoint the study of algorithms is the cornerstone of computer science.

2. What is algorithmic?
The study of algorithms is called algorithmic. It is more than a branch of computer science. It is the core of computer science and is said to be relevant to most of science, business and technology.

3. What is an algorithm?
An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in finite amount of time.
An algorithm is step by step procedure to solve a problem.

4. Give the diagram representation of Notion of algorithm.



5. What is the formula used in Euclid's algorithm for finding the greatest common divisor of two numbers?
Euclid's algorithm is based on repeatedly applying the equality
Gcd(m,n)=gcd(n,m mod n) until m mod n is equal to 0, since gcd(m,0)=m.

6. What are the three different algorithms used to find the gcd of two numbers?
The three algorithms used to find the gcd of two numbers are
- Euclid's algorithm
- Consecutive integer checking algorithm
- Middle school procedure

7. What are the fundamental steps involved in algorithmic problem solving?
The fundamental steps are
- Understanding the problem
- Ascertain the capabilities of computational device

- Choose between exact and approximate problem solving
- Decide on appropriate data structures
- Algorithm design techniques
- Methods for specifying the algorithm
- Proving an algorithms correctness
- Analyzing an algorithm
- Coding an algorithm

8. What is an algorithm design technique?
An algorithm design technique is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

9. What is pseudocode?
A pseudocode is a mixture of a natural language and programming language constructs to specify an algorithm. A pseudocode is more precisethan a natural language and its usage often yields more concise algorithm descriptions.

10. What are the types of algorithm efficiencies?
The two types of algorithm efficiencies are

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

11. Mention some of the important problem types?
Some of the important problem types are as follows

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

12. What are the classical geometric problems?
The two classic geometric problems are

- *The closest pair problem:* given n points in a plane find the closest pair among them
- *The convex hull problem:* find the smallest convex polygon that would include all the points of a given set.

13. What are the steps involved in the analysis framework?
The various steps are as follows

- Measuring the input’s size
- Units for measuring running time
- Orders of growth
- Worst case, best case and average case efficiencies

14. What is the basic operation of an algorithm and how is it identified?

- The most important operation of the algorithm is called the basic operation of the algorithm, the operation that contributes the most to the total running time.
- It can be identified easily because it is usually the most time consuming operation in the algorithms innermost loop.

15. What is the running time of a program implementing the algorithm?
The running time T(n) is given by the following formula

$$T(n) \approx c_{op}C(n)$$

cop is the time of execution of an algorithm’s basic operation on a particular computer and
C(n) is the number of times this operation needs to be executed for the particular algorithm.

16. What are exponential growth functions?

The functions 2n and n! are exponential growth functions, because these two functions grow so fast that their values become astronomically large even for rather smaller values of n.

17. What is 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.

18. What is 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 for which the algorithm runs the fastest among all possible inputs of that size.

19. What is average case efficiency?

The average case efficiency of an algorithm is its efficiency for an average case input of size n. It provides information about an algorithm behavior on a “typical” or “random” input.

20. What is amortized efficiency?

In some situations a single operation can be expensive, but the total time for the entire sequence of n such operations is always significantly better that the worst case efficiency of that single operation multiplied by n. this is called amortized efficiency.

21. Define O-notation?

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 n_0 such that

$$T(n) <= cg(n) \text{ for all } n >= n_0$$

22. Define Ω-notation?

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 n_0 such that

$$T(n) >= cg(n) \text{ for all } n >= n_0$$

23. Define θ-notation?

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

$$c_2 g(n) <= t(n) <= c_1 g(n) \text{ for all } n >= n0$$

24. Mention the useful property, which can be applied to the asymptotic notations and its use?

If $t_1(n)$ ε $O(g_1(n))$ and $t_2(n)$ ε $O(g_2(n))$ then $t_1(n)+t_2(n)$ ε max $\{g_1(n),g_2(n)\}$ this property is also true for Ω and θ notations. This property will be useful in analyzing algorithms that comprise of two consecutive executable parts.

25. What are the basic asymptotic efficiency classes?

The various basic efficiency classes are

- Constant : 1
- Logarithmic : log n
- Linear : n
- N-log-n : nlog n
- Quadratic : n2
- Cubic : n3
- Exponential : 2n
- Factorial : n!

26. What is algorithm visualization?
Algorithm visualization is a way to study algorithms. It is defined as the use of images to convey some useful information about algorithms. That information can be a visual illustration of algorithm's operation, of its performance on different kinds of inputs, or of its execution speed versus that of other
algorithms for the same problem.

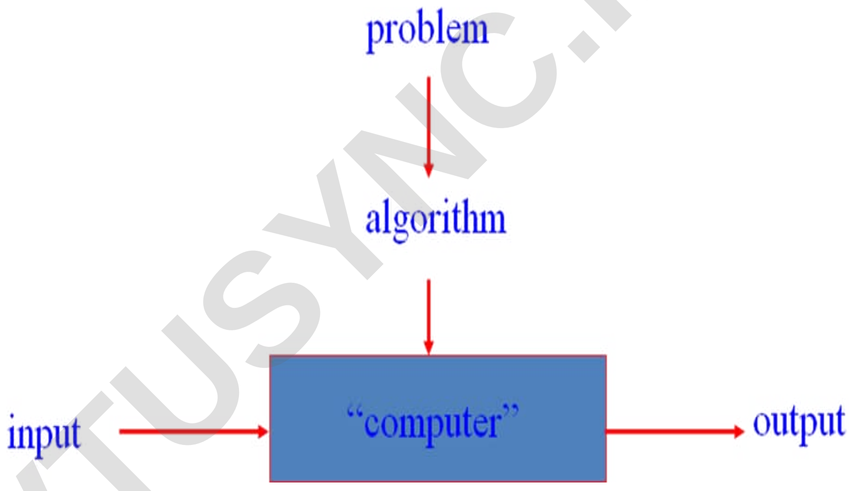
27. What are the two variations of algorithm visualization?

- The two principal variations of algorithm visualization" ._*Static algorithm visualization*: It shows the algorithm's progress
- through a series of still images ._*Dynamic algorithm visualization*: Algorithm animation shows a
- continuous movie like presentation of algorithms operations

28. What is order of growth?
Measuring the performance of an algorithm based on the input size n is called order of growth.

16 marks

1. **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.

2. Write short note on Fundamentals of Algorithmic Problem Solving

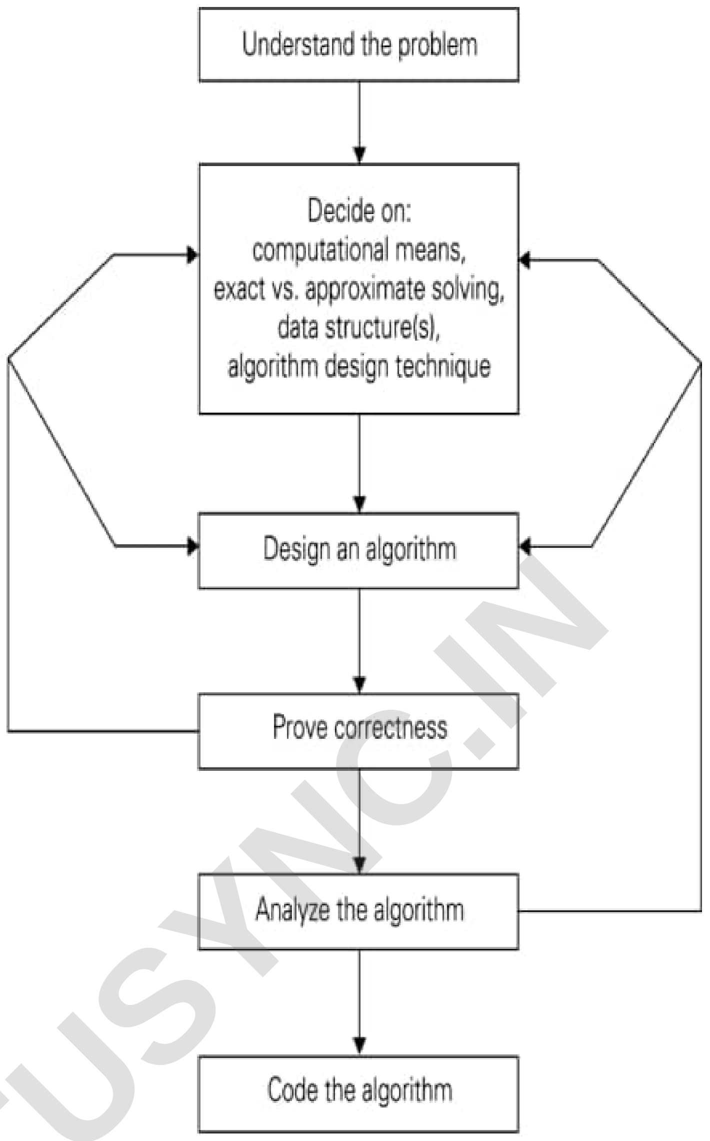
✓ Understanding the problem
  - Asking questions, do a few examples by hand, think about special cases, etc.

✓ Deciding on
  - Exact vs. approximate problem solving
  - Appropriate data structure

✓ Design an algorithm

✓ Proving correctness

✓ Analyzing an algorithm



Understand the problem
Decide on: computational means, exact vs. approximate solving, data structure(s), algorithm design technique
Design an algorithm
Prove correctness
Analyze the algorithm
Code the algorithm

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

✓ Coding an algorithm

3. 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
✓ Numerical problems

4. Discuss Fundamentals of the analysis of algorithm efficiency elaborately.

✓ Algorithm’s efficiency
✓ Three notations
✓ Analyze of efficiency of Mathematical Analysis of Recursive Algorithms
✓ Analyze of efficiency of Mathematical Analysis of non-Recursive Algorithms

Analysis of algorithms means to investigate an algorithm’s efficiency with respect to resources: running time and memory space.
- Time efficiency: how fast an algorithm runs.
- Space efficiency: the space an algorithm requires.

✓ Measuring an input’s size
✓ Measuring running time
✓ Orders of growth (of the algorithm’s efficiency function)
✓ Worst-base, best-case and average efficiency

■ **Measuring Input Sizes**
- Efficiency is defined as a function of input size.
- Input size depends on the problem.
- Example 1, what is the input size of the problem of sorting n numbers?
- Example 2, what is the input size of adding two n by n matrices?

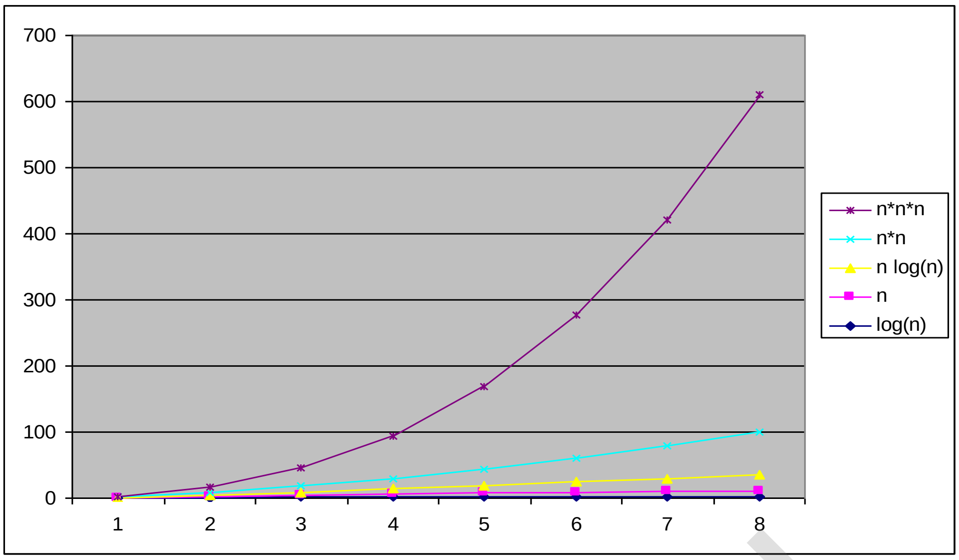
- **Units for Measuring Running Time**
  - Measure the running time using standard unit of time measurements, such as seconds, minutes?
  - Depends on the speed of the computer.
  - count the number of times each of an algorithm's operations is executed.
  - Difficult and unnecessary
  - count the number of times an algorithm's basic operation is executed.
  - Basic operation: the most important operation of the algorithm, the operation contributing the most to the total running time.
  - For example, the basic operation is usually the most time-consuming operation in the algorithm's innermost loop.
- **Orders of Growth**
  - consider only the leading term of a formula
  - Ignore the constant coefficient.
- **Worst-Case, Best-Case, and Average-Case Efficiency**
  - Algorithm efficiency depends on the input size n
  - For some algorithms efficiency depends on type of input.

    Example: Sequential Search
  - *Problem:* Given a list of n elements and a search key K, find an element equal to K, if any.
  - *Algorithm:* Scan the list and compare its successive elements with K until either a matching element is found (successful search) of the list is exhausted (unsuccessful search)

  Worst case Efficiency
  - Efficiency (# of times the basic operation will be executed) for the worst case input of size n.
  - The algorithm runs the longest among all possible inputs of size n.

  Best case
  - Efficiency (# of times the basic operation will be executed) for the best case input of size n.
  - The algorithm runs the fastest among all possible inputs of size n.

  Average case:
  - Efficiency (#of times the basic operation will be executed) for a typical/random input of size n.
  - NOT the average of worst and best case

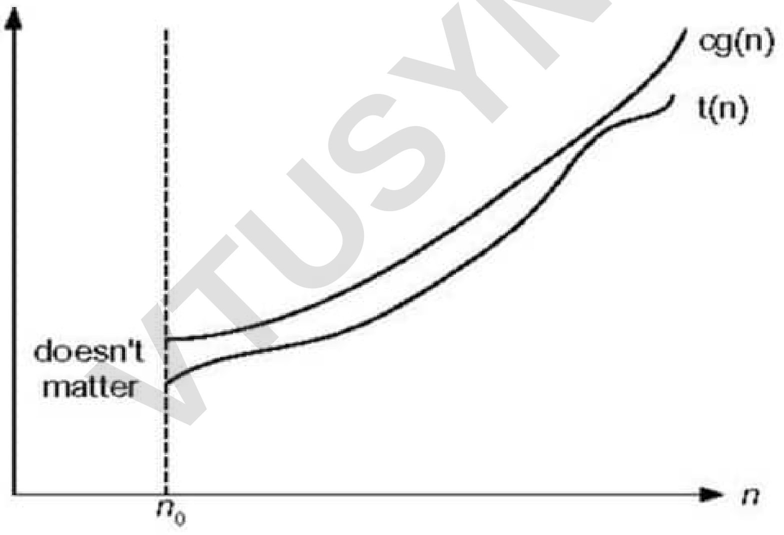


5. Explain Asymptotic Notations

Three notations used to compare orders of growth of an algorithm's basic operation count

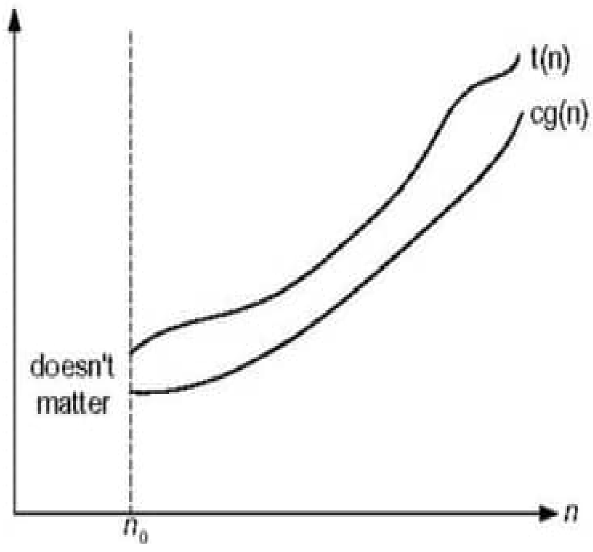
a. O(g(n)): class of functions f(n) that grow *no faster* than g(n)

A function *t(n)* is said to be in *O(g(n)),* denoted *t(n)* $\in$*O(g(n)),* if *t(n)* is bounded above by some constant multiple of *g(n)* for all large *n*, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that $t(n) \le cg(n)$ for all $n \ge n_0$



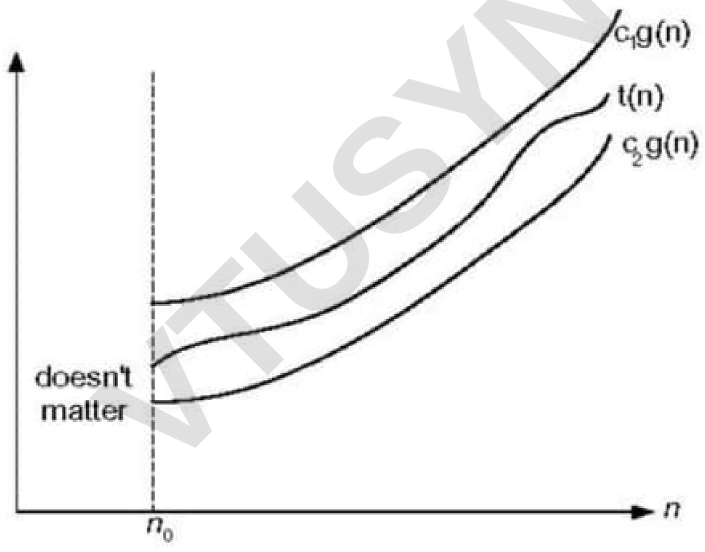
b. Ω(g(n)): class of functions f(n) that grow *at least as fast* as g(n)

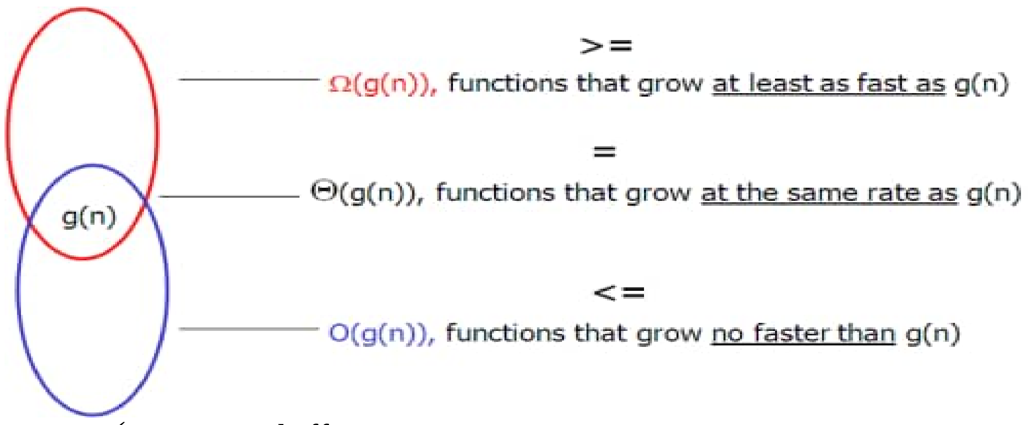
A function *t(n)* is said to be in *Ω(g(n)),* denoted *t(n)* $\in$ *Ω(g(n)),* if *t(n)* is bounded below by some constant multiple of *g(n)* for all large *n*, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that $t(n) \ge cg(n)$ for all $n \ge n_0$



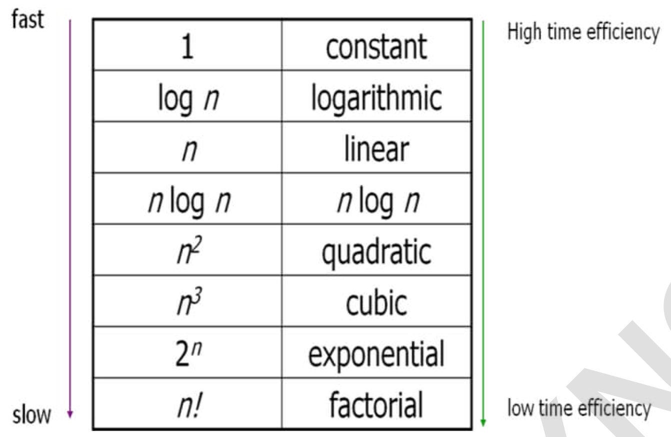
c. Θ (g(n)): class of functions f(n) that grow at same rate as g(n)

A function *t(n)* is said to be in Θ*(g(n))*, denoted *t(n)* ∈ Θ*(g(n))*, if *t(n)* is bounded both above and below by some positive constant multiples of *g(n)* for all large *n*, i.e., if there exist some positive constant c_1 and c_2 and some nonnegative integer n_0 such that $c_2\, g(n) \le t(n) \le c_1\, g(n)$ for all $n \ge n_0$




>=
Ω(g(n)), functions that grow at least as fast as g(n)
=
Θ(g(n)), functions that grow at the same rate as g(n)
g(n)
<=
O(g(n)), functions that grow no faster than g(n)

- ✓ Amortized efficiency


fast
slow
High time efficiency
low time efficiency

1	constant
log *n*	logarithmic
n	linear
n log *n*	*n* log *n*
n^2	quadratic
n^3	cubic
2^n	exponential
n!	factorial

6. List out the Steps in Mathematical Analysis of non recursive Algorithms.

- ✓ Steps in mathematical analysis of nonrecursive algorithms:
  - Decide on parameter n indicating input size
  - Identify algorithm’s basic operation
  - Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate worst, average, and best case efficiency separately.
  - Set up summation for C(n) reflecting the number of times the algorithm’s basic operation is executed.
- ✓ Example: Finding the largest element in a given array

```
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]
for i ← 1 to n-1 do
        if A[i] > maxval
                maxval ← A[i]
return maxval
```

7. List out the Steps in Mathematical Analysis of Recursive Algorithms.

- ✓ Decide on parameter *n* indicating *input size*
- ✓ Identify algorithm’s *basic operation*
- ✓ Determine *worst*, *average*, and *best* case for input of size *n*

- ✓ Set up a recurrence relation and initial condition(s) for $C(n)$-the number of times the basic operation will be executed for an input of size n (alternatively count recursive calls).
- ✓ Solve the recurrence or estimate the order of magnitude of the solution

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1) * (n-2)\dots 3 * 2 * 1 & \text{if } n > 0 \end{cases}$$

- ✓ Recursive definition

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * F(n-1) & \text{if } n > 0 \end{cases}$$

```
Algorithm  F(n)
    if n=0
            return 1                    //base case
    else
        return F (n -1) * n        //general case
```

Example Recursive evaluation of *n* ! (2)

- ✓ Two Recurrences

The one for the factorial function value: F(n)

$F(n) = F(n – 1) * n$ for every $n > 0$

$F(0) = 1$

The one for number of multiplications to compute n!, M(n)

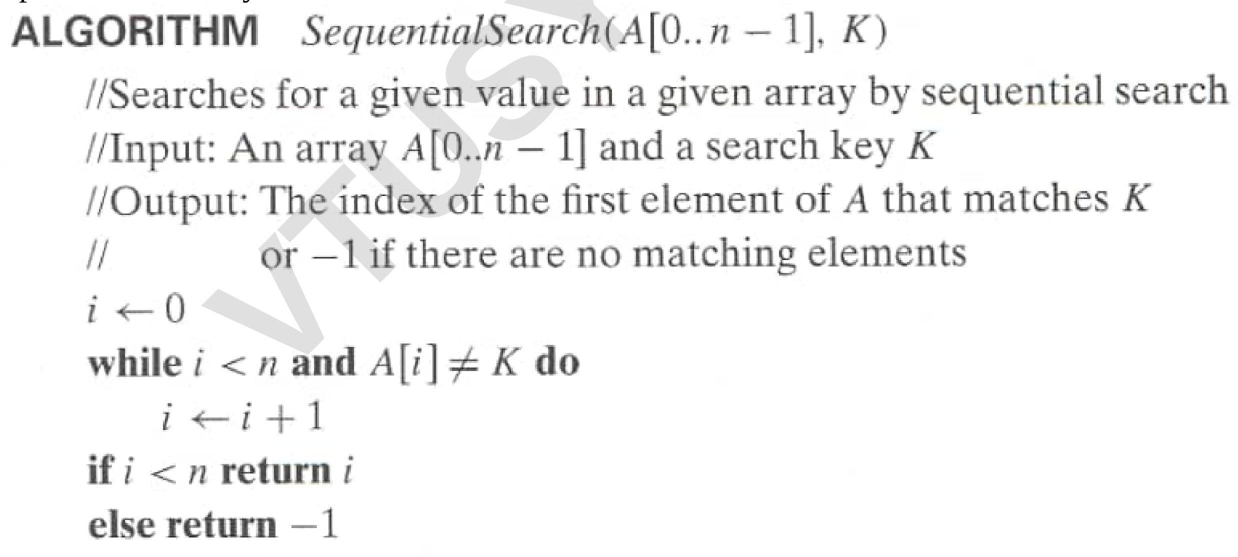
$M(n) = M(n – 1) + 1$ for every $n > 0$

$M(0) = 0$

$M(n) \in \Theta(n)$

8. **Explain in detail about linear search.**

Sequential Search searches for the key value in the given set of items sequentially and returns the position of the key value else returns -1.



```
ALGORITHM  SequentialSearch(A[0..n − 1], K)
    //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: The index of the first element of A that matches K
    //          or −1 if there 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
```

Analysis:

For sequential search, best-case inputs are lists of size n with their first elements equal to a search key; accordingly,

$$C_{bw}(n) = 1.$$

Average Case Analysis:

The standard assumptions are that

(a) the probability of a successful search is equal *top* (0 <=*p*<-=1) and
 (b) the probability of the first match occurring in the ith position of the list is the same for every i.
Under these assumptions- the average number of key comparisons $C_{avg}(n)$ *is found* as follows.

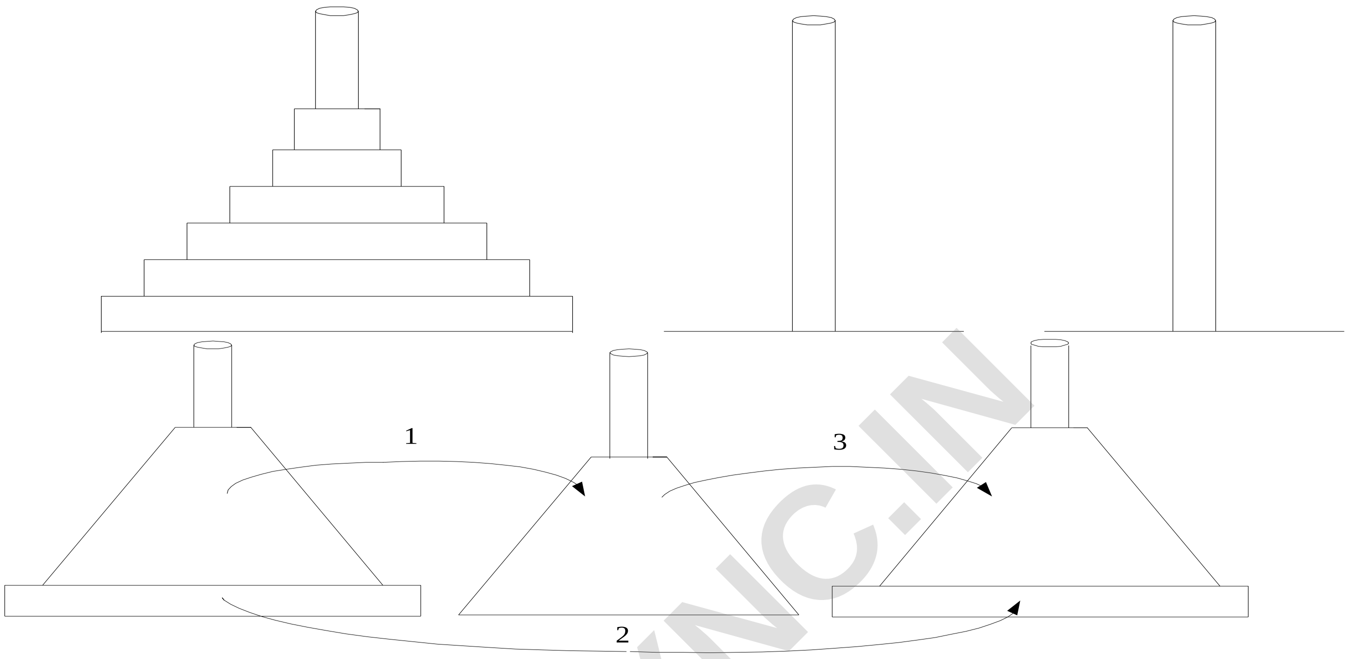
In the case of a successful search, the probability of the first match occurring in the i th position of the list is *p / n* for every i, and the number of comparisons made by the algorithm in such a situation is obviously *i.* In the case of an unsuccessful search, the number of comparisons is *n* with the probability of such a search being (1- *p).* Therefore,

$$\begin{aligned} C_{avg}(n) &= [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1 - p) \\ &= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1 - p) \\ &= \frac{p}{n}\frac{n(n+1)}{2} + n(1 - p) = \frac{p(n+1)}{2} + n(1 - p). \end{aligned}$$

For example, if *p* = 1 (i.e., the search must be successful), the average number of key comparisons made by sequential search is (n + 1) /2; i.e., the algorithm will inspect, on average, about half of the list's elements. If *p* = 0 (i.e., the search must be unsuccessful), the average number of key comparisons will be *n* because the algorithm will inspect all *n* elements on all such inputs.

9. Explain in detail about Tower of Hanoi.

In this puzzle, there are *n* disks of different sizes and three pegs. 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, if necessary. On1y one disk can be moved at a time, and it is forbidden to place a larger disk on top of a smaller one.



The general plan 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:

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

With the obvious initial condition M(1) = 1, the recurrence relation for the number of moves M(n) is:

$$M(n) = 2M(n-1) + 1 \text{ for } n > 1,\ M(1) = 1.$$

The total number of calls made by the Tower of Hanoi algorithm: ***n-1***

$$C(n) = \sum_{l=0}^{n-1} 2^l$$

$$= 2n-1$$