CS8451- DESIGN AND ANALYSIS OF ALGORITHMS

SYLLABUS

UNIT I INTRODUCTION 9

Notion of an Algorithm – Fundamentals of Algorithmic Problem Solving – Important Problem Types – Fundamentals of the Analysis of Algorithmic Efficiency –Asymptotic Notations and their properties. Analysis Framework – Empirical analysis - Mathematical analysis for Recursive and Non-recursive algorithms – Visualization

UNIT II FORCE AND DIVIDE-AND-CONQUER 9

Brute Force – Computing an – String Matching - Closest-Pair and Convex-Hull Problems - Exhaustive Search - Travelling Salesman Problem - Knapsack Problem - Assignment problem. Divide and Conquer Methodology – Binary Search – Merge sort – Quick sort – Heap Sort - Multiplication of Large Integers – Closest-Pair and Convex - Hull Problems.

UNIT III DYNAMIC PROGRAMMING AND GREEDY TECHNIQUE 9

Dynamic programming – Principle of optimality - Coin changing problem, Computing a Binomial Coefficient – Floyd‘s algorithm – Multi stage graph - Optimal Binary Search Trees – Knapsack Problem and Memory functions. Greedy Technique – Container loading problem - Prim‘s algorithm and Kruskal's Algorithm – 0/1 Knapsack problem, Optimal Merge pattern - Huffman Trees.

UNIT IV ITERATIVE IMPROVEMENT 9

The Simplex Method - The Maximum-Flow Problem – Maximum Matching in Bipartite Graphs, Stable marriage Problem.

UNIT V COPING WITH THE LIMITATIONS OF ALGORITHM POWER 9

Lower - Bound Arguments - P, NP NP- Complete and NP Hard Problems. Backtracking – n-Queen problem - Hamiltonian Circuit Problem – Subset Sum Problem. Branch and Bound – LIFO Search and FIFO search - Assignment problem – Knapsack Problem – Travelling Salesman Problem - Approximation Algorithms for NP-Hard Problems – Travelling Salesman problem – Knapsack problem.

TOTAL: 45 PERIODS

TEXT BOOKS:

1. Anany Levitin, ―Introduction to the Design and Analysis of Algorithms‖, Third Edition, Pearson Education, 2012.
2. Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, Computer Algorithms/ C++, Second Edition, Universities Press, 2007.

REFERENCES:

1. Thomas H.Cormen, Charles E.Leiserson, Ronald L. Rivest and Clifford Stein, ―Introduction to Algorithms‖, Third Edition, PHI Learning Private Limited, 2012.
2. Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, ―Data Structures and Algorithms‖, Pearson Education, Reprint 2006.
3. Harsh Bhasin, ―Algorithms Design and Analysis‖, Oxford university press, 2016.
4. S. Sridhar, ―Design and Analysis of Algorithms‖, Oxford university press, 2014.
5. http://nptel.ac.in/

UNIT I

PART A

1. **What is an algorithm? Or Define an algorithm. (Apr\May- 2017) Or Define algorithm with its properties.(April/May 2021)**
   - An algorithm is a finite set of instructions that, if followed, accomplishes a particular task.
   - In addition, all algorithms must satisfy the following criteria:
     - input
     - Output
     - Definiteness
     - Finiteness
     - Effectiveness.

2. **Define Program.**
   A program is the expression of an algorithm in a programming language.

3. **What is performance measurement?**
   Performance measurement is concerned with obtaining the space and the time requirements of a particular algorithm.

4. **Write the For LOOP general format.**
   The general form of a for Loop is

```
For variable : = value 1 to value 2
Step do
 {
        <statement 1>
        <statement n >
 }
```

5. **What is recursive algorithm?**
   - ✓ Recursive algorithm makes more than a single call to itself is known as recursive call.
   - ✓ An algorithm that calls itself is Direct recursive.
   - ✓ Algorithm A is said to be indeed recursive if it calls another algorithm,which in turn calls A

6. **What is space complexity?**
   The space complexity of an algorithm is the amount of memory it needs to run to completion.

7. **What is time complexity?**
   The time complexity of an algorithm is the amount of time it needs to run to completion.

8. **Give the two major phases of performance evaluation.**
   Performance evaluation can be loosely divided into two major phases:
   - a prior estimates (performance analysis)
   - a posterior testing (performance measurement)

9. **Define input size.**
   The input size of any instance of a problem is defined to be the number of elements needed to describe that instance.

10. **Define best-case step count.**
    The best-case step count is the minimum number of steps that can be executed for the given parameters.

11. Define worst-case step count.

The worst-case step count is the maximum number of steps that can be executed for the given parameters.

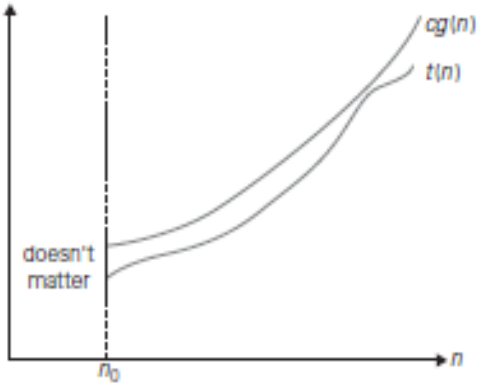
12. Define average step count.

The average step count is the average number of steps executed an instances with the given parameters.

13. Define the asymptotic notation "Big oh" (0)

A function t(n) is said to be in O(g(n)) **(t(n) Є O(g(n))),** if t(n) is bounded above by constant multiple of g(n) for all values of n, and if there exist a positive constant c and non negative integer n0 such that

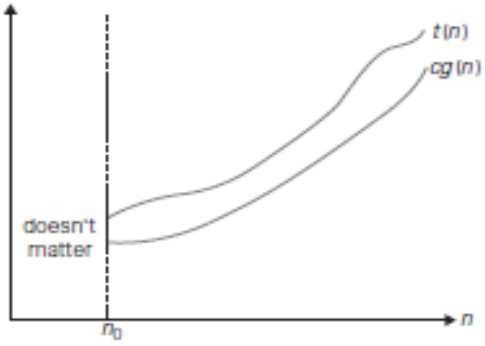
t(n) ≤ c*g(n) for all n ≥ n0.



1 Big-oh notation: $t(n) \in O(g(n))$.

14. Define the asymptotic notation "Omega" (Ω). NOV/DEC 2021

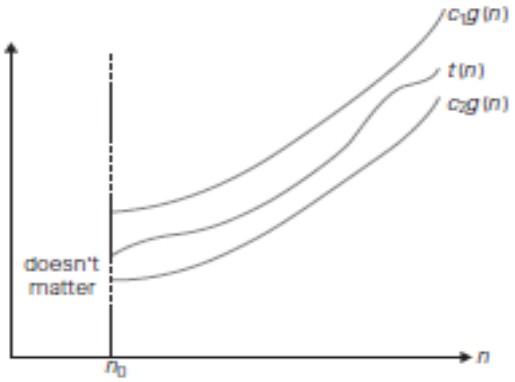
A function t(n) is said to be in Ω(g(n)) **(t(n) Є Ω(g(n))),** if t(n) is bounded below by constant multiple of g(n) for all values of n, and if there exist a positive constant c and non negative integer n0 such that **t(n) ≥ c*g(n)** for all n ≥ n0.



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

15. Define the asymptotic notation "theta" (Θ)

A function t(n) is said to be in Θ(g(n)) **(t(n) Є Θ(g(n))),** if t(n) is bounded both above and below by constant multiple of g(n) for all values of n, and if there exist a positive constant c1 and c2 and non negative integer n0 such that **C2*g(n) ≤ t(n) ≤ c1*g(n)** for all n ≥ n0.



3 Big-theta notation: $t(n) \in \Theta(g(n))$.

16. What is a Computer 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 a finite amount of time.

17. What are the features of an algorithm?

More precisely, an algorithm is a method or process to solve a problem satisfying the following properties:

Finiteness-Terminates after a finite number of steps

Definiteness-Each step must be rigorously and unambiguously specified.

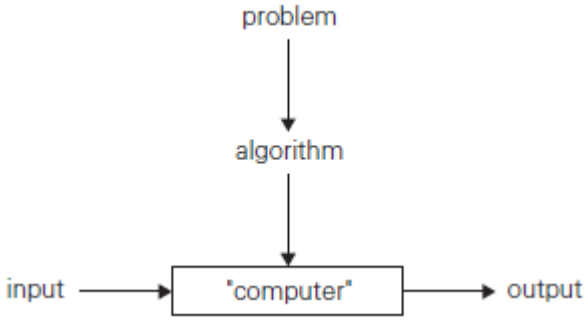
Input-Valid inputs must be clearly specified.

Output-Can be proved to produce the correct output given a valid input.

Effectiveness-Steps must be sufficiently simple and basic.

18. Show the notion of an algorithm. *Dec 2009 / May 2013*

An algorithm is a sequence of unambiguous instructions for solving a problem in a finite amount of time.



19. What are different problem types?

- Sorting
- Searching
- String Processing
- Graph problems
- Combinatorial Problems
- Geometric problems
- Numerical problems

20. What are different algorithm design techniques/strategies?

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer

- Space and time tradeoffs
- Greedy approach
- Dynamic programming
- Backtracking
- Branch and bound

21. How to measure an algorithm's running time? Nov/Dec 2017

Unit for measuring the running time is the algorithms basic operation. The running time is measured by the count of no. of times the basic operations is executed.

Basic operation: the operation that contributes the most to the total running time.

Example: the basic operation is usually the most time-consuming operation in the algorithm's innermost loop.

22. How time efficiency is analyzed?

Let c_{op} – execution time of algorithms basic operation on a particular computer.

c(n) – no. of times this operation need to be executed.

T(n) – running time.

Running time is calculated using the formula

$$T(n) \approx c_{op}\, c(n)$$

23. What are orders of growth?

Orders of Growth

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

24. What are basic efficiency classes?

Basic Efficiency classes

1	Constant
log *n*	Logarithmic
n	Linear
n log *n*	*Linearithmic*
n^2	Quadratic
n^3	Cubic
2^n	Exponential
n!	Factorial

25. Give an example for basic operations.

Input size and basic operation examples

Problem	***Input size measure***	***Basic operation***
Searching for key in a list of *n* items	Number of list's items, i.e. *n*	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers

Checking primality of a given integer *n*	size = number of digits (in binary representation)	Division
Typical graph problem	Number of vertices and/or edges	Visiting a vertex or traversing an edge

26. What are six steps processes in algorithmic problem solving? ***Dec 2009***

1. Understanding the problem.
2. Ascertaining the capabilities of a computational device.
3. Choosing between exact and approximate problem solving.
4. Deciding on appropriate data structures.
5. Algorithm Design Techniques.
6. Methods of specifying an algorithm
7. Proving an algorithm's correctness.
8. Analysing an algorithm.
9. Coding an algorithm.

27. What do you mean by Amortized Analysis?

- ✓ Amortized analysis finds the average running time per operation over a worst case sequence of operations.
- ✓ Amortized analysis differs from average-case performance in that probability is not involved; amortized analysis guarantees the time per operation over worst-case performance.

28. Define order of an algorithm.

Measuring the performance of an algorithm in relation with the input size n is known as order of growth.

29. How is the efficiency of the algorithm defined? Or . How do you measure the efficiency of an algorithm? May/June 2019

The efficiency of an algorithm is defined with the components.

(i) Time efficiency -indicates how fast the algorithm runs
(ii) Space efficiency -indicates how much extra memory the algorithm needs

30. What are the characteristics of an algorithm?

Every algorithm should have the following five characteristics

(i) Input
(ii) Output
(iii) Definiteness
(iv) Effectiveness
(v) Termination

31. What are the different criteria used to improve the effectiveness of algorithm?

(i) The effectiveness of algorithm is improved, when the design, satisfies the following constraints to be minimum.

Time efficiency - how fast an algorithm in question runs.
Space efficiency – an extra space the algorithm requires.

(ii) The algorithm has to provide result for all valid inputs.

32. Analyse the time complexity of the following segment:

```
for(i=0;i<N;i++)
for(j=N/2;j>0;j--)
sum++;
```

Time Complexity= $N * N/2 = N^2/2 \in O(N^2)$

33. Write general plan for analysing non-recursive algorithms.

i. Decide on parameter indicating an input's size.

ii. Identify the algorithm's basic operation

iii. Check the no. of times basic operation executed depends on size of input. if it depends on some additional property, then best, worst, average cases need to be investigated

iv. Set up sum expressing the no. of times the basic operation is executed. (establishing order of growth)

34. How will you measure input size of algorithms?

The time taken by an algorithm grows with the size of the input. So the running time of the program depends on the size of its input. The input size is measured as the number of items in the input that is a parameter n is indicating the algorithm's input size.

35. Write general plan for analysing recursive algorithms.

i. Decide on parameter indicating an input's size.

ii. Identify the algorithm's basic operation

iii. Checking the no. of times basic operation executed depends on size of input. if it depends on some additional property, then best, worst, average cases need to be investigated

iv. Set up the recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed

v. Solve recurrence (establishing order of growth)

36. What do you mean by Combinatorial Problem?

Combinatorial Problems are problems that ask to find a combinatorial object-such as permutation, a combination, or a subset-that satisfies certain constraints and has some desired properties.

37. Define Little "oh".

The function $f(n) = 0(g(n))$ if and only if

$$\lim_{n \to \infty} f(n) / g(n) = 0$$

38. Define Little Omega.

The function $f(n) = \omega(g(n)))$ if and only if

$$\lim_{n \to \infty} f(n) / g(n) = 0$$

39. Write algorithm using iterative function to fine sum of n numbers.

Algorithm

```
sum(a, n)
{
    S := 0.0
    For i=1 to n
      do
        S : - S + a[i];
        Return S;
}
```

40. Write an algorithm using Recursive function to fine sum of n numbers.

Algorithm

```
Rsum (a, n)
{
        If(n≤0) then
                Return 0.0;
        Else
                Return Rsum(a, n- 1) + a(n);
}
```

41. Describe the recurrence relation for merge sort?

If the time for the merging operation is proportional to n, then the computing time of merge sort is described by the recurrence relation

$$T(n) = \begin{cases} a & n = 1, \quad a \text{ a constant} \\ 2T(n/2) + n & n > 1, \quad c \text{ a constant} \end{cases}$$

42. What is time and space complexity? *Dec 2012* Part A – Refer Q. No. 6 & 7

43. Define Algorithm validation. *Dec 2012*

The process of measuring the effectiveness of an algorithm before it is coded to know whether the algorithm is correct for every possible input. This process is called validation.

44. Differentiate time complexity from space complexity. *May 2010*

Part A – Refer Q. No. 6 & 7

45. What is a recurrence equation? *May 2010*

A recurrence [relation] is an equation or inequality that describes a function in terms of its values on smaller inputs.

Examples:

Factorial: multiply n by (n –1)!

$T(n) = T(n – 1) + O(1)$ -> $O(n)$

Fibonacci: add fibonacci(n – 1) and fibonacci(n – 2)

$T(n) = T(n – 1) + T(n – 2)$ -> $O(2^n)$

46. What do you mean by algorithm? *May 2013* Part A – Refer Q. No. 1, 16 & 18

47. Define Big Oh Notation. *May 2013* Part A – Refer Q. No. 13

48. What is average case analysis? *May 2014*

The average case analysis of an algorithm is analysing the algorithm for the average input of size n, for which the algorithm runs at an average between the longest and the fastest time.

49. Define program proving and program verification. *May 2014*

- ✓ Given a program and a formal specification, use formal proof techniques (e.g. induction) to prove that the program behaviour fits the specification.
- ✓ Testing to determine whether a program works as specified.

50. Define asymptotic notation. *May 2014*

Asymptotic notations are mathematical tools to represent time complexity of algorithms for measuring their efficiency.

Types :

- Big Oh notation - 'O'
- Omega notation - 'Ω'
- Theta notation - 'Θ'
- Little Oh notation - 'o '
- Little Omega notation - 'ω'

51. What do you mean by recursive algorithm? *May 2014* Part A – Refer Q. No. 5

52. Establish the relation between O and Ω *Dec 2010*

$f(n) \in \Omega(g(n)) \Leftrightarrow g(n) \in O(f(n))$

Proof:

$O(f(n))=\{g:N\to N \mid \exists c,n0\in N \ \forall n\geq n0: g(n)\leq c\cdot f(n)\}$

$\Omega(g(n))=\{f:N\to N \mid \exists c,n0\in N \ \forall n\geq n0: f(n)\geq c\cdot g(n)\}$

Step 1/2: $f(n) \in \Omega(g(n)) \Leftrightarrow g(n) \in O(f(n))$

$\exists c,n0 \in N\ \forall n \geq n0$: $f(n) \geq c \cdot g(n) \Rightarrow f(n)g(n) \geq c \Rightarrow 1g(n) \geq cf(n) \Rightarrow g(n) \leq 1c \cdot f(n)$

And this is exactly the definition of O(f(n)).

Step 2/2: $f(n) \in \Omega(g(n)) \Leftarrow g(n) \in O(f(n))$

$\exists c,n0 \in N\ \forall n \geq n0$: $g(n) \leq c \cdot f(n) \Rightarrow ... \Rightarrow f(n) \geq 1c \cdot g(n)$

Hence proved.

53. If $f(n) = a_m n^m + ... + a_1 n + a_0$. Prove that $f(n)=O(n^m)$. *Dec 2010* Refer Class note.

54. What is best case analysis? Or Best case efficiency.

The best case analysis of an algorithm is analysing the algorithm for the best case input of size n, for which the algorithm runs the fastest among all the possible inputs of that size.

55. what do you mean worst case efficiency of algorithm.Nov/Dec 2017

The worst case analysis of an algorithm is analysing the algorithm for the worst case input of size n, for which the algorithm runs the longest among all the possible inputs of that size.

56.Consider an algorithm that finds the number of binary digits in the binary representation ofa positive decimal integer. (AU april/may 2015)

Number of major comparisons=$\lfloor \log 2n \rfloor + 1 \in \log 2n$.

Algorithm 3: Finding the number of binary digits in the binary representation of a positive decimal integer.

```
Algorithm Binary(n)
count:=1;
whilen >1
do
count:=count+ 1;
n:=⌊n/2⌋;
end
return count;
```

57.write doun the properties of asymptotic notations.(AU april/may 2015)

The following property is useful in analyzing algorithms that comprise two consecutively executed parts.

Theorem

If $t_1(n)$ $O(g_1(n))$ and $t_2(n) \in O(g_2(n))$ then,

$t_1(n) + t_2(n) \in (\max \{g_1(n), g_2(n)\})$

Proof

Since $t_1(n) \in O(g_1(n))$, there exist some constant C_1 and some non negative integer n_1 such that

$t_1(n) \leq C_1 (g_1(n))$ for all $n \geq n_1$

Since

$t_2(n)$ $O(g_2(n))$

$t_2(n) \leq C_2 (g_2(n))$ for all $n \geq n_2$

Let us denote,

$C_3 = \max \{C_1, C_2\}$ and

Consider $n \geq \max \{n_1, n_2\}$, so that both the inequalities can be used.

The addition of two inequalities becomes,

$$t_1(n) + t_2(n) \leq C_1 (g_1(n)) + C_2 (g_2(n))$$
$$\leq C_3 (g_1(n)) + C_3 (g_2(n))$$
$$\leq C_3\, 2 \max\{g_1(n), (g_2(n))\}$$

Hence,

$t_1(n) + t_2(n) \in O(\max \{g_1(n), g_2(n)\})$,

with the constants C and n_0 required by the definition being $2C_3 = 2 \max (C_1, C_2)$ and $\max \{n_1, n_2\}$ respectively.

The property implies that the algorithms overall efficiency will be determined by the part with a larger order of growth.

(i.e.) its least efficient part is

$t_1(n) \in O(g_1(n))$
$t_2(n) \in O(g_2(n))$
} $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$

58. Give the Euclid's algorithm for computing *gcd(m, n)* (AU nov 2016) or write an algorithm to compute the greatest common divisor of two numbers (Apr/ May-2017)(or)

Give the Euclid's algorithm for computing gcd of two numbers. (May/June 2018)

	2	
Polynomial	**Quadratic**	**Quadratic**
1	0	1
2	1	4
4	6	16
8	28	64
10	45	10^2
10^2	4950	10^4
Complexity	Low	High
Growth	Low	high

ALGORITHM *Euclid_gcd(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 $n \neq 0$ **do**
    $r \leftarrow m \bmod n$
    $m \leftarrow n$
    $n \leftarrow r$
return *m*

Example: gcd*(*60, 24*)* = gcd*(*24, 12*)* = gcd*(*12, 0*)* = 12.

59.Compare the order of growth n(n-1)/2 and n^2. (AU nov 2016)

n(n-1)/2 is lesser than the half of n^2

60.The *(log n)*th smallest number of n unsorted numbers can be determined in O(n) average-case time

Ans: True

61.Fibonacci algorithm and its recurrence relation

Algorithm for computing Fibonacci numbers

First method

Algorithm F(n)

//Computes the nth Fibonacci number recursively by using its definition.

//Input: A nonnegative integer n

//Output: The nth Fibonacci number

```
if n<1
        return n
Else
        return F(n-1)+(n-2)
```

the algorithm's basic operation is addition.

Let A(n) is the number of additions performed by the algorithm to compute F(n).

The number of additions needed to compute F(n-1) is A(n-1) and the number of additions needed to compute F(n-2) is A(n-2).

62. Design an algorithm to compute the area and circumference of a circle

1. Find Area, Diameter and Circumference of a Circle.

ALGORITHM

Step 1: Start
Step 2: Initialize PI to 0
Step 3: Read radius of the circle
Step 4: Calculate the product of radius with itself and PI value
Step 5: Store the result in variable area
Step 6: Calculate the product of radius with 2
Step 7: Store the result in variable diameter
Step 8: Calculate the product of PI and diameter value
Step 9: Store the result in variable circumference
Step 10: Print the value of area, diameter and circumference
Step 11: Stop

PSEUDO CODE

Step 1: Begin
Step 2: Set PI to 3.14
Step 3: Read radius
Step 4: Compute the product of radius with itself and PI value
Step 5: Set the result to area
Step 6: Compute the product of radius with 2
Step 7: Set the result to diameter
Step 8: Calculate the product of PI and diameter value
Step 9: Set the result to circumference
Step 10: Display area, diameter, circumference
Step 11: End

63. What is a basic operation?

A basic operation could be: An assignment. A comparison between two variables. An arithmetic operation between two variables. The worst-case input is that input assignment for which the most basic operations are performed.

Basic Operations on Sets. The set is the basic structure underlying all of mathematics. In algorithm design, sets are used as the basis of many important abstract data types, and many techniques have been developed for implementing set-based abstract data types.

64. Define algorithm. List the desirable properties of an algorithm.

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

An algorithm must satisfy the following properties: Input: The algorithm must have input valuesfrom a specified set. ... The output values are the solution to a problem. Finiteness: For any input, the algorithm must terminate after a finite number of steps. Definiteness: All steps of the algorithm must be precisely defined.

65. Define best, worst, average case time complexity.

- The *worst-case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size *n*. It represents the curve passing through the highest point of each column.
- The *best-case complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size *n*. It represents the curve passing through the lowest point of each column.
- Finally, the *average-case complexity* of the algorithm is the function defined by the average number of steps taken on any instance of size *n*.

66.Prove that the of f(n)=o(g(n)) and g(n)=o(f(n)),then f(n)=θ g(n). OR state the transpose symmetry property of O and Ω **April/May 2019,Nov/Dec 2019**

Given function:

f(n) and g(n)

f(n)= O(g(n)) when f(n) $\leq C_1 g(n)$ for all $n \geq n_0$--------(1)

f(n)= Ω(g(n)) when f(n) $\geq C_2 g(n)$ for all $n \geq n_0$--------(2)

from (1) and (2)

$C_2\, g(n) \leq f(n) \leq C_{1g(n)}$ for all $n \geq n_0$ -------(3)

(i.e) Θ(g(n)) = O(g(n))Ω(g(n))

From (3) f(n) = Θ(g(n)) hence proved

67. Define recursion

A function may be recursively defined in terms of itself. A familiar example is the Fibonacci number sequence: F(n) = F(n − 1) + F(n − 2).

For such a definition to be useful, it must be reducible to non-recursively defined values: in this case F(0) = 0 and F(1) = 1. ccurs when a thing is defined in terms of itself or of its type. Recursion is used in a variety of disciplines ranging from linguistics to logic.

The most common application of recursion is in mathematics and computer science, where a function being defined is applied within its own definition.

While this apparently defines an infinite number of instances (function values), it is often done in such a way that no loop or infinite chain of references can occur.

68. List the reasons for choosing an approximate algorithm.

Approximation algorithms are typically used **when finding an optimal solution is intractable**, but can also be used in some situations where a near-optimal solution can be found quickly and an exact solution is not needed. Many problems that are NP-hard are also non-approximable assuming P≠NP.

PART – B

1. Explain the notion of an algorithm with diagram. *May2014*

Synopsis:

- Introduction
- Definition
- Diagram
- Characteristics of an Algorithm / Features of an Algorithm
- Rules for writing an Algorithm
- Implementation of an Algorithm
- Order of an Algorithm
- Program
- Example : GCD

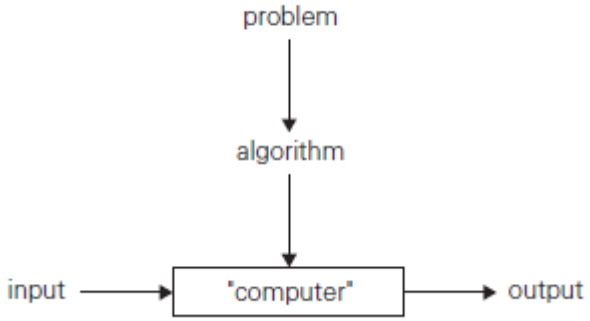
Introduction:

- An algorithm is a sequence of finite number of steps involved to solve a particular problem.
- An input to an algorithm specifies an instance of the problem the algorithm solves.
- An algorithm can be specified in a natural language or in a pseudo code.
- Algorithm can be implemented as computer programs.
- The same algorithm can be represented in several different ways.
- Several algorithms for solving the same problem may exist.
- Algorithms for the same problem can be based on different ideas and can solve the problem with dramatically different speeds.

Definition:

- An algorithm is a sequence of non ambiguous instructions for solving a problem in a finite amount of time.
- Each algorithm is a module, designed to handle specific problem.
- The non ambiguity requirement for each step of' an algorithm cannot be compromised.
- The range of inputs for which an algorithm works has to be specified carefully.

Diagram:



Characteristics of an algorithm / Features of an Algorithm

The important and prime characteristics of an algorithm are,

- ✓ **Input:**Zero or more quantities are externally supplied.
- ✓ **Output:**At least one quantity is produced.
- ✓ **Definiteness:**Each instruction is clear and unambiguous.
- ✓ **Finiteness:**For all cases the algorithm terminates after a finite number of steps.
- ✓ **Efficiency:**Every instruction must be very basic.
- ✓ An algorithm must be expressed in a fashion that is completely free of ambiguity.
- ✓ It should be efficient.
- ✓ Algorithms should be concise and compact to facilitate verification of their correctness.

Writing an algorithm

- Algorithm is basically a sequence of instructions written in simple English language.

- The algorithm is broadly divided into two sections

Algorithm heading
It consists of name of algorithm, problem description , input and output.
Algorithm Body
It consists of logical body of the algorithm by making use of various programming constructs and assignment statement.

Rules for writing an algorithm.

Algorithm is a product consisting of heading and body. The heading consists of keyword **algorithm** and name of the algorithm and parameter list. The syntax is

Algorithm name (p1, p2,.....pn)

1. Then in the heading section we should write following things :
   // Problem Description;
   // Input:
   //Output:
2. Then body of an algorithm is written, in which various programming constructs like if , for , while or some assignment statement may be written.
3. The compound statements should be enclosed within { and } brackets.
4. Single line comments are written using // as beginning of comment.
5. The **identifier** should begin by latter and not by digit. An identifier can be a combination of alphanumeric string.
   - It is not necessary to write data types explicitly for identifiers. It will be represented by the context itself.
   - Basic data types used are integer, float, and char, Boolean and so on.
   - The pointer type is also used to point memory locations.
   - The compound data type such as structure or record can also be used.
6. Using assignment operator ← an assignment statement can be given.
   For instance: **Variable ← expression**
7. There are other types of operators' such as Boolean operators such as true or false. Logical operators such as AND, OR, NOT. And relational operators such as < , <= , >, >=, = , !=.
8. The array indices are stored with in square brackets '[' ']'. The index of array usually starts at zero. The multidimensional arrays can also be used in algorithm.
9. The inputting and outputting can be done using read and write.
   For example:
   Write ("this message will be displayed on console ");
   Read (Val);
10. The conditional statements such as if –then – else are written in following form
    If (condition) then statement
    If (condition) then statement else statement
    If the **if – then** statement is of compound type then {and} should be used for enclosing block
11. While statement can be written as :

```
While (condition)do
{
        Statement 1
        Statement 2
          :
```

Statement n

}

While the condition is true the block enclosed with { } gets executed otherwise statement after} will be executed.

12. The general form for writing for loop is :

For variable ← value$_1$ to value$_n$ do

{

Statement 1

Statement 2

:

Statement n

}

Here value$_1$ is initialization condition and value$_n$ is a terminating condition the step indicates the increments or decrements in value$_1$ for executing the for loop.

Sometime a keyword step is used to denote increment or decrement the value of variable for example

For i ← 1 to n step 1 ←

Here variable i is incremented by 1 at each iteration

{

Write (i)

}

13. The **repeat – until** statement can be written as

Repeat

Statement 1

Statement 2

:

Statement n

Until (condition)

14. The **break** statement is used to exit from inner loop. The return statement is used to return control from one point to another. Generally used while exiting from function

Note: The statements in an algorithm executes in sequential order i.e. in the same order as they appear – one after the other

Example 1 : Write an algorithm to count the sum of n numbers

```
Algorithm  sum (1, n)
        //Problem description : this algorithm is for finding the
        //sum of given n numbers
        //Input: 1 to n numbers
        //Output: the sum of n numbers
    Result ← 0
     For i 1 to n do
          i  ← i+1
    Result ← result + i
 Return result
```

Example 2: Write an algorithm to check whether given number is even or odd.

```
Algorithm eventest ( val)
        //Problem description : this algorithm test whether given
        //number is even or odd
        //Input: the number to be tested i.e .val
        //Output: appropriate messages indicating even or odd
```

```
If (val % 2 = 0) then
        Write ("given number is even ")
Else
        Write ("given number is odd")
```

Example 3: Write an algorithm for sorting the elements.

```
Algorithm sort (a, n)
        //Problem description: sorting the elements in ascending
        //order
        //Input: an array in which the elements in ascending order
        //is total number of elements in the array
        //Output: the sorted array
        For i 1 to n do
                For j i + 1 to  n-1 do
                        If (a[i]>a[j]) then
                        {
                                temp ← a[i]
                                a[i] ←a[j]
                                a[j] ←temp
                        }
        Write ( " list is sorted ")
```

Example 4: Write an algorithm to find factorial of n number.

```
Algorithm fact (n)
        //Problem description: this algorithm finds the factorial.
        //for given number n
        //Input : the number n of which the factorial is to be
        //calculated.
        //Output : factorial value of given n number.
        If( n ← 1) then
                Return 1
        Else
                Return n * fact(n-1)
```

Example 5:

Write an algorithm to perform multiplication of two matrices

```
Algorithm mul (A, b, n)
        //Problem description: this algorithm is for computing
        //multiplication of two matrices
        //Input : the two matrices A, B and order of them as n
        //Output : The multiplication result will be in matrix c
        For i ← 1 to n do
                For j ← 1 to n do
                        C [i,j] ← 0

                For k ← 1 to n do
                        C[I ,j ] ←c[i, j] +A[i,k]B[k,j]
```

Implementation of algorithms

An algorithm describes what the program is going to perform. It states some of the actions to be executed and the order in which these actions are to be executed.

The various steps in developing algorithm are,

1. Finding a method for solving a problem. Every step of an algorithm should be in a precise and in a clear manner. Pseudo code is also used to describe the algorithm.
2. The next step is to validate the algorithm. This step includes, all the algorithm should be done manually by giving the required input, performs the required steps including in the algorithm and should get the required amount of output in an finite amount of time.
3. Finally, implement the algorithm in terms of programming language.

Order of an algorithm

The order of an algorithm is a standard notation of an algorithm that has been developed to represent function that bound the computing time for algorithms. It is an order notation. It is usually referred as O-notation.

Example

Problem size = 'n'
Algorithm = 'a' for problem size n
The document mechanism execution = Cn^2 times
where C – constant
Then the order of the algorithm 'a' = $O(n^2)$
where n^2 = Complexity of the algorithm 'a'.

Program

- A set of explicit and unambiguous instructions expressed using a programming languages constructs is called a program.
- An algorithm can be converted into a program, using any programming language. Pascal, Fortran, COBOL, C and C++ are some of the programminglanguages.

Difference between program and algorithm:

Sno	Algorithm	Program
1	Algorithm is finite.	Program need to be finite.
2	Algorithm is written using natural language or algorithmic language.	Programs are written using a specific programming language.

1.A. write an algorithm using recursion that determines the GCD of two numbers.Determine the time and space complexity Nov/Dec 2019

Example : Calculating Greatest common Divisor

The Greatest common Divisor (GCD) of two non zero numbers a and b is basically the largest integer that divides both a and b evenly i.e with a remainder of zero.
GCD using three methods

1. Euclid's algorithm
2. Consecutive integer checking algorithm
3. Finding Gusing repetitive factors

Euclid's algorithm to compute Greatest Common Divisor (GCD) of two non negative integers.

Euclid's algorithm is based on applying related equality

gcd (m, n) = gcd (n, m mod n) until the m and n is equal to 0

Where m mod n is the remainder of the division of m by n

Step 1: Start
Step 2: If n = 0, return the value of m as the answer and stop,
otherwise proceed to step 3.

Step 3: Divide m by n and assign the value of the remainder to r.
Step 4: Assign the value of n to m and the value of r to n. Goto step 2
Step 5: Stop

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 $n \neq 0$ **do**
$r \leftarrow m \bmod n$
$m \leftarrow n$
$n \leftarrow r$
return m

Example, gcd (60,24) can be computed as follows,

gcd (60,24)	gcd (m, n) m =60, n=24; m/n = 2 (remainder 12) n=m=24 r=n=12
gcd (24, 12)	m/2 = 2 (remainder 0) n=m=12 r=n=0
gcd (12, 0) =12	

Hence, gcd(60, 24) = gcd(24,l2)=gcd(12,0)=12

2. Consecutive integer checking algorithm

In this method while finding the GCD of a and b we first of all find the minimum value of them. Suppose if , value of b is minimum then we start checking the divisibility by each integer which is lesser than or equal to b.

Example:

a = 15 and b =10 then
t= min(15,10)
since 10 is minimum we will set value of t = 10 initially.

Consecutive integer checking algorithm for computing gcd(m, n)

Step 1: Start
Step 2: Assign the value of mini {m, n} to t
Step 3: Divide m by t. If the remainder of this division is 0, go to step 4,
Otherwise goto step 5.
Step 4: 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 step 5.
Step 5: Decrease the value of t by I. Go to step 3.
Step 6: Stop

Algorithm GCD intcheck (a,b)

```
//Problem description : this algorithm computes the GCD of //two numbers a and b using consecutive integer checking
//method
//Input : two integers a and b
//Output: GCD value of a and b
 t ← min ( a, b)
while (t>=1) do
{
        If ( a mod t == 0 AND b mod t == 0) then
                Return t
```

```
        Else
                t ← t-1
    }
    Return 1
```

3. Finding GCD using repetitive factors

The third procedure for finding the greatest common divisor is middle school procedure.

Middle School Method

For the numbers 60 and 24

```
2 | 60          2 | 24
  | 30            | 12
3 | 15          2 |  6
  |  5            |  3
```

60=2x2x3 x5
24=2x2x3 x2
gcd (60,24) =2x2x3 =12

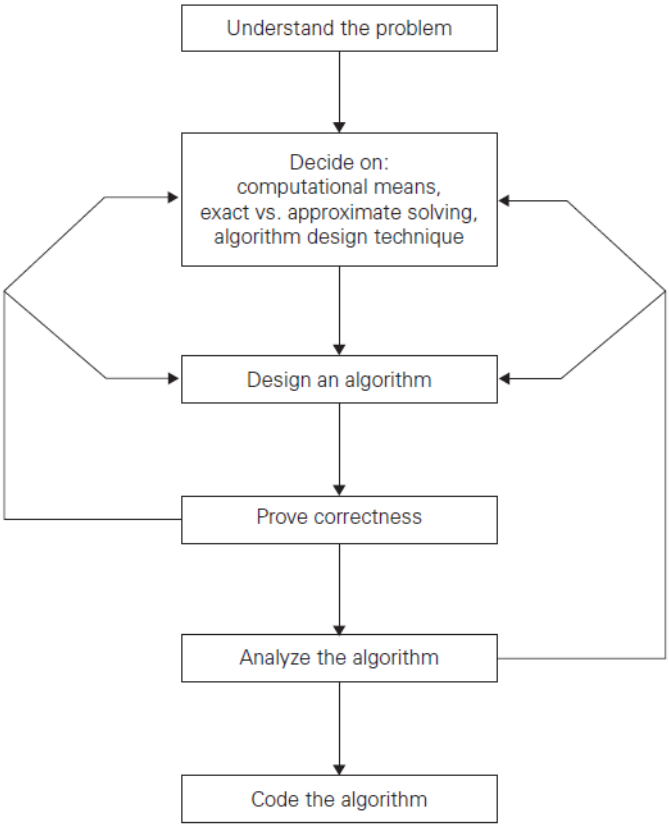
Algorithm:

Step 1: Start
Step 2: Find the prime Factor of m.
Step 3: Find the prime factors of n.
Step 4: Identify all the common factors in the two prime expressions Found in step 2 and step 3. If' P is a common factor occurring pm and pn times in m and n respectively. It should be repeated min (pm, and pn) times.
Step 5: Compute the product of the all the common factors and return it as the greatest common divisor of the numbers given.
Step 6: Stop.

2. Explain the Fundamentals of Algorithmic problem solving. Or explain the steps involved in problem solving *May 2014* ,April/May 2019

Sequential steps in designing and analysing an algorithm

1. Understanding the problem.
2. Ascertaining the capabilities of a computational device.
3. Choosing between exact and approximate problem solving.
4. Deciding on appropriate data structures.
5. Algorithm Design Techniques.
6. Methods of specifying an algorithm
7. Proving an algorithm's correctness.
8. Analysing an algorithm.
9. Codinganalgorithm.



1.Understanding the problem:

- ✓ To design an algorithm, understand the problem completely by reading the problem's description carefully.
- ✓ Read the problem description carefully and clear the doubts.
- ✓ Specify exactly the range of inputs the algorithm need to handle.
- ✓ Once the problem is clearly understandable, then determine the overall goals but it should be in a precise manner.
- ✓ Then divide the problem into smaller problems until they become manageable size.

2. Ascertaining the capabilities of a computational device

Sequential Algorithm:

- ✓ Instructions are executed one after another, one operation at a time.
- ✓ This is implemented in RAM model.

Parallel Algorithm:

- ✓ Instructions are executed in parallel or concurrently.

3. Choosing between exact and appropriate problem solving

- ✓ The next principal decision is to choose between solving the problem exactly or solving the problem approximately.
- ✓ The algorithm used to solve the problem exactly called **exact algorithm**.
- ✓ The algorithm used to solve the problem approximately is called **approximation algorithm.**

Reason to choose approximate algorithm

- There are important problems that simply cannot be solved exactly such as
  - Extracting square roots.
  - Solving non linear equations.
  - Evaluating definite integrals.
- ✓ Available algorithms for solving problem exactly can be unacceptably slow, because

of the problem's intrinsic complexity. Ex: Travelling salesman problem

4. Deciding on appropriate data structures

Data structure is important for both design and analysis of algorithms.

Algorithm + Data Structures = Programs.

In Object Oriented Programming, the data structure is important for both design and analysis of algorithms.

The variability in algorithm is due to the data structure in which the data of the program are stored such as

1. How the data are arranged in relation to each other.
2. Which data are kept in memory
3. Which data are kept in files and how the files are arranged.
4. Which data are calculated when needed?

5. Algorithm Design Techniques

An algorithm design techniques or strategy or paradigm is general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

Uses

- ✓ They provide guidance for designing algorithms or new problems.
- ✓ They provide guidance to problem which has no known satisfied algorithms.
- ✓ Algorithm design technique is used to classify the algorithms based on the design idea.
- ✓ Algorithm design techniques can serve as a natural way to categorize and study the algorithms.

6. Methods of specifying an algorithm

There are two options, which are widely used to specify the algorithms.

They are

- o Pseudo code
- o Flowchart

Pseudo code

- o A pseudo code is a mixture of natural language and programming language constructs.
- o A pseudo code is more precise than a natural language
- o For simplicity, declaration of the variables is omitted.
- o For, if and while statements are used to show the scope of the variables.
- o "←" (Arrow) - used for the assignment operation.
- o "//" (two slashes) - used for comments.

Flow chart

- o It is a method of expressing an algorithm by a collection of connected geometric shapes containing description of the algorithms steps.
- o It is very simple algorithm.
- o This representation technique is inconvenient.

7. Proving an Algorithm's correctness

Once an algorithm has been specified, then its correctness must be proved.

- ✓ An algorithm must yield a required result for every legitimate input in a finite amount of time.
- ✓ A mathematical induction is a common technique used to prove the correctness of the algorithm.
- ✓ In mathematical induction, an algorithm's iterations provide a natural sequence of steps needed for proofs.
- ✓ If the algorithm is found incorrect, need to redesign it or reconsider other decisions.

8. Analysing an algorithm

- ✓ Efficiency of an algorithm is determined by measuring the time, space and amount of

resources, it uses for executing the program.

- ✓ The efficiency of the algorithm is determined with respect to central processing units time and internal memory.
- ✓ There are two types of algorithm efficiency.

They are

- Time efficiency (or) Time Complexity
- Space efficiency (or) Space Complexity

Time Efficiency / Time Complexity

- ✓ Time efficiency indicates how fast the algorithm runs.
- ✓ The time taken by a program to complete its task depends on the number of steps in an algorithm.
- ✓ The time required by a program to complete its task will not always be the same.
- ✓ It depends on the type of problem to be solved.

It can be of two types.

- Compilation Time
- Run Time (or) Execution Time

- ✓ The time (T) taken by an algorithm is the sum of the compile time and execution time.

Compilation Time

- ✓ The amount of time taken by the compiler to compile an algorithm is known as compilation time.
- ✓ During compilation time, it does not calculate the executable statements, it calculates only the declaration statements and check for any syntax and semantic errors.
- ✓ The different compilers can take different times to compile the same program.

Execution Time

- ✓ The execution time depends on the size of the algorithm.
- ✓ If the number of instructions in an algorithm is large then the run time is also large.
- ✓ If the number of instructions in an algorithm is small then the time need to execute the program is small.
- ✓ The execution time is calculated for executable statements and not for the declaration statements.
- ✓ The complexity is normally expressed as an order of magnitude.
- ✓ Example: O (n^ 2)
- ✓ The time complexity of a given algorithm is defined as computation of function f() as a total number of statements that are executed for computing the value f(n).
- ✓ The time complexity is a function which depends on the value of n.

The time complexity can be classified as 3 types.

They are

1. Worst Case analysis
2. Average Case analysis
3. Best Case analysis

Worst Case Analysis

- ✓ The worst case complexity for a given size corresponds to the maximum complexity encountered among all problem of the same size.
- ✓ Worst case complexity takes a longer time to produce a desired result.

This can be represented by a function f(n).

$f(n) = n^2$ or $n \log n$

Average Case Analysis

- ✓ The average case analysis is also known as the expected complexity which gives measure of the behaviour of an algorithm averaged over all possible problem of the same size.
- ✓ Average case is the average time taken by an algorithm for producing a desired output.

Best Case Analysis

- ✓ Best case is a shortest time taken by an algorithm to produce the desired result.

Space Complexity

- ✓ Space efficiency indicates how much extra memory the algorithm needs.
- ✓ The amount of storage space taken by the algorithm depends on the type of the problem to be solved.
- ✓ The space can be calculated as,
- ✓ A fixed amount of memory occupied by the space for the program code is space occupied by the variable used in the program.
- ✓ A variable amount of memory occupied by the component variable dependent on the problem is being solved.
- ✓ This space is more or less depending upon whether the program uses iterative or recursive procedures.

There are three different space considered for determining the amount of memory used by the algorithm.

They are

- o Instruction Space
- o Data Space
- o Environment Space

Instruction Space

- ✓ When the program gets compiled, then the space needed to store the compiled instruction in the memory is called instruction space.
- ✓ The instruction space independent of the size of the problem

Data Space

- ✓ The memory space used to hold the variables of data elements are called data space.
- ✓ The data space is related to the size of the Problem

Environment Space

- ✓ It is the space in memory used only on the execution time for each Function call.
- ✓ It maintains runtime stack in that it holds returning address of the previous functions.
- ✓ Every function on the stack has return value and a pointer on it.

Characteristics of an algorithms

- o Simplicity
- o Generality

Simplicity

- o Simpler algorithms are easier to understand.
- o Simpler algorithms are easier to program.
- o The resulting programs contains only few bugs.
- o Simpler algorithms are more efficient compared to the complicated alternatives.

Generality

- o The characteristic of an algorithm generality has two issues.
- o They are
  - ▪ Generality' of the problem the algorithm solves.
  - ▪ Range of inputs it accepts.

9. Coding an Algorithm

- ✓ Implementing an algorithm correctly is necessary but not sufficient to diminish the algorithm's power by an inefficient implementation.
- ✓ The standard tricks such as computing a loop's invariant (an expression that does not change its value) outside the loop, collecting common sub expressions, replacing expensive operations by cheaper ones and so on should be known to the programmers such factors can speed up a program only by a constant factor, where as a better algorithm can make a difference in running time by orders of magnitude.

- ✓ Once an algorithm has been selected, a 10-50% speed up may be worth an effort.
- ✓ An algorithm's optimality is not about the efficiency of an algorithm but about the complexity of the problem it solves.

3. Explain the important problem types.

Some of the most important problem types are

1. Sorting
2. Searching
3. String Matching (or) String processing
4. Graph Problems
5. Combinatorial problems
6. Geometric problems
7. Numerical Problems

1. Sorting

- ✓ Sorting means arranging the elements in increasing order or in decreasing order.
- ✓ The sorting can be done on numbers , characters (alphabets), string or employees record.
- ✓ Many algorithms are used to perform the task of sorting.
- ✓ Sorting is the operation of arranging the records of a table according to the key value of the each record.
- ✓ A table of a file is an ordered sequence of records r[l], r[2].. r[n] each containing a key k[l], k[2]....k[n]. The table is sorted based on the key.

Properties of Sorting Algorithms

The two properties of Sorting Algorithms are

1. Stable
2. In-place

Stable:

- ✓ A sorting algorithm is called **stable**, if it preserves the relative order of any two equal elements in its input.
- ✓ In other words, if an input list contain two equal elements in positions i and j, where i<j, then in the sorted list they have to be in position i' and j' respectively, such that i' < j'

In-place

- ✓ An algorithm is said to be **in-place** if it does not require extra memory, except, possibly for a few memory units.

The important **criteria for the selection of a sorting** method for the given set of data items are as follows.

1. Programming time of the sorting algorithm.
2. Execution time of the program
3. Memory space needed for the programming environment

The **main objectives** involved in the design of sorting algorithms are

1. Minimum number of exchanges.
2. Large volume of data blocks movement.

Types of Sorting

The two major classification of sorting methods are

1. Internal Sorting methods
2. External Sorting methods

Internal Sorting

- ✓ The key principle of internal sorting is that all the data items to be sorted are retained in the main memory and random access memory.
- ✓ This memory space can be effectively used to sort the data items.
- ✓ The various internal sorting methods are

1. Bubble sort
2. Selection sort
3. Shell sort

4. Insertion sort
5. Quick sort
6. Heap sort

External Sorting

- ✓ The idea behind the external sorting is to move data from secondary storage to mail memory in large blocks for ordering the data.
- ✓ The most commonly used external sorting method is merge sort.

2. Searching

- ✓ One of the important applications of array is searching
- ✓ Searching is an activity by which we can find out the desired element from the list. The element which is to be searched is called **search key**
- ✓ There are many searching algorithm such as sequential search , Fibonacci search and more.

Searching in dynamic set of elements

- ✓ There may be of elements in which repeated addition or deletion of elements occur.
- ✓ In such a situation searching an element is difficult.
- ✓ To handle such lists supporting data structures and algorithms are needed to make the list balanced (organized)

3. String processing

A **string is a collection** of characters from an alphabet.

Different type of strings are

- o Text string
- o Bit string

Text String It is a collection of letters, numbers and special characters.

Bit String It is collection of zeros and ones.

- Operations performed on a string are
  1. Reading and writing strings
  2. String concatenation
  3. Finding string length
  4. String copy
  5. String comparison
  6. Substring operations
  7. Insertions into a string
  8. Deletions from a string
  9. Pattern matching

Pattern Matching or String matching

The process of searching for an occurrence of word in a text is called Pattern matching.

Some of the algorithms used for pattern matching are

1. Simple pattern matching algorithm
2. Pattern matching using Morris Pratt algorithm
3. Pattern matching using Knuth-Morris-Pratt algorithm

4. Graph Problems

- ✓ Graph is a collection of vertices and edges.
- ✓ Formally, a graph G={ V, E } is defined by a pair of two sets.
- ✓ A finite set V of items called Vertices and a set E of pairs of these items called edges.
- ✓ If the pairs of vertices are ordered, then G is called a directed graph because every edge is directed.
- ✓ In a directed graph the direction between two nodes are not same G(V,W)!=G(W,V)
- ✓ If the pair of the vertices are unordered then G is called an undirected graph.
- ✓ In undirected graph, the edges has no specific direction.
- ✓ The graph problems involve graph traversal algorithms, shortest path algorithm and topological sorting and so on. Some graph problems are very hard to solve.
- ✓ For example travelling salesman problem, graph colouring problems

5. Combinatorial Problems

- ✓ The travelling salesman problem and the graph colouring problems are examples of combinatorial problems.
- ✓ A combinatorial object such as a permutation a combination or a subset that satisfies certain constraints and has some desired property such as maximizes a value or minimizes a cost should be find.
- ✓ Combinatorial problems are the **most difficult problems**.

The reason is,

1. As problem size grows the combinatorial objects grow rapidly and reach to huge value. size.
2. There is no algorithms available which can solve these problems in finite amount of time
3. Many of these problems fall in the category of unsolvable problem.

Some combinatorial problems can be solved by efficient algorithms.

6. Geometric Problems

- ✓ Geometric algorithms deal with geometric objects such as points ,lines and polygons.
- ✓ The procedure for solving a variety of geometric problems includes the problems of constructing simple geometric shapes such as triangles, circles and so on.

The two classic problems of computational geometry are the

1. Closest pair problem
2. Convex hull problem

- ✓ The closest pair problem is self explanatory. Given n points in the plane, find the closest pair among them.
- ✓ The convex hull problem is used to find the smallest convex polygon that would include all the points of a given set.
- ✓ The geometric problems are solved mainly in applications to computer graphics or in robotics

6.Numerical problems

- ✓ Numerical problems are problems that involve mathematical objects of continuous nature such as solving equations and systems of equations computing definite integrals evaluating functions and so on.
- ✓ Most of the mathematical problems can be solved approximate algorithms.
- ✓ These algorithms require manipulating of the real numbers; hence we may get wrong output many times.

3.Explain the fundamentals of the analysis framework. Or explain time-space trade off of the algorithm designed. April/May 2019

- Efficiency of an algorithm can be in terms of time or space.
- This systematic approach is modelled by a frame work called as analysis frame work.

Analysis framework

- The efficiency of an algorithm can be decided by measuring the performance of an algorithm.
- The performance of an algorithm is computed by two factors
  - amount of time required by an algorithm to execute
  - amount of storage required by an algorithm

Overview

- Space complexity
- Time complexity
- Measuring an Input's size
- Measuring Running Time
- Orders of Growth

Space complexity

- The space complexity can be defined as amount of memory required by an algorithm to run.

- To compute the space complexity we use two factors: constant and instance characteristics.

- The space requirement S(p) can be given as **S(p) = C+ S(p)**
  Where C is a constant i.e. fixed part and it denotes the space of inputs and outputs.

Time complexity

- The time complexity of an algorithm is the amount of computer time required by an algorithm to run to completion.
- For instance in multiuser system, executing time depends on many factors such as
  - System load
  - Number of other programs running
  - Instruction set used
  - Speed underlying hardware
- The time complexity is therefore given in term of frequency count
- Frequency count is a count denoting number of times of execution of statement

Example

```
For (i=0; i<n; i++)
{
        sum = sum + a[i];
}
```

Statement	**Frequency count**
i=0	1
i<n	This statement executes for (n+1) times. When conditions is true i.e. when i<n is true , the execution happens to be n times , and the statement execute once more when i<n is false
i++	n times
sum = sum + a[i]	n times
Total	**3n + 2**

Measuring an Input's size

- All algorithms run longer on larger inputs.
- Ex: Sorting larger arrays, multiply larger matrices etc.
- Investigates an algorithm efficiency as a function of some parameter n indicating the algorithm input size.
- Example:
  - In problem of evaluating a polynomial $p(x) = a_n x^n + \ldots + a_0$ of degree n, the parameter will be the polynomial's degree or the number of its coefficients which is larger by one than its degree.
- In spell checking algorithm,
  - If algorithm examines the individual character of its input, then the size of the input is the no. of characters.
  - If the algorithm processes the word, the size of the input is the no. of words.

Measuring Running Time

- Some units of time measurement such as a second, a millisecond and so on can be used to measure the running time of a program implementing the algorithm.
- **Drawbacks**
  1. Dependence on the speed of a particular computer
  2. Dependence on the quality of a program implementing the algorithm.
  3. The compiler used in generating the machine code.

4. The difficulty of clocking the actual running time of the program

- Since we are in need to measure an algorithm's efficiency, we should have a metric that does not depend on these factors.
- One possible approach is to count the number of times of the algorithm's operations is executed. But this approach is difficult and unnecessary.
- 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.
- It is not so difficult to identify the basic operation of an algorithm: it is usually the most time consuming operation in the algorithm's innermost loop.

Example

- Most sorting algorithms work by comparing the elements (keys) of a list being sorted with each other. For such algorithms the basic operation is a Key Comparison.

Problem statement	**Input Size**	**Basic operation**
Searching a key element from the list of n elements	List of n elements	Comparison of key with every element of list
Performing matrix multiplication	The two matrixes with order n×n	Actual multiplication of the elements in the matrices
Computing GCD of two numbers	Two numbers	Division

The formula to compute the execution time using basic operation is

. $\mathbf{T(n) \approx C_{op}\, C(n)}$

Where T(n) – running time

C(n) – no. of times this operation is executed.

Cop – execution time of algorithms basic operation.

Orders of Growth

- Measuring the performance of an algorithm in relation with the input size n is called order of growth.

Worst Case, Best Case and Average Case efficiencies

- It is reasonable to measure an algorithm's efficiency as a function of a parameter indicating the size of the algorithm's input.
- But for many algorithms the running time depends not only on an input size but also on the specifics of a particular input.

Example: **Sequential Search or Linear Search AU: Dec -11, Marks 10**

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 in 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
```

- This algorithm searches for a given item using some search key K in a list of 'n' elements by checking successive elements of the list until a match with the search key

is found or the list is exhausted.

- The algorithm makes the largest number of key comparisons among all possible inputs of size n: $C_{worst}(n)=n$

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 of that size.
- The way to determine the worst case efficiency of an algorithm is that:
  - Analyse the algorithm to see what Kind of inputs yield the largest value of the basic operations count C(n) among all possible inputs of size n and then compute is w value **$C_{worst} = (n)$.**

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.
- The way to determine the best case efficiency of an algorithm is as follows.
  - First, determine the kind of inputs of size n.
  - Then ascertain the value of C(n) on these inputs.
- **Example:** For sequential search, the best case inputs will be lists of size 'n' with their first elements equal to a search key: **$C_{best}(n) = 1$.**

Average case efficiency

- It yields the necessary information about an algorithm's behaviour on a "typical" or "random" input.
- To determine the algorithm's average case efficiency some assumptions about possible inputs of size 'n'.
- The average number of key comparisons C_{avg} (n) can be computed as follows:
  - In case of a successful search the probability of the first match occurring in the 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 case of an unsuccessful search, the number of comparisons is 'n' with the probability of such a search being (1-p). Therefore,

$$C_{avg}(n)=[1.\frac{p}{n}+2.\frac{p}{n}+.....i.\frac{p}{n}+...n.\frac{p}{n}]+n.(1-p)$$

$$= \frac{p}{n}[1+2+3+....+i+...+n]+n(1-p)$$

$$= \frac{p}{n}\frac{n(n+1)}{2}+n(1-p)$$

$$\mathbf{C_{avg}(n)= \frac{p(n+1)}{2n} + n(1-p)}$$

There may be n elements at which chances of 'not getting element' are possible. Hence n . (1-p)

Example:

- If p = 1 (i.e.) if the search is successful, then the average number of key comparisons made by sequential search is (n+1)/2.
- If p = 0 (i.e.) if the search is unsuccessful, then the average number of key comparisons will be 'n' because the algorithm will inspect all n elements on all such inputs.

4. Explain the Asymptotic Notations and its properties? Or explain briefly Big oh notation , Omega notation and Theta notation give an example (Apr/May-2017) or what are the Rules of Manipulate Big-Oh Expression and about the typical growth rates of

algorithms? Nov/Dec 2017 Nov/Dec 2018

Define Big O notation, Big Omega and Big Theta Notation. Depict the same graphically and explain. May/June 2018 , Nov/Dec 2019

Explain the importance of asymptotic analysis for running time of an algorithm with an example. (April/May 2021)

Asymptotic notations are mathematical tools to represent time complexity of algorithms for measuring their efficiency. Types :

- Big Oh notation - 'O'
- Omega notation - 'Ω'
- Theta notation - 'Θ'
- Little Oh notation - 'o '

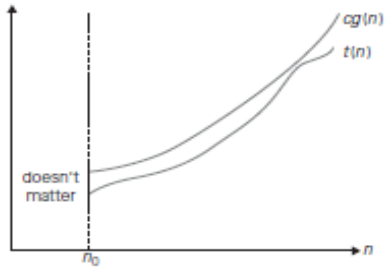
Big Oh notation (O)

- The big oh notation is denoted by **'O'.**
- It is a method of representing the **upper bound of algorithm's running time.**
- Using big oh notation we can give **longest amount of time taken** by the algorithm to complete.

Definition

A function t(n) is said to be in O(g(n)) **(t(n) Є O(g(n))),** if t(n) is bounded above by constant multiple of g(n) for all values of n, and if there exist a positive constant c and non negative integer n0 such that

- **t(n) ≤ c*g(n)** for all n ≥ n0.



- 1 Big-oh notation: $t(n) \in O(g(n))$.

Example 1:

Consider function t(n) = 2n + 2 and g(n) = $n^{2.}$ Then we have to find some constant c, so that f(n) ≤ c*g(n).

As t(n) = 2n + 2 and g(n) = $n^{2.}$ Then we find c for n=1 then

t(n) = 2n + 2
= 2(1) +2
t(n) = 4

And g(n) = n^2
= $(1)^2$
g(n) = 1

i.e t(n) > g(n)

if n = 2 then,

t(n) = 2n + 2
= 2(2) +2
t(n) = 6

And g(n) = n^2
= $(2)^2$
g(n) = 4

i.e t(n) > g(n)

if n = 3 then,

t(n) = 2n + 2

= 2(3) +2

t(n) = 8

And g(n) = n^2

= $(3)^2$

g(n) = 9

i.e t(n) < g(n) is true.

Hence we can conclude that for n> 2, we obtain

t(n) < g(n)

Thus always upper bound of existing time is obtained by big oh notation.

Omega Notation (Ω)

Omega notation is denoted by '**Ω**'.

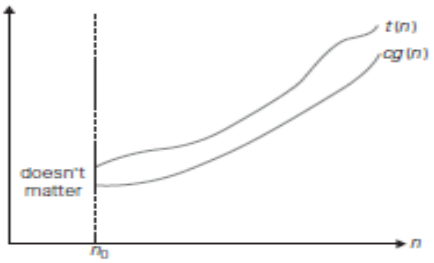
This notation is used is to represent the **lower bound of algorithm's** running time.

Using omega notation we can denote **shortest amount of time taken** by algorithm.

Definition

A function t(n) is said to be in Ω(g(n)) **(t(n) Є Ω(g(n))),** if t(n) is bounded below by constant multiple of g(n) for all values of n, and if there exist a positive constant c and non negative integer n0 such that

- **t(n) ≥ c*g(n)** for all n ≥ n0.



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

- **Example 1:**

Consider t(n)=$2n^2$ + 5 and g(n) = 7n

Then if n = 0

t(n) = 2 $(0)^2$ + 5

= 5

g(n) = 7(0)

= 0 i.e t(n) > g(n)

But if n = 1

t(n) = 2 $(1)^2$ + 5

= 7

g(n) = 7(1)

= 7 i.e t(n) = g(n)

But if n = 2

t(n) = 2 $(2)^2$ + 5

= 9

g(n) = 7(2)

= 12 i.e t(n) < g(n)

But if n = 3

t(n) = 2 $(3)^2$ + 5

= 18 + 5

= 23

g(n) = 7(3)

= 21 i.e t(n) > g(n)

Thus for n>3 we get t(n) > c * g(n).

It can be represented as

$2n^2 + 5 \in \Omega(n)$

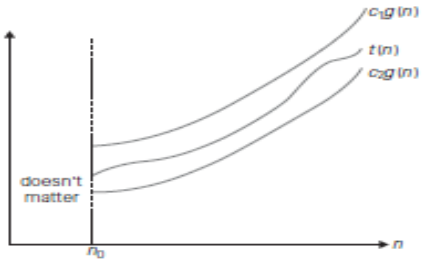
Theta Notation (Θ)

The theta notation is denoted by Θ. By this method the **running time is between upper bound and lower bound.**

Definition

A function t(n) is said to be in Θ(g(n)) **(t(n) Є Θ(g(n))),** if t(n) is bounded both above and below by constant multiple of g(n) for all values of n, and if there exist a positive constant c1 and c2 and non negative integer n0 such that

- **C2*g(n) ≤ t(n) ≤ c1*g(n)** for all n ≥ n0.



- 3 Big-theta notation: $t(n) \in \Theta(g(n))$.

Example 1:

If t(n) = 2n + 8 and g(n) = 7n, 5n

Where n ≥ 2

C2*g(n) ≤ t(n) ≤ c1*g(n) for all n ≥

Θ(g(n)) = O(g(n)) Ω(g(n))

(t(n) Є Θ(g(n)))

Similarly **t(n) = 2n + 8**

g(n) = 7n

g(n) = 5n

i.e **5n < 2n + 8 < 7n** for n ≥ 2

Here $c_2 = 5$ and $c_1 = 7$ with $n_0 = 2$

Little oh notation(o)

The function t(n) = o(g(n)), if O(*g*(*n*)) and t(n) <> Ω(*g*(n))

Example

t(n) = 3n+2

Where n>0, $3n+2 \leq 5\ n^2$

By definition of Big Oh

t(n) = Cg(n)

C = 5; $g(n) = n^2$

But t(n) = 3n+2 < > $\Omega(n^2)$

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

Useful property involving the Asymptotic notation:

The following property is useful in analyzing algorithms that comprise two consecutively executed parts.

Theorem

If $t_1(n)$ Є $O(g_1(n))$ and $t_2(n)$ Є $O(g_2(n))$ then,

$t_1(n) + t_2(n)$ Є $(\max \{g_1(n), g_2(n)\})$

Proof

Since $t_1(n)$ Є $O(g_1(n))$, there exist some constant C_1 and some non negative integer n_1 such that

$t_1(n) \leq C_1 (g_1(n))$ for all $n \geq n_1$

Since

$t_2(n) \in O(g_2(n))$

$t_2(n) \leq C_2 (g_2(n))$ for all $n \geq n_2$

Let us denote,

$C_3 = \max \{C_1, C_2\}$ and

Consider $n \geq \max \{n_1, n_2\}$, so that both the inequalities can be used.

The addition of two inequalities becomes,

$$t_1(n) + t_2(n) \leq C_1 (g_1(n)) + C_2 (g_2(n))$$
$$\leq C_3 (g_1(n)) + C_3 (g_2(n))$$
$$\leq C_3\, 2 \max\{g_1(n), (g_2(n))\}$$

Hence,

$$t_1(n) + t_2(n) \in O (\max \{g_1(n), g_2(n)\}),$$

with the constants C and n_0 required by the definition being $2C_3 = 2 \max (C_1, C_2)$ and $\max \{n_1, n_2\}$ respectively.

The property implies that the algorithms overall efficiency will be determined by the part with a larger order of growth.

(i.e.) its least efficient part is

$t_1(n) \in O(g_1(n))$
$t_2(n) \in O(g_2(n))$
} $t_1(n) + t_2(n) \in O (\max \{g_1(n), g_2(n)\})$

Using limits for comparing orders of growth

There are 3 principal cases,

$\lim_{n \to \infty} t(n)/g(n)$

- 0, Implies that (n) has a smaller order of growth than g(n)
- C, Implies that (n) has a same order of growth than g(n)
- ∞, Implies that (n) has a larger order of growth than g(n)

L' Hospital's rule.

$$\lim_{n \to \infty} t(n)/g(n) = \lim_{n \to \infty} t(n)/g(n)$$

Stirling's formula

$n! \approx \sqrt{2\pi n} \left(\frac{n}{k}\right)^n$ for large values of n.

Asymptotic Growth Rate

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

- $O(g(n))$: class of functions $f(n)$ that grow *no faster* than $g(n)$
- $\Omega(g(n))$: class of functions $f(n)$ that grow *at least as fast* as $g(n)$
- $\Theta(g(n))$: class of functions $f(n)$ that grow *at same rate* as $g(n)$

Example

1. Compare orders of growth of $\frac{1}{2}n(n-1)$ and n^2

$$\lim_{n\to\infty}\frac{\frac{1}{2}n(n-1)}{n^2}=\frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n}$$

$$=\frac{1}{2}\lim_{n\to\infty}\left(1-\frac{1}{n}\right)$$

$$=\frac{1}{2}$$

Since the limit is equal to a positive constant, the functions have the same order of growth or symbolically

$$\frac{1}{2}n(n-1)\in\theta(n^2)$$

2. Compare orders of growth of $\log_2 n$ and $\sqrt{n}$

$$\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}}=\lim_{n\to\infty}\frac{(\log_2 n)'}{(\sqrt{n})'}$$

$$=\lim_{n\to\infty}\frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{2}}}$$

$$=2\log_2 e\lim_{n\to\infty}\frac{\sqrt{n}}{n}$$

$$=0$$

Since the limit is equal to zero, $\log_2 e$ has a smaller order of growth than $\sqrt{n}$ or symbolically.

$$\log_2 e\in o(\sqrt{n}).$$

Little Oh notation is rarely used in analysis of algorithms.

3. Compare orders of growth n! and 2^n.

By using the Stirling's formula,

$$\lim_{n\to\infty}\frac{n!}{2^n}=\lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{e}{n}\right)}{2^n}$$

$$=\lim_{n\to\infty}\sqrt{2\pi n}\frac{n^n}{2^n e^n}$$

$$=\lim_{n\to\infty}\sqrt{2\pi n}\left(\frac{n}{2e}\right)^n$$

$$=\infty$$

Basic Asymptotic Efficiency Classes

Class	Name	Comments
1	Constant	Short of best-case efficiencies
$\log_n$	Logarithmic	Cutting a problem size by a constant factor
n	Linear	Algorithms that scan a list of size n.(eg sequential search)
n $\log_n$	n-log-n	Many divide and conquer algorithm
n^2	Quadratic	Efficiency of algorithm with two embedded loops.
n^3	Cubic	Efficiency of algorithm with three embedded loops.
2^n	Exponential	Generate all the subsets of an n element set.

n!	Factorial	Algorithm that generate all permutations of an n element set

5. Explain the Mathematical analysis for non-recursive algorithm or write an algorithm for determining the uniqueness of an array. Determine the time complexity of your algorithm. (Apr/May-2017) April/May 2019

General plan for analyzing efficiency of non-recursive algorithm

1. Decide the **input size** based on parameter n.
2. Identify the algorithm **basic operation(s)**.
3. Check whether the number of times the **basic operation is executed** depends on only on the size of the input.
4. Set up a **sum** expressing the number of times the algorithm basic operation is excited
5. Simplify the sum using standard formula and rules

Example 1: Problem for finding the value of the largest element in a list of n numbers

The pseudo code to solving the problem is

```
ALGORITHM MaxElement(A[0..n-1])
//Problem Description : This algorithm is for finding the
//maximum value element from the array
//Input:An array A[0..n-1] of real numbers
//Output: Returns the largest element from array
Maxval ← A[0]
For i ← 1 to n-1 do ──→ Searching the maximum element from an array
{
        If ( A[i]>max_value)then
        Maxval ← A[i]  ──→ If any value is large than current Max_ Value then set new Max_value by obtained larger value
}
        Return Max_value
```

Mathematical Analysis

Step 1: The input size is the number of elements in the array(ie.),n

Step 2 : The basic operation is comparison in loop for finding larger value There are two operations in the for loop

- ✓ Comparison operation a[i]->maxval
- ✓ Assignment operation maxval->a[i]

Step 3: The comparison is executed on each repetition of the loop. As the comparison is made for each value of n there is no need to find best case worst case and average case analysis.

Step 4: Let C(n) be the number of times the comparison is executed. The algorithm makes comparison each time the loop executes. That means with each new value of I the comparison is made. Hence for i= 1 to n – 1 times the comparison is made . therefore we can formulate C(n) as

C(n) = one comparison made for each value of i

Step 5 : let us simplify the sum

Thus $C_{(n)} = \sum_{i=1}^{n-1} 1$

$= n-1 \in \theta$ **(n)**

Using the rule $\sum_{i=1}^{n} 1 = n \in \theta(n)$

The frequently used two basic rules of sum manipulation are,

$$\sum_{i=1}^{u} Ca_i = C\sum_{i=1}^{u} a_i \quad \text{——————} \quad R1$$

$$\sum_{i=1}^{u} (a_i + b_i) = \sum_{i=1}^{u} a_I + \sum_{i=1}^{u} b_i \quad \text{——————} \quad R2$$

The two summation formulas are

1. $\sum_{i=1}^{n} 1 = u - l + 1$

Where $l \leq u$ are some lower and upper integer limits ——————— S1

2. $\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1+2+\ldots..+n$

$= n(n+1)/2$

$= 1/2n^2 \in \Theta(n^2)$ ——————— S2

Example 2: Element uniqueness problem-check whether all the element in the list are distinct April/May 2019

ALGORITHM UniqueElements(A[0..n-1])
//Checks whether all the elements in a given array are distinct
//Input :An array A[0..n-1]
//Output Returns 'true' if all elements in A are distinct and 'false'
//otherwise
for i ← to n-2 do
for j ←i+1 to n-1 do
if a[i] = a[j] then
return false
else
return true

If any two elements in the array are similar then return .false indicating that the array elements

Mathematical analysis

Step 1: Input size is n i.e total number of elements in the array A

Step 2: The basic iteration will be comparison of two elements . this operation the innermost operation in the loop . Hence if a[i] = a[j] then comparison will be the basic operation .

Step 3 : The number of comparisons made will depend upon the input n . but the algorithm will have worst case complexity if the same element is located at the end of the list. Hence the basic operation depends upon the input n and worst case

Worst case investigation

Step 4: The worst case input is an array for which the number od elements comparison $c_{worst}(n)$ is the largest among the size of the array.

There are two kinds of worst case inputs, They are

1.Arrays with no equal elements.

2.Arrays in which the last two elements are pair of equal elements.

For the above inputs, one comparison is made for each repetition of the inter most loop (ie) for each value of the loop's variable 'j' between its limits i+1 and n-1 and this is repeated limit for each values of the outer loop (ie) for each value of the loop's variable `i' between 0 and n-2. Accordingly,

C worst (n) = Outer loop × Inner loop

$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$

Step 5: now we will simplify **C worst as follows**

$= \sum_{i=0}^{n-1} [(n-1) - (i+1) + 1] \quad \Theta \sum_{i=k}^{n} [1 = n - k + 1 \text{ is the rule}]$

$= \sum_{i=0}^{n-2}[(n-1-i)]$

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

Now taking (n-1) as a common factor, we can write

$$= (n-1)\sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

This can be obtained using formula $\sum_{i=1}^{n} i = n(n+1)/2$

$$= (n-1)\sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

This can be obtained using formula

$$\sum_{i=1}^{n} 1 = (n-1+1) = n \text{ i.e when}$$

$$\sum_{i=0}^{n-2} 1 = (n-2)-0+1 = (n-1)$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2}$$

Solving this equation we will get

$= 2(n-1)(n-1) - (n-2)(n-1)/2$

$= (2(n^2 - 2n + 1) - (n^2 - 3n + 2))/2$

$= ((n^2 - n)/2$

$= 1/2\, n^2$

$\in \Theta(n^2)$

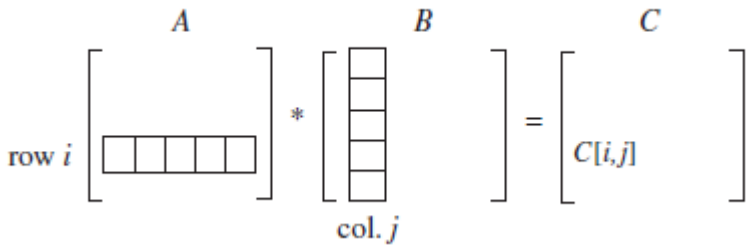
We can say that in the worst case the algorithm needs to compare all $n(n-1)/2$ distinct of its n elements.

Therefore $\mathbf{C_{worst}(n) = 1/2n^2 \in \Theta(n^2)}$

EXAMPLE 3 : Obtaining matrix multiplication

Given two *n* × *n* matrices *A* and *B*, find the time efficiency of the definition-based algorithm for computing their product *C* = *AB*, where A and B are n by n (n*n) matrices.

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 matrix *B*:



where $C[i, j] = A[i, 0]B[0, j] + \ldots + A[i, k]B[k, j] + \ldots + A[i, n-1]B[n-1, j]$ for every pair of indices $0 \le i, j \le n-1$.

$$C = \begin{pmatrix} a_{00} & a_{01} & a_{03} \\ 1 & 2 & 3 \\ a_{10} & a_{11} & a_{12} \\ 4 & 5 & 6 \end{pmatrix}_{2\times 3} \times \begin{pmatrix} b_{00} & b_{01} \\ 1 & 2 \\ b_{10} & b_{11} \\ 3 & 4 \\ b_{20} & b_{21} \\ 5 & 6 \end{pmatrix} \quad 3 \times 2$$

The formula for multiplication of the above two matrices is

$$\begin{pmatrix} a_{00}\times b_{00} + a_{01}\times b_{10} + a_{02}\times b_{20} & a_{00}\times b_{01} + a_{01}\times b_{11} + a_{02}\times b_{21} \end{pmatrix}$$

C =

$a_{10}\times b_{00} + a_{11}\times b_{10} + a_{12}\times b_{20}$ $a_{10}\times b_{01} + a_{11}\times b_{11} + a_{12}\times b_{21}$

$$C = \begin{bmatrix} 1\times 1 + 2\times 3 + 3\times 5 & 1\times 2 + 2\times 4 + 3\times 6 \\ 4\times 1 + 5\times 3 + 6\times 5 & 4\times 2 + 5\times 4 + 6\times 6 \end{bmatrix}$$

$$C = \begin{bmatrix} 22 & 28 \\ 49 & 64 \end{bmatrix}$$

Now the algorithm for matrix multiplication is

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 and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] * B[k, j]
return C
```

Mathematical analysis

Step 1: The input's size of above algorithm is simply order of matrices i.e n.

Step 2: The basic operation is in the innermost loop and which is

$C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$

There are two arithmetical operations in the innermost loop here

1. Multiplication
2. Addition

Step 3: The basic operation depends only upon input size. There are no best case, worst case and average case efficiencies**.** Hence now we will go for computing **sum.** There is just one multiplication which is repeated no each execution of innermost loop**. (a for loop using variable k).** Hence we will compute the efficiency for innermost loops.

Step 4: The sum can be denoted by M (n).

M(n) = outermost × inner loop × innermost loop (1 execution)

=[for loop using i]×[for loop using j]×[for loop using k]× (1 execution)

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

$$= \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} n$$

$$\sum_{j=1}^{n-1} = n$$

$$= \sum_{i=0}^{n-1} n^2$$

$M(n)= n^3$

Thus the simplified sum is $n3$. Thus the time complexity of matrix multiplication Θ (n3)

Running time of the Algorithm T(n)

The estimation of running time of the algorithm on a particular machine is calculated by using the product.

$T(n) \approx c_m M(n) = c_m n^3$

Where- c_m is the time of one multiplication on the machine in question.

We would get a more accurate estimate if we took into account the time spent on the additions, too:

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3$$

$$T(n) \approx c_m M(n) = c_m n^3$$

where c_m is the time of one multiplication on the machine in question. We would get a more accurate estimate if we took into account the time spent on the additions, too:

Time spend addition CA (n)

The time speed to perform the addition operation is given by

$$T(n) = c_a A(n) = c_a n^3$$

Where

c_a is the time taken to perform the one addition.

Hence the running time of the algorithm is given by

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3$$

The estimation differs only by the multiplication constants and not by the order of growth.

EXAMPLE 4:The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer.

ALGORITHM *Binary*(n)

```
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
count ← 1
while n > 1 do
    count ← count + 1
    n ← ⌊n/2⌋
return count
```

Mathematical analysis

Step 1: The input size is n i.e . The positive integer whose binary digit in binary representation needs to be checked.

Step 2 : The basic operation is denoted by while loop. And it is each time checking whether n > 1. The while loop will be executed for the number of time at which n>1 is true . it will be executed once more when n>1 is false . but when n>1 is false the statements inside while loop wont get executed.

Step 3: The value of n is halved on each repetition of the loop. Hence efficiency algorithm is equal to **log2 n**

Step 4: hence total number of times the while loop gets executed is **$[\log_2 n] + 1$**

Hence time complexity for counting number of bits of given number is **$\Theta(\log_2 n)$.** this indicates floor value of **$\log_2 n$**

6. Explain the Mathematical analysis for recursive algorithm. (Apr/May-2017) or

Discuss the steps in Mathematical analysis for recursive algorithms. Do the same for finding factorial of a number. Nov/Dec 2017 or solve the following recurrence equations using iterative method or tree Nov/Dec 2019

Discuss various methods used for mathematical analysis of recursive algorithms.May/June 2018

General plan for analyzing efficiency of recursive algorithms

1. Decide the input size based on parameter n .
2. Identify algorithms basic operations
3. Check how many times the basic operation is executed.
   To find whether the **execution of basic operation** depends upon the input size n. **determine worst, average , and best case** for input of size n. if the basic operation depends upon worst case average case and best case then that has to be analyzed separately.
4. Set up the **recurrence relation** with some initial condition and expressing the basic operation.
5. Solve the recurrence or at least determine the order of growth. While solving the recurrence we will use the **forward and backward substitution method**. And then correctness of formula can be proved with the help of **mathematical induction** method.

Example 1:Computing factorial of some number n.

To compare the factorial F(n)=n! for an arbitrary non negative integer

N! =1.2.3……(n-1).n

= (n-1)!.n ,for n≥1

0! =1

By definition F(n)=F(n-1)!.n

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

Mathematical Analysis:

Step 1: The algorithm's input size is n.

Step 2: The algorithm's basic operation in computing factorial is multiplication .

Step 3 : The recursive function call can be formulated as

According to the formula, F(n) is computed as

F(n) = F(n-1) * n, for n>0

And the number of execution is denoted by M(n).

The number of multiplication M(n) is computed as

M(n) = M(n-1) + 1, for n>0

To compute F(n-1) To multiply F(n-1) by n

M(n-1) multiplication are spent to compute F(n-1).

One more multiplication is needed to multiply the result by n.

Step 4: in step 3 the recurrence relation is obtained.

The equation is

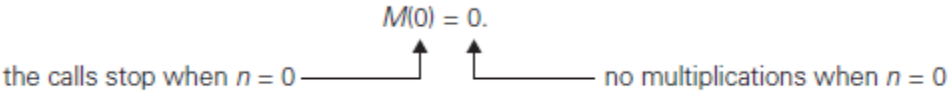
M(n)=M(n-1) +1, for n>0

Defines M(n)not explicitly(i.e.)as a function of n, but implicitly as function of its value at another point, namely n-1. These equations are called a**s recurrence relations or recurrences.**

- Recurrences relations play an important role in the analysis of algorithm and some area of applied mathematics.
- To solve a recurrence relation M(n)=M(n-1)+1 the formula for the sequence M(n) in terms of n only should be find.
- To determine the unique solution, an initial condition is needed that tells the value with which the sequence starts.
- The initial value is obtained from the condition if n=0 return 1 that makes the algorithm stops.

The condition, if n=0 return 1 tells 2 things

1. The recursive call stops when n=0 the smallest value for which the algorithm is executed. Hence M(n)=0.
2. When n=0 the algorithm performs no multiplication



Forward Substitution:

M(1) = M(0) +1

M(2) = M(1) + 1 = 1 + 1 =2

M(3) = M(2) + 1 = 2 + 1=3

The recurrence relation and the initial condition for the algorithm number of multiplication M(n) is

M(n)=M(n-1)+1,for n≤0, M(0)=0

Backward substitution:

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

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

Now M(n) becomes

M(n) = [M(n-2)+1]+1

= M(n-2) + 2

Substitute **M(n-2) =M(n-3)+1**

Now M(n) becomes

M(n)=[M(n-3)+1] + 2

= M(n-3) + 3

From the substitution method we can establish a general formula as :

M(n)= M(n-i) + i;

Since n=0, substitute i=n;

Now let us prove correctness of this formula using mathematical induction as follows

Proof

M(n) = n by using mathematical induction

Basis : let n = 0 then

M(n) =0

i.e M(0) = 0=n

Induction: if we assume M(n – 1) = n-1 then

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

= n-1 + 1

= n

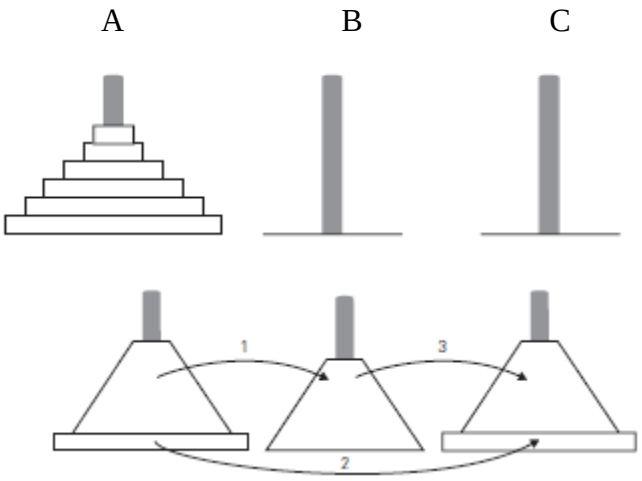
i.e M(n) = n Thus the time complexity of factorial function is Θ (n)

Give the general plan for Analyzing the time efficiency of Recursive Algorithms and use recurrence to find number of moves for Towers of Hanoi problem. May/June 2018

Example 2:Tower of Hanoi puzzle

- ✓ In this puzzle, there are n disks of different sizes, and three pegs.
- ✓ Initially all the disks are on the first peg in order if size, the largest on the bottom and the smallest on the top.
- ✓ The goal is to move all the disks from peg 1 to peg 3 using peg 2 as auxiliary.
- ✓ One disk should be moved at a time and do not place a larger disk on top of a smaller one.
- ✓ The following steps are used to move n>1 disks from peg 1 to peg 3, peg 2 as auxiliary.
    1. Move n-1 disks recursively from peg 1 to peg 3.(peg 2 as auxiliary).
    2. Move the largest disk directly from peg 1 to peg 3.
    3. Move n-1 disks recursively from peg 2 to peg 3.(peg 2 as auxiliary).

For example, if n=1 then the single disks is moved from source peg to destination peg directly.



General plan to tower of Hanoi problem

The input size is the number of disks “n”.

The algorithm basic operation is moving one disks at a time.

The number of moves M(n) depends only on n.

The recurrence equation is,

M(n)=M(n-1)+1+M(n-1),for n>1;

M(n)=2M(n-1)+1, for n>1;

The initial condition M(1)=1

Now the recurrence relation for number of moves is,

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

M(1)=1

The recurrence relation is solved by using backward substitution method

Backward substitution Method

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

Substitute

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

M(n)=2[2M(n-2)+1]+1

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

Substitute

M(n-2)=2M(n-3)+1

Now, M(n) becomes

M(n)=2^2[2M(n-3)+1]+2+1

M(n)=2^3[M(n-3)+2^2+2+1

Hence after I substitution M(n) becomes

M(n)=2^iM(n-i)+2^{i-1}+2^{i-2}+2^{i-3}+…….2+1

=2^iM(n-i)+2^i-1

Therefore te general formula is **2^iM(n-i)+2^i-1**

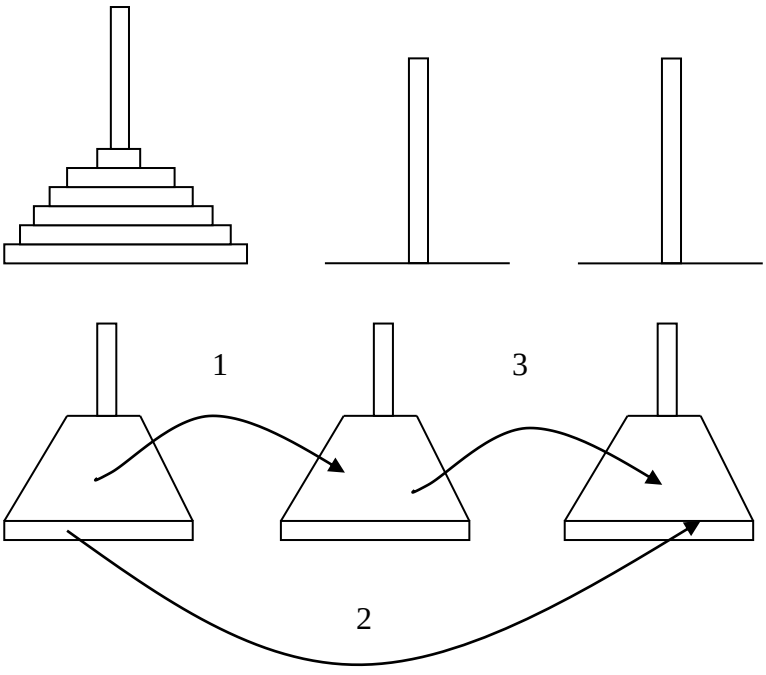


Fig. recursive solution to the Tower of Hanoi puzzle

Solution to recurrence relation is

Since the initial condition is n=1 becomes i=n-1.

The recurrence relation is

$$M(n)=2^iM(n-i)+2^i-1 \quad \text{.....(1)}$$

Substitute I=n-1 in (1)

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

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

$$=2^{n-}1+2^{n-1}-1$$

$$=2^n-1$$

$M(n)= 2^n-1$ Thus this is an exponential algorithm, It runs unimaginably long time for moderate values of n.

Example 3 :To find the number of binary digits in binary representation

```
Algorithm BinRec(n)
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if n=1
    return 1
else
    return BinRec([n/2])+1
```

Recurrence and Initial Condition

A Recurrence for the number of addition A(n) made by the algorithm is the number of addition made in computing BinRec([n/2]) is A([n/2]) plus one more addition is made

Thus recurrence is

A(n)=A([n/2])+1,for $\geq$n

A(n)->number of addition made by the algorithm

A([n/2])->number of addition made to compute A9[n/2])

The recursive call end when n is equal to 1 and no addition is made.

The initial condition is **A(1) = 0**

To solve the recurrence, backward substitutions cannot be used.The reason is the presence on [n/2] in the functions argument and the value of n is not power of 2.

A theorem called **Smoothness rule** is used to solve the recurrence.

The standard approach for solving such recurrence is to solve it only for $n = 2^k$.

The order of growth observed for $n = 2^k$ gives a correct answer about the order of growth of all values of n.

$n = 2^k$ takes the form

$A(2^k) = A(2^{k-1}) + 1$ for $k > 0$,

$A(2^0) = 0$.

Now, backward substitutions can be applied.

Backward Substitution Method

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

substitute $A(2^{k-1}) = A(2^{k-2}) + 1$

$= [A(2^{k-2}) + 1] + 1$

$= A(2^{k-2}) + 2$

substitute $A(2^{k-2}) = A(2^{k-3}) + 1$

$= [A(2^{k-3}) + 1] + 2$

$= A(2^{k-3}) + 3$ ... ...

After i iteration

$A(2^k) = A(2^{k-i}) + i$

$= A(2^{k-k}) + k$

$= A(2^0) + k$

$= A(1) + k$

Thus, we end up with

$A(2^k) = A(1) + k = k$

After returning to the original variable

$n = 2k$ and hence $k = \log_2 n$,

$A(n) = \log_2 n \in \Theta(\log n)$

Example 4: Fibonacci series

A sequence of Fibonacci numbers is 0,1,1,2,3,5,8,13,21,34……..

The Fibonacci sequence can be defined by the simple recurrence

$F(n)=F(n-1)+F(n-2)$,for $n>1$……1

The two initial conditions are

$F(0)=0$
$F(1)=1$

Explicit formula for the n^{th} Fibonacci number

Backward substitution method is not used to solve the recurrence $F(n)=F(n-1)+F(n-2)$,for $n>1$,because which fails to produce easily discernible pattern.

So, the theorem that describes solution to a homogeneous second order linear recurrence with constant coefficient is used to solve the problem.

The homogenous with constant coefficient is

$$ax(n)+bx(n-1)+cx(n-2)=0 \quad \ldots\ldots\ldots\ldots\ldots(2)$$

Where,

a,b,c are fixed real numbers called the coefficients of recurrence and $a \neq 0$

x(n) is the unknown sequence to be found

The characteristics equation of the recurrence equation is

$$\mathbf{Ar^2+br+c=0} \quad \ldots\ldots\ldots\ldots\ldots\ldots(3)$$

The recurrence relation can be written as

$$\mathbf{F(n)-F(n-1)-F(n-2)=0} \ldots\ldots\ldots\ldots(4)$$

The characteristics equation for (4)

$$\mathbf{r^2-r-1=0}$$

The roots are

$$\mathbf{R_{1,2}}=\frac{-1 \pm \sqrt{1-4(1)}}{2}$$

$$\mathbf{R_{1,2}}=\frac{1 \pm \sqrt{5}}{2}$$

$$\mathbf{R_1}=\frac{1+\sqrt{5}}{2}$$

$$\mathbf{R_2}=\frac{1-\sqrt{5}}{2}$$

The characteristics equation has two distinct real roots.

Now the recurrence relation is

$$\mathbf{X(n)=\alpha r_1^n+\beta r_2^n} \quad \ldots\ldots\ldots(5)$$

Substitute r1 and r2 in (5),

$$\mathbf{F(n)= \alpha\left(\frac{1+\sqrt{5}}{2}\right)^n+\beta\left(\frac{1+\sqrt{5}}{2}\right)^n} \quad \ldots\ldots\ldots(6)$$

Now substitute the value of f(0) and F(1) in equation(6)

$$\mathbf{F(0) = \alpha\left(\frac{1+\sqrt{5}}{2}\right)^0+\beta\left(\frac{1+\sqrt{5}}{2}\right)^0 =0} \quad \ldots\ldots\ldots(7)$$

$$\mathbf{F(1)= \alpha\left(\frac{1+\sqrt{5}}{2}\right)^1+\beta\left(\frac{1+\sqrt{5}}{2}\right)^1 =0} \quad \ldots\ldots\ldots(8)$$

By solving equation (7) and (8),the linear equation in two unknown α and β

$$\alpha+\beta=0$$

$$\alpha\left(\frac{1+\sqrt{5}}{2}\right)+\beta\left(\frac{1+\sqrt{5}}{2}\right)=0 \quad \ldots\ldots\ldots(11)$$

(11)-(10) gives

$$\left(\frac{1+\sqrt{5}}{2}\right)\beta-\left(\frac{1+\sqrt{5}}{2}\right)\beta=-1$$

$$\frac{\beta}{2}+\frac{\sqrt{5}}{2}\beta-\frac{\beta}{2}+\frac{\sqrt{5}}{2}\beta=-1$$

$$2\,\frac{\sqrt{5}}{2}\beta=-1$$

$$\beta=-\frac{\sqrt{5}}{2}$$

Substitute $\beta=-\frac{\sqrt{5}}{2}$ in (9)

$$\alpha+\beta=0$$

$$\alpha-\frac{1}{\sqrt{5}}=0$$

$$\alpha=\frac{1}{\sqrt{5}}\quad \beta=-\frac{\sqrt{5}}{2}$$

Substitute the value of α and β in equation (6)

$$F(n)=\frac{1}{\sqrt{5}}\left[\frac{1+\sqrt{5}}{2}\right]^n-\frac{1}{\sqrt{5}}\left[\frac{1+\sqrt{5}}{2}\right]^n$$

$$F(n)=\frac{1}{\sqrt{5}}[\phi n-\phi\hat{}n]$$

Where

$$\Phi=\frac{1+\sqrt{5}}{2}$$

$$\Phi=1.61803$$

$$\Phi\hat{}=-\frac{1}{\Phi}$$

$$\Phi\hat{}=-0.61803$$

The constant Φ is known as, Golden Ratio.

The value of Φ^ is lies between -1 and 0.

When n goes to infinity, Φ^ gets infinitely small value. So, it can be omitted.

Therefore $F(n)=\frac{1}{\sqrt{5}}\Phi^n$

So, for every non negative n, $F(n)=\frac{1}{\sqrt{5}}\Phi^n$ is rounded to the nearest integer.

Algorithm for computing Fibonacci numbers

First method

Algorithm F(n)

//Computes the nth Fibonacci number recursively by using its definition.

//Input: A nonnegative integer n

//Output: The nth Fibonacci number

```
if n<1
        return n
```

```
    Else
        return F(n-1)+(n-2)
```

the algorithm's basic operation is addition.

Let A(n) is the number of additions performed by the algorithm to compute F(n).

The number of additions needed to compute F(n-1) is A(n-1) and the number of additions needed to compute F(n-2) is A(n-2).

The algorithm needs one more addition to compute the sum of A(n-1) and A(n-2).

Thus the recurrence for A(n) is

A(n)=A(n-1) + A(n-2)+1, for n>1

A(0)=0

A(1)=0

The recurrence A(n)-A(n-1)-A(n-2)=1 is same as F(n)-F(n-1)-F(n-2)=0, but its right hand side not equal to zero. These recurrences are called **inhomogeneous recurrences**.

General techniques are used to solve inhomogeneous recurrences.

The inhomogeneous recurrences is converted into homogeneous recurrence by rewriting the in homogeneous recurrence as,**A(n)+1]-[A(n-1)+1]-[A(n-2)+1]=0 (14)**

Now substitute, **B(n)=A(n)+1**

Now (14) becomes, **B(n)-B(n-1)-B(n-2)=0**

B(0)=0

B(1)=1

Here **B(n)=F(n+1)**

Since **B(n)=A(n)+1**

B(n-1)=A(n)

So **A(n)=B(n)-1**

Substitute F(n+1)-1 ……(15)

We know that

F(n)=$\frac{1}{\sqrt{5}}(\varnothing^n - \varnothing^n)$

F(n+1)=$\frac{1}{\sqrt{5}}(\varnothing^{n+1} - \varnothing^{n+1})$ ……(16)

Substitute (16) in (15)

A(n)=$\frac{1}{\sqrt{5}}(\varnothing^{n+1} - \varnothing^{n+1})$**-1**

Hence

A(n)€ $\Theta(\varnothing^n)$

The poor efficiency class of algorithm could be anticipated from the class of recurrence

The reason behind the algorithm inefficiency can be traced by looking at the tree of recursive calls n=6

The same values of the function are evaluated again and again which is extremely inefficiently.

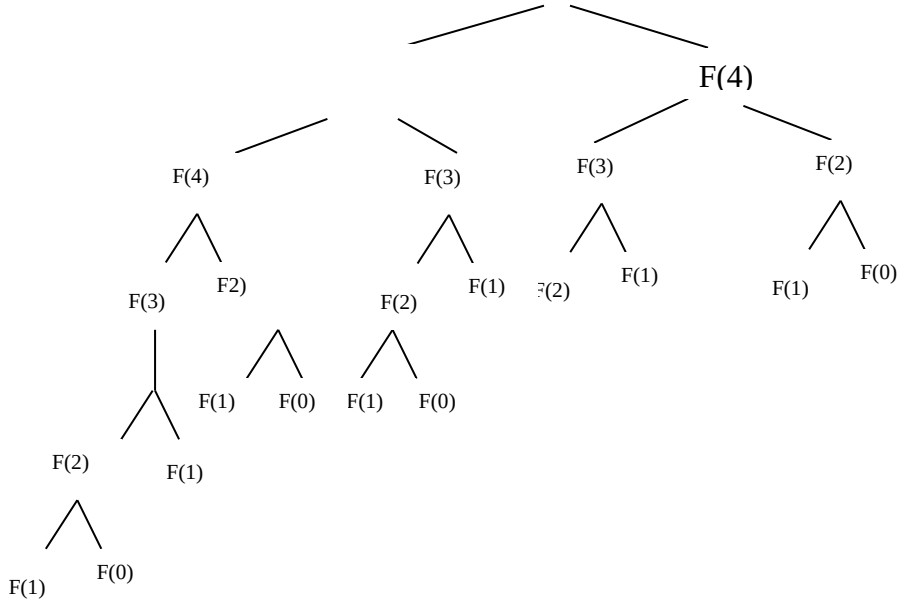


Fig Tree of recursive calls for computing the Fibonacci number for n = 6

7. Find the time complexity and space complexity of the following problems. Factorial using recursion and compute the nth Fibonacci number using iterative statements. *Dec 2012*

8.Solve the following recurrence relations: or solve the following recurrence equation:

T(n)=T(n/2)+1,where n=2^k for all k>=0

T(n)= T(n/3)+ T(2n/3)+cn,where 'c' is a constant and 'n' is the input size.

Dec 2012 **April/May 2019**

1. T(n)= $\begin{cases} 2T(n/2)+3 & n>2 \\ 2 & n=2 \end{cases}$

T(n)=2T(n/2)+3

=2{(2T(n/2)+3)/2}+3

=2{(2T(n/4)+3/2}+3

....

=4T(n/4)+6

= 4{(2T(n/2)+3)/4}+6

.....

=8T(n/8)+9

$=2^kT(n/2^k)+3n$

T(n)=nlogn+3n

Time complexity=o(nlog n)

2. T(n)= $\begin{cases} 2T(n/2)+cn & n>1 \\ a & n=1 \end{cases}$ where a and c constants

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

=2{(2T(n/2)+cn)/2}+cn

=2{(2T(n/4)+cn/2}+cn

=4T(n/4)+cn+cn

= 4{(2T(n/8)+cn/4}+ cn+cn

=8T(n/8)+ cn+cn+cn

$=2^kT(n/2^k)+k(cn)$

T(n)=nlogn+ k(cn)

Time complexity=o(nlog n)

8. Show the following equalities are correct *June 2013*

i. $5n^2-6n = \Phi(n^2)$

ii. $n!=O(n^n)$

iii. $n^3+10^6n^2=\Theta(n^3)$

iv. $2n^22^n + n \log n = \Theta(n^22^n)$

i. $5n^2-6n = \Phi(n^2)$ =>higest order of grouth is n^2

ii. $n!=O(n^n)$ =>higest order of grouth O(n)

iii. $n^3+10^6n^2=\Theta(n^3)$ =>higest order of grouth $O(n^3)$

iv. $2n^22^n + n \log n = \Theta(n^22^n)$=>higest order of grouth $O(n^2)$

Nov 2010

9. Prove that for any two functions f(n) and g(n), we have f(n)-> Θ(g(n)) if and only if f(n) -> O(g(n)) and f(n) ->Ω(g(n)) *Nov 2010*

Given function:

f(n) and g(n)

f(n)= O(g(n)) when $f(n) \le C_1g(n)$ for all $n \ge n_0$--------(1)

f(n)= Ω(g(n)) when $f(n) \ge C_2g(n)$ for all $n \ge n_0$--------(2)

from (1) and (2)

$C_2\, g(n) \le f(n) \le C_{1g(n)}$ for all $n \ge n_0$ -------(3)

(i.e) Θ(g(n)) = O(g(n))Ω(g(n))

From (3) f(n) = Θ(g(n)) hence proved

10. (a)If you have to solve the searching problem for a list of n numbers, how can you take

advantage of the fact that the list is known to be sorted? Give separate answers for lists represented as arrays lists represented as linked lists. (AU april/may 2015)

For a sorted array do a binary search to divide the array in half for each query, thus O(lg n). If the list is linked you must you do a linear search which is O(n), unless you use a linked binary search tree, which is O(lg n)

11. The best-case analysis is not as important as the worst-case analysis of an algorithm". Yes or No ? Justify your answer with the help of an example. (April/May 2021)

The Best Case analysis is bogus. **Guaranteeing a lower bound on an algorithm doesn't provide any information as** in the worst case, an algorithm may take years to run. For some algorithms, all the cases are asymptotically the same, i.e., there are no worst and best cases. For example, Merge Sort.

11.Derive the worst case analysis of merge sort using suitable illustration (AU april/may 2015)

Efficiency of Merge Sort

- In merge sort algorithm the two recursive calls are made. Each recursive call focuses on n/2 elements of the list .
- After two recursive calls one call is made to combine two sublist i.e to merge all n elements.
- Hence we can write recurrence relation as

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

T(n/2) = Time taken by left sublist

T(n/2) = time taken by right sublist

T(n) = time taken for combining two sublists

where n> 1 T (1) = 0

The time complexity of merge sort can be calculated using two methods

- Master theorem
- Substitution method

Master theorem Let , the recurrence relation for merge sort is

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

Let T(n) = aT(n/b) + f(n) be a recurrence relation

i.e. T(n) = 2T(n/2) + cn ------- (1)

T(1) = 0 ---------- (2)

As per master theorem $T(n) = \Theta (n^d \log n)$ if a = b

As equation (1),a =2 , b = 2 and f(n) = cn and $a = b^d$ i.e $2 = 2^`$

This case gives us , $T (n) = \Theta (n \log 2 n)$

Hence the average and worst case time complexity of merge sort is

$$C_{worst}(n) = (n \log 2 n)$$

Substitution method Let, the recurrence relation for merge sort be

T(n) = T(n/2) + T(n/2) + cn for n>1

i.e. T(n) = 2T(n/2) + cn for n>1 ------- (3)

$T(1) = 0$ -------(4)

Let us apply substitution on equation (3) .

Assume $n=2^k$

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

$T(n) = 2T(2^k/2) + c.2^k$

$T(2^k) = 2T(2^k-1) + c.2^k$

If k = k-1 then,

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

$T(2^k) = 2[2T(2^{k-2}) + c.2^k -1] + c.2^k$

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

$T(2^k) = 2^2 T(2^{k-2}) + 2.c.2^k /2 + c.2^k$

$T(2^k) = 2^2 T(2^{k-2}) + c.2^k + c.2^k$

$T(2^k) = 2^2 T(2^{k-2}) + 2c .2^k$

Similarly we can write,

$T(2^k) = 2^3 T(2^{k-3}) + 3c .2^k$

$T(2^k) = 2^4 T(2^{k-4}) + 4c .2^k$

…..

….

$T(2^k) = 2^k T(2^{k-k}) + k.c.2^k$

$T(2^k) = 2^k T(2^0) + k.c.2^k$

$T(2^k) = 2^k T(1) + k.c.2^k$ -------- (5)

But as per equation (4), T(1) =0

There equation (5) becomes ,

$T(2^k) = 2^k .0 +. k. c . 2^k$

$T(2^k) = k. c . 2^k$

But we assumed $n=2^k$, taking logarithm on both sides.i.e. log 2 n = k

Therefore T(n) = log 2 n. cn

Therefore T (n) =Θ (n log2 n)

Hence the average and worst case time complexity of merge sort is

$C_{worst}(n) = (n \log_2 n)$

Time complexity of merge sort

Best case	**Average case**	**Worst case**
Θ (n log2 n)	Θ (n log2 n)	Θ (n log2 n)

12.write Insertion sort algorithm and estimate its running time.

- ✓ Like selection sort, insertion sort loops over the indices of the array. It just calls `insert` on the elements at indices `1,2,3,…,n−1`. Just as each call to `indexOfMinimum` took an amount of time that depended on the size of the sorted subarray, so does each call to `insert`. Actually, the word "does" in the previous sentence should be "can," and we'll see why.
- ✓ Let's take a situation where we call `insert` and the value being inserted into a subarray is less than every element in the subarray.
- ✓ For example, if we're inserting 0 into the subarray [2, 3, 5, 7, 11], then every element in the subarray has to slide over one position to the right. So, in general, if we're inserting into a subarray with `k` elements, all `k` might have to slide over by one position.
- ✓ Rather than counting exactly how many lines of code we need to test an element against a key and slide the element, let's agree that it's a constant number of lines; let's call that constant `ccc`. Therefore, it could take up to `c·k` lines to insert into a subarray of `k` elements.
- ✓ Suppose that upon every call to `insert`, the value being inserted is less than every element in the subarray to its left. When we call `insert` the first time, `k=1`. The second time, `k=2`. The third time, `k=3`. And so on, up through the last time, when `k=n−1`.

Therefore, the total time spent inserting into sorted subarrays is **$c\cdot1+c\cdot2+c\cdot3+\cdots c\cdot(n-1)=c\cdot(1+2+3+\cdots+(n-1))$**

That sum is an arithmetic series, except that it goes up to `n−1n−1n−1` rather than `nnn`. Using our formula for arithmetic series, we get that the total time spent inserting into sorted subarrays is

$c\cdot(n-1+1)((n-1)/2)=cn2/2-cn/2.$

Using big-Θ notation, we discard the low-order term `cn/2` and the constant factors `c` and 1/2, getting the result that the running time of insertion sort, in this case, is `Θ(n2)`.

Can insertion sort take *less* than `Θ(n2)` time? The answer is yes. Suppose we have the array [2, 3, 5, 7, 11], where the sorted subarray is the first four elements, and we're inserting the value 11. Upon the first test, we find that 11 is greater than 7, and so no elements in the subarray need to slide over to the right.

- ✓ Then this call of `insert` takes just constant time. Suppose that *every* call of `insert` takes constant time. Because there are `n−1` calls to `insert`, if each call takes time that is some constant `ccc`, then the total time for insertion sort is `c·(n−1)` which is `Θ(n)`, not `Θ(n2)`.
- ✓ Can either of these situations occur? Can each call to `insert` cause every element in the subarray to slide one position to the right? Can each call to `insert` cause no elements to slide? The answer is yes to both questions.
- ✓ A call to `insert` causes every element to slide over if the key being inserted is less than every element to its left. So, if every element is less than every element to its left, the running time of insertion sort is `Θ(n2)`.
- ✓ What would it mean for every element to be less than the element to its left? The array would have to start out in *reverse* sorted order, such as [11, 7, 5, 3, 2]. So a reverse-sorted array is the worst case for insertion sort.
- ✓ How about the opposite case? A call to `insert` causes no elements to slide over if the key being inserted is greater than or equal to every element to its left. So, if every element is greater than or equal to every element to its left, the running time of insertion sort is `Θ(n)`.
- ✓ This situation occurs if the array starts out already sorted, and so an already-sorted array is the best case for insertion sort.

What else can we say about the running time of insertion sort? Suppose that the array starts out in a random order. Then, on average, we'd expect that each element is less than half the elements to its left.

- ✓ In this case, on average, a call to `insert` on a subarray of `k` elements would slide `k/2` of them. The running time would be half of the worst-case running time. But in asymptotic notation, where constant coefficients don't matter, the running time in the average case would still be $\Theta(n^2)$, just like the worst case.
- ✓ What if you knew that the array was "almost sorted": every element starts out at most some constant number of positions, say 17, from where it's supposed to be when sorted?
- ✓ Then each call to `insert` slides at most 17 elements, and the time for one call of `insert` on a subarray of `kkk` elements would be at most $17 \cdot c$. Over all $n-1$ calls to `insert`, the running time would be $17 \cdot c \cdot (n-1)$, which is $\Theta(n)$, just like the best case. So insertion sort is fast when given an almost-sorted array.

To sum up the running times for insertion sort:

- **Worst case: $\Theta(n^2)$.**
- **Best case: $\Theta(n)$.**
- **Average case for a random array: $\Theta(n^2)$.**
- **"Almost sorted" case: $\Theta(n)$.**

f you had to make a blanket statement that applies to all cases of insertion sort, you would have to say that it runs in $O(n^2)$ time. You cannot say that it runs in $\Theta(n^2)$ time in all cases, since the best case runs in $\Theta(n)$ time. And you cannot say that it runs in $\Theta(n)$ time in all cases, since the worst-case running time is $\Theta(n^2)$.

13.Show how to implement a stack using two queues.Analyze the running time of the stack operations.

Show how to implement a stack using two queues. Analyze the running time of the stack operations.

- The pseudocode is as follows.

```
public class TwoQueueStack
{
  Queue q1;
  Queue q2;
  int flag = 0;
  // 0: the stack is empty;
  // 1: all data are stored in q1;
  // 2: all data are stores in q2;

  // always push into the empty queue, then move all data from
  // the other queue to the current queue.
  // always pop element from the queue containing data.
  // suppose q1 and q2 are implemented by linked list. The queues never
  // get full.

  public void push(Object e)
  {
    switch(flag)
    {
      case 0: q1.enQueue(e);
              flag = 1;
              break;
      case 1: q2.enQueue(e);
              while (!q1.isEmpty())
                 q2.enQueue(q1.deQueue());
              flag = 2;
              break;
      case 2: q1.enQueue(e);
              while (!q2.isEmpty())
                 q1.enQueue(q2.deQueue());
              flag = 1;
              break;
      default: error 'illegal state';
    }
  }


 public Object pop()
 {
   switch(flag)
   {
     case 0: error 'underflow -- stack is empty, can't pop';
     case 1: retElement = q1.deQueue();
             if (q1.isEmpty())
                flag = 0;
             return retElement;
     case 2: retElement = q2.deQueue();
             if (q2.isEmpty())




                       flag = 0;
                  return retElement;
           default: error 'illegal state';
         }
       }
     }
```

- The running time for `push` operation is $O(n)$. The running time for `pop` operation is $\Theta(1)$.

14. find the closest asymptotic tight bound by solving the recurrence equation *T(n)=8T(n/2)+n2* with *(T(1)=1)* using recursion tree method.[Assume that T(1)ЄΘ(1)]

Example: Solve $T(n) = 8T(n/2) + n^2 \quad (T(1) = 1)$

$$\begin{aligned}
T(n) &= n^2 + 8T(n/2) \\
&= n^2 + 8(8T(\frac{n}{2^2}) + (\frac{n}{2})^2) \\
&= n^2 + 8^2T(\frac{n}{2^2}) + 8(\frac{n^2}{4})) \\
&= n^2 + 2n^2 + 8^2T(\frac{n}{2^2}) \\
&= n^2 + 2n^2 + 8^2(8T(\frac{n}{2^3}) + (\frac{n}{2^2})^2) \\
&= n^2 + 2n^2 + 8^3T(\frac{n}{2^3}) + 8^2(\frac{n^2}{4^2})) \\
&= n^2 + 2n^2 + 2^2n^2 + 8^3T(\frac{n}{2^3}) \\
&= \dots \\
&= n^2 + 2n^2 + 2^2n^2 + 2^3n^2 + 2^4n^2 + \dots
\end{aligned}$$

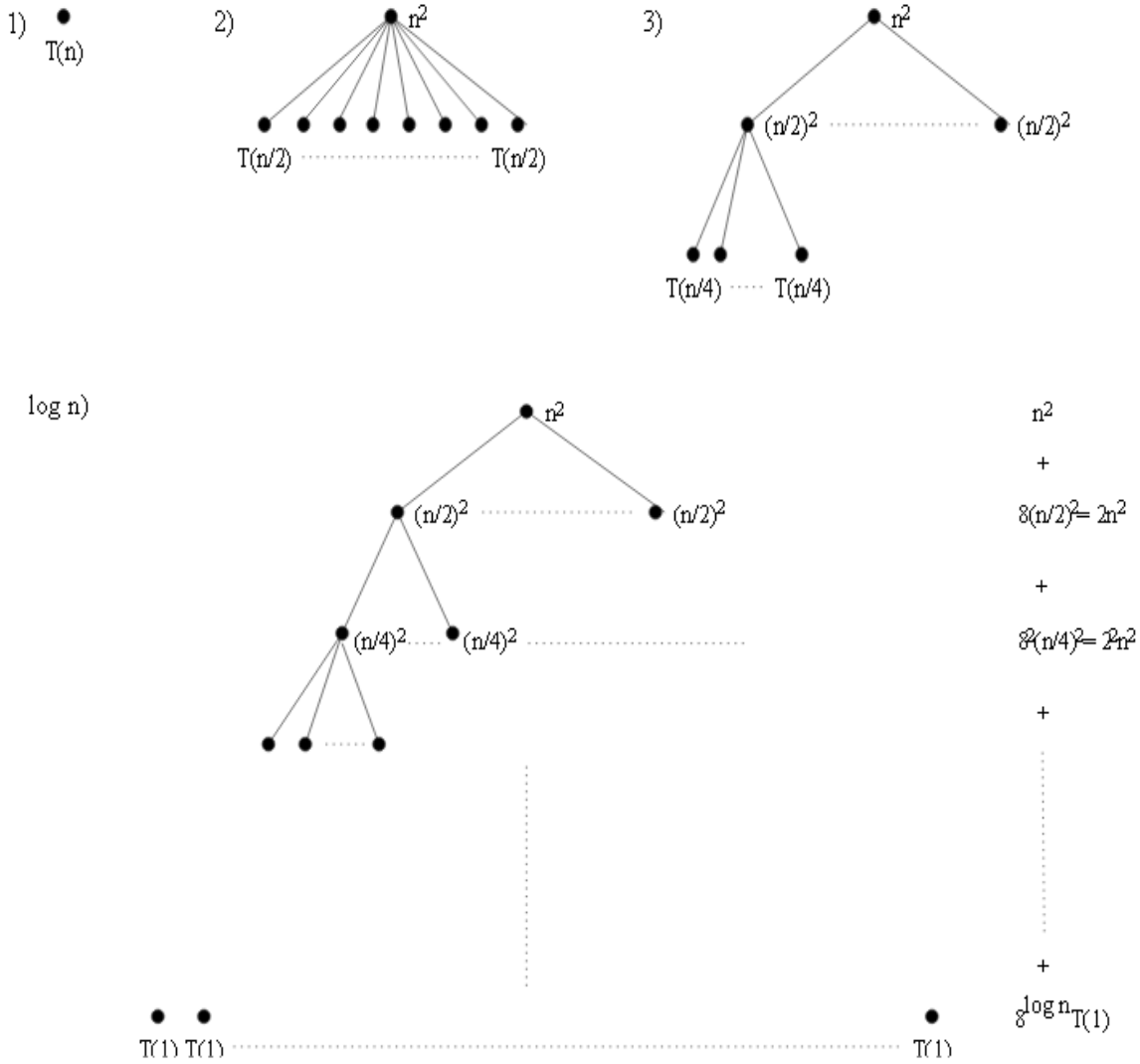
- Recursion depth: How long (how many iterations) it takes until the subproblem has constant size? i times where $\frac{n}{2^i} = 1 \Rightarrow i = \log n$
- What is the last term? $8^iT(1) = 8^{\log n}$

$$\begin{aligned}
T(n) &= n^2 + 2n^2 + 2^2n^2 + 2^3n^2 + 2^4n^2 + \dots + 2^{\log n - 1}n^2 + 8^{\log n} \\
&= \sum_{k=0}^{\log n - 1} 2^k n^2 + 8^{\log n} \\
&= n^2 \sum_{k=0}^{\log n - 1} 2^k + (2^3)^{\log n}
\end{aligned}$$

- Now $\sum_{k=0}^{\log n - 1} 2^k$ is a geometric sum so we have $\sum_{k=0}^{\log n - 1} 2^k = \Theta(2^{\log n - 1}) = \Theta(n)$
- $(2^3)^{\log n} = (2^{\log n})^3 = n^3$

$$\begin{aligned}
T(n) &= n^2 \cdot \Theta(n) + n^3 \\
&= \Theta(n^3)
\end{aligned}$$

- we draw out the recursion tree with cost of single call in each node—running time is sum of costs in all nodes
- if you are careful drawing the recursion tree and summing up the costs, the recursion tree is a direct proof for the solution of the recurrence, just like iteration and substitution
- Example: $T(n) = 8T(n/2) + n^2 \quad (T(1) = 1)$



$$T(n) = n^2 + 2n^2 + 2^2n^2 + 2^3n^2 + 2^4n^2 + \ldots + 2^{\log n - 1}n^2 + 8^{\log n}$$

15.Derive a loose bound on the following equation: $F(x)=35\ x^8 -22x^7+14x^5 -2x^4 -4x^2+x-15$

$f(x) = 35x^8 - 22x^7 + 14x^5 - 2x^4 - 4x^2 + x - 15$

Solution : Let, f(n) and g(n) are two non-negative functions.

Let c be some constant.

The equation

$f(n) \leq c * g(n)$

then $f(n) \in O(g(n)$ with tight bound.

But if $f(n) < c * g(n)$

then $f(n) \in O(g(n)$ with loose bound.

Consider the function

$f(x) = 35x^8 - 22x^7 + 14x^5 - 2x^4 - 4x^2 + x - 15$

and $g(x) = x^8$

If $x = 1$, then

$f(x) = 35(1)^8 - 22(1)^7 + 14(1)^5 - 2(1)^4 - 4(1)^2 + 1 - 15$

$f(x) = 7$

$g(x) = x^8 = (1)^8$

If we assume $c = 35$ then

We will always get

$f(x) < g(x)$ for $x \geq 1$

16.Solve the recurrence relations

X(n) =x(n-1) +5 for n > 1 x(1)=0

X(n) =3x(n-1) for n > 1 x(1)=4

X(n) =x(n-1) +n for n > 0 x(0)=0

X(n) =x(n/2) +n for n > 1 x(1)=1 (solve for n= 2^k)

X(n) =x(n/3) +1 for n > 1 x(1)=1 (solve for n= 3^k)

X(n) =x(n-1) +5 for n > 1 x(1)=0

X(1)=0
If n=2
X(2)=x(2-1)+5
=x(1)+5
=0+5
=5
If n=3
X(3)=x(3-1)+5

=x(2)+5
=5+5
=10

If n=4
X(4)=x(4-1)+5
=x(3)+5
=10+5
=15.....

17.Use the most appropriate notation to indicate the time efficiency class of sequential search algorithm in the worst case,best case and the average case.

Solution : Sequential search

"Given a target value and a random list of values, find the location of the target in the list, if it occurs, by checking each value in the list in turn"

```
get (NameList, PhoneList, Name)
i = 1
N = length(NameList)
Found = FALSE
while ( (not Found) and (i <= N) ) {
   if ( Name == NameList[i] ) {
      print (Name, "'s phone number is ", PhoneList[i])
      Found = TRUE
   }
   i = i+1
}
if ( not Found ) { print (Name, "'s phone number not found!") }
```

Central unit of work: operations that occur most frequently

Central unit of work in sequential search:

Comparison of target Name to each name in the list

Also add 1 to i

Typical iteration: two steps (one comparison, one addition)

Given a large input list:

Best case: smallest amount of work algorithm must do

Worst case: greatest amount of work algorithm must do

Average case: depends on likelihood of different scenarios occurring

- **Best case:** target found with the first comparison (**1 iteration**)
- **Worst case**: target never found or last value (*N* iterations)
- **Average case:** if each value is equally likely to be searched, work done varies from 1 to *N*, on average ***N/2*** **iterations**

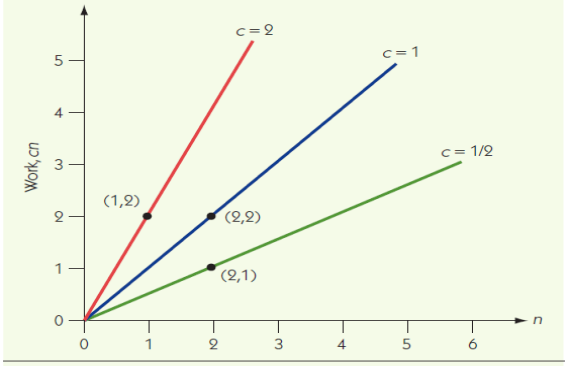
Sequential search worst case (N) grows linearly in the size of the problem 2N steps (one comparison and one addition per loop) Also some initialization steps...

On the last iteration, we may print something...After the loop, we test and maybe print...

To simplify analysis, disregard the "negligible" steps (which don't happen as often), and ignore the coefficient in 2N Just pay attention to the dominant term (N)

Order of magnitude O(N): the class of all linear functions (any algorithm that takes C1N + C2 steps for any constants C1 and C2)

FIGURE 3.4



Work = *cn* for various values of *c*

18.(i) Prove that if g(n) is Ω(f(n)) then f(n) is O(g(n)).May/June 2018

f(n) ∈ Ω(g(n)) ⟺ g(n) ∈ O(f(n))

Proof:

$O(f(n))=\{g:N\to N \mid \exists c,n0\in N \ \forall n\geq n0: g(n)\leq c\cdot f(n)\}$

$\Omega(g(n))=\{f:N\to N \mid \exists c,n0\in N \ \forall n\geq n0: f(n)\geq c\cdot g(n)\}$

Step 1/2: $f(n)\in\Omega(g(n)) \iff g(n)\in O(f(n))$

$\exists c,n0\in N\ \forall n\geq n0$: $f(n)\geq c\cdot g(n)\Rightarrow f(n)g(n)\geq c\Rightarrow 1g(n)\geq cf(n)\Rightarrow g(n)\leq 1c\cdot f(n)$

And this is exactly the definition of O(f(n)).

Step 2/2: $f(n)\in\Omega(g(n))\Leftarrow g(n)\in O(f(n))$

$\exists c,n0\in N\ \forall n\geq n0$: $g(n)\leq c\cdot f(n)\Rightarrow ...\Rightarrow f(n)\geq 1c\cdot g(n)$

Hence proved.

19. Explain briefly about Empirical Analysis of Algorithm.

The principal alternative to the mathematical analysis of an algorithm's efficiency is its empirical analysis. This approach implies steps spelled out in the following plan.

General Plan for the Empirical Analysis of Algorithm Time Efficiency

1. Understand the experiment's purpose.

2. Decide on the efficiency metric *M* to be measured and the measurement unit(an operation count vs. a time unit).

3. Decide on characteristics of the input sample (its range, size, and so on).

4. Prepare a program implementing the algorithm for the experimentation.

5. Generate a sample of inputs.

6. Run the algorithm (or algorithms) on the sample's inputs and record the data observed.

7. Analyze the data obtained.

1. **Purpose:**
   - To ensure theoretical assertion about the algorithm's efficiency
   - comparing the efficiency of several algorithms for solving the same problem or different implementations of the same algorithm
   - developing a hypothesis about the algorithm's efficiency class
   - ascertaining the efficiency of the program implementing the algorithm on a particular machine.
2. **how & What to measure**

- Include a variable counter, to count the number of times the algorithm's basic operation is executed.
- In the implementing the algorithm , measure the running time of basic operation

Example

- In unix, the system command time may be used.
- computing the difference between the two $(t_{finish} - t_{start})$.

Disadvantages of Measuring the system time

1. System's time is typically not very accurate, and you might get somewhat different results on repeated runs of the same program on the same inputs. An obvious remedy is to make several such measurements and then take their average (or the median) as the sample's observation point.
2. In the high speed of modern computers, the running time may fail to register at all and be reported as zero. The standard trick to overcome this obstacle is to run the program in an extra loop many times, measure the total running time, and then divide it by the number of the loop's repetitions.
3. The computer running under a time-sharing system such as UNIX, the reported time may include the time spent by the CPU on other programs, which obviously defeats the purpose of the experiment. Therefore, you should take care to ask the system for the time devoted specifically to execution of your program. (In UNIX, this time is called the "user time," and it is automatically provided by the time command.)

Advantage of Measuring physical running time

(i) the physical running time provides very specific information about an algorithm's performance in a particular computing environment

(ii) Measuring time spent on different segments of a program can pinpoint a bottleneck in the program's performance that can be missed by an abstract deliberation about the algorithm's basic operation profiling.

4. Deciding on a sample of inputs

   Sample size: (it is sensible to start with a relatively small sample and increase it later if necessary)

   Range of input sizes: (typically neither trivially small nor excessively large)

   - procedure for generating instances in the range chosen.
   - The instance sizes can either adhere to some pattern (e.g., 1000, 2000, 3000, . . . , 10,000 or 500, 1000, 2000, 4000, . . . , 128,000) or be generated randomly within the range chosen.
   - Several instances of the same size should be included or not.

5. Generate a sample of inputs (random numbers)

Typically, its output will be a value of a (pseudo)random variable uniformly distributed in the interval between 0 and 1. If a different (pseudo)random variable is desired, an appropriate transformation needs to be made. For example, if x is a continuous random variable uniformly distributed on the interval $0 \le x < 1$, the variable $y = l + \lfloor x(r - l) \rfloor$ will be uniformly distributed among the integer values between integers l and $r - 1 (l < r)$.

Alternatively, you can implement one of several known algorithms for generating (pseudo)random numbers. The most widely used and thoroughly studied of such algorithms is the linear congruential method

ALGORITHM

Random*(n, m,* seed*, a , b)*

//Generates a sequence of *n* pseudorandom numbers according to the linear

/ / c o n g r u e n t i a l m e t h o d

//Input: A positive integer *n* and positive integer parameters *m*, seed, *a , b*

//Output: A sequence *r*1, . . . , *rn* of *n* pseudorandom integers uniformly

/ / d i s t r i b u t e d a m o n g i n t e g e r v a l u e s b e t w e e n 0 a n d *m*−1

//Note: Pseudorandom numbers between 0 and 1 can be obtained

/ / b y t r e a t i n g t h e i n t e g e r s g e n e r a t e d a s d i g i t s a f t e r t h e d e c i m a l p o i n t

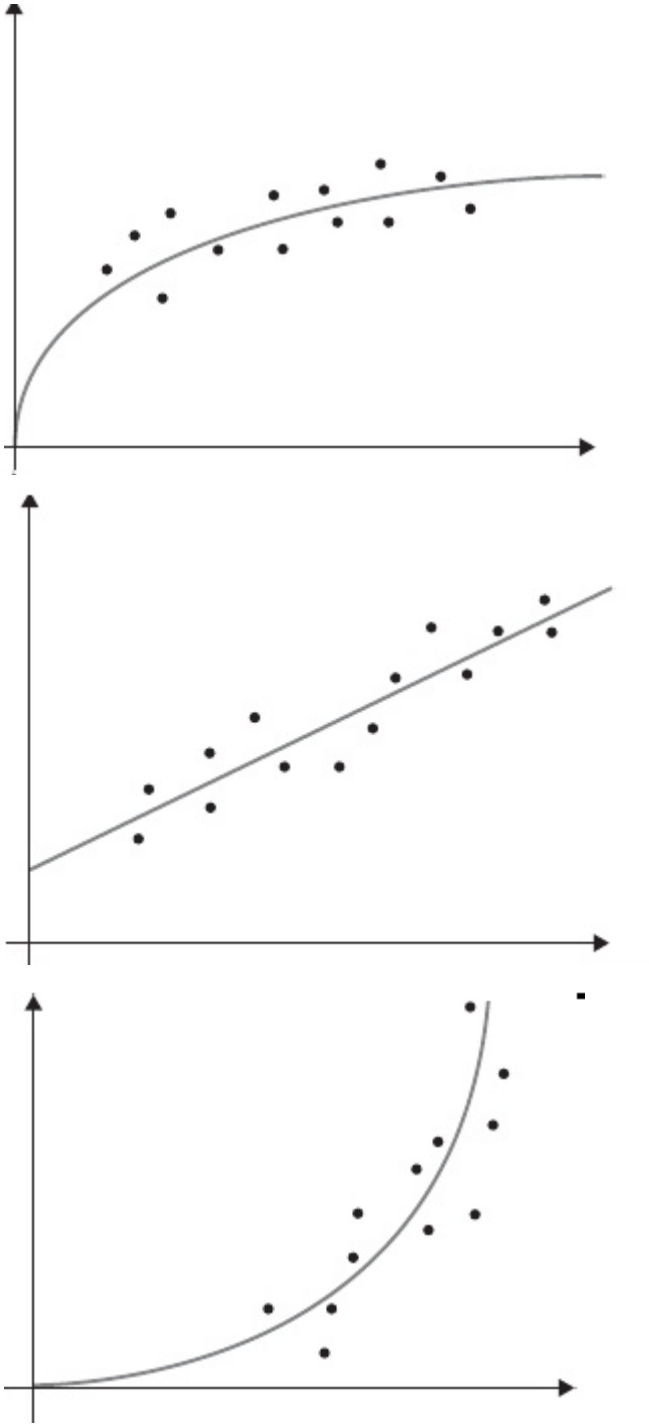
*r*0← seed

for *i* ←1 **to** *n* **do**

ri ←*(a* ∗*ri*−1+*b)* mod *m*

6. Data analysis
    - It is a good idea to use both these options whenever it is feasible because both methods have their unique strengths and weaknesses.
    - The advantages of tabulated data lies in the opportunity to manipulate it easily and to find efficiency class of the algorithm.
    - The Scatter plot representation helps in the analysis of algorithm efficiency class as given in figure

Shape of the scatter plot	**Efficiency class**
Concave shape	Logarithmic
Point around straight line or between two straight line	Linear
Convex shape	Quadratic and nlogn
Convex shape with rapid increase in the metrics valus	Cubic



Typical scatter plots. (a) Logarithmic. (b) Linear. (c) One of the convex functions

Application:

1. Predicting the algorithm performance on a sample size not included in the experiment sample.
2. The standard techniques of statistical data analysis and prediction can also be done.

20. Explain briefly about Algorithm Visualization.

Algorithm visualization is defined as the use of images to convey some useful information about algorithms. That information can be a visual illustration with the following combinations.

1. Algorithm's operation on different kinds of inputs
2. Same input for different algorithms to compare the execution speed.

An algorithm visualization uses graphic elements—points, line segments, two- or three-dimensional bars, and so on—to represent some "interesting events" in the algorithm's operation.

There are two principal variations of algorithm visualization:

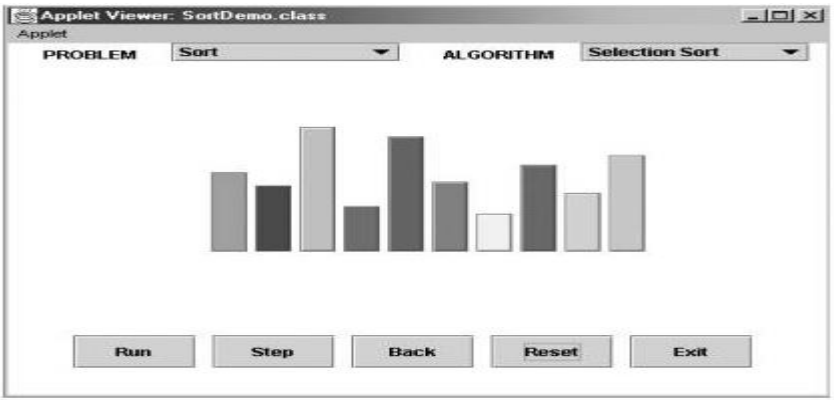
1. Static algorithm visualization
2. Dynamic algorithm visualization, also called algorithm animation

Static algorithm visualization shows an algorithm's progress through a series of still images. Algorithm animation, on the other hand, shows a continuous, movie-like presentation of an algorithm's operations. Animation is an arguably more sophisticated option, which, of course, is much more difficult to implement.

The features of an animations user interface was suggested by Peter Gloor is listed below

- Be consistent
- Be Interactive
- Be clear and concise
- Be forgiving to the user
- Adapt to the knowledge level of the user
- Emphasis the visual component
- Keep the user interested
- Incorporate both symbolic and iconic representations
- Include algorithm analysis and comparisons with other algorithm for the same problem
- Include execution history

The success of *Sorting Out Sorting* made sorting algorithms a perennial favorite for algorithm animation. Indeed, the sorting problem lends itself quite naturally to visual presentation via vertical or horizontal bars or sticks of different heights or lengths, which need to be rearranged according to their sizes (Figure 2.8). This presentation is convenient, however, only for illustrating actions of a typical sorting algorithm on small inputs. For larger files, *Sorting Out Sorting* used the ingenious idea of presenting data by a scatterplot of points on a coordinate plane, with the first coordinate representing an item's position in the file and the second one representing the item's value; with such a representation, the process of sorting looks like a transformation of a "random" scatterplot of points into the points along a frame's diagonal (Figure 2.9). In addition, most sorting algorithms work by comparing and exchanging two given items at a time—an event that can be animated relatively easily.



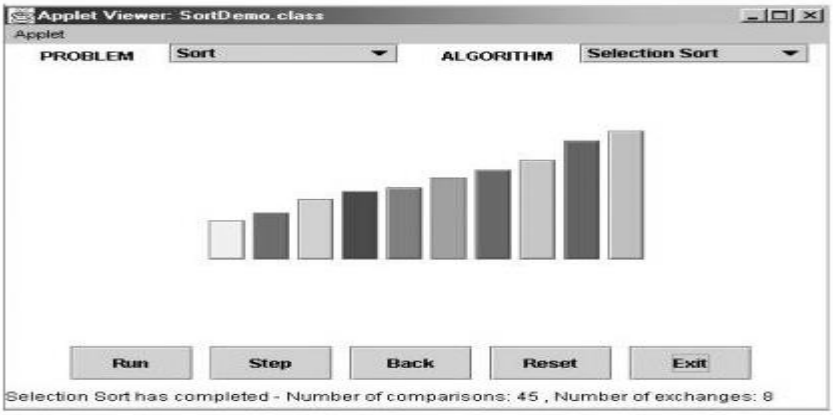


FIGURE 2.8 Initial and final screens of a typical visualization of a sorting algorithm using the bar representation.

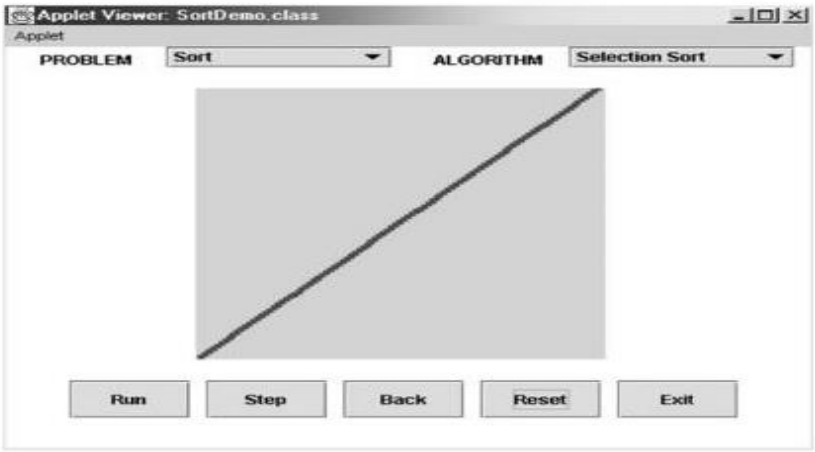
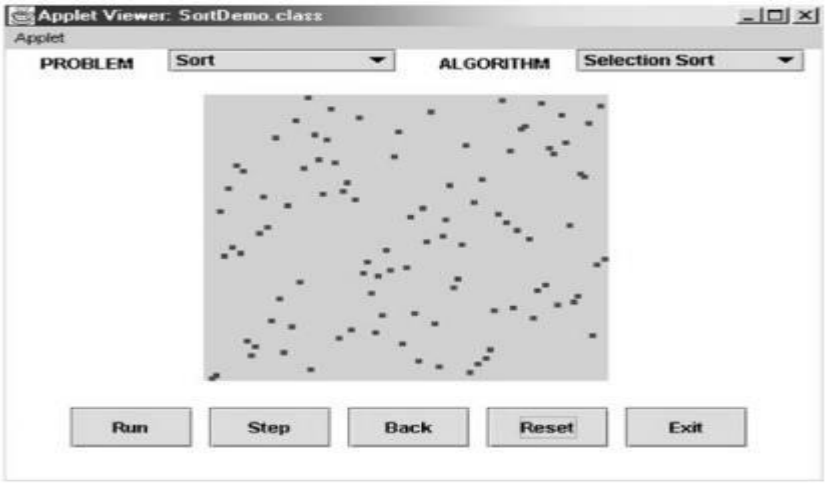


FIGURE 2.9 Initial and final screens of a typical visualization of a sorting algorithm using the scatterplot representation.



Applications:

1. **Education - Seeks to help students learning algorithms.**
2. **Research - Helps to uncover some unknown features of algorithms.**

IMPORTANT QUESTIONS

Part A

1. Show the notion of an algorithm. ***Dec 2009 / May 2013***
2. What are six steps processes in algorithmic problem solving? ***Dec 2009***
3. What is time and space complexity? ***Dec 2012***
4. Define Algorithm validation. ***Dec 2012***
5. Differentiate time complexity from space complexity. ***May 2010***
6. What is a recurrence equation? ***May 2010***
7. What do you mean by algorithm? ***May 2013***
8. Define Big Oh Notation. ***May 2013***
9. What is average case analysis? ***May 2014***
10. Define program proving and program verification. ***May 2014***
11. Define asymptotic notation. ***May 2014***
12. What do you mean by recursive algorithm? ***May 2014***
13. Establish the relation between O and Ω ***Dec 2010***
14. If $f(n) = a_m n^m + ... + a_1 n + a_0$. Prove that $f(n)=O(n^m)$.***Dec 2010***
15. Define the Fundamentals of Algorithmic Problem Solving
16. Short notes on Important Problem Types
17. .Define Fundamentals of the Analysis of Algorithm Efficiency
18. Show the Analysis Framework
19. Define Asymptotic Notations and its properties
20. Define Mathematical analysis for Recursive and Non-recursive algorithms**.**

Part B

1. Explain the notion of algorithm. ***May 2014***
2. Explain the fundamentals of algorithm. ***May 2014***
3. Find the time complexity and space complexity of the following problems. Factorial using recursion and compute the nth Fibonacci number using iterative statements. ***Dec 2012***
4. Solve the following recurrence relations: ***Dec 2012***

1. $T(n)= \begin{cases} 2T(n/2)+3 & n>2 \\ 2 & n=2 \end{cases}$

2. $T(n)= \begin{cases} 2T(n/2)+cn & n>1 \\ a & n=1 \end{cases}$ where a and c constants

5. Distinguish between Big Oh, Theta and Omega notation. ***Dec 2012***
6. Analyse the best case, average and worst case analysis for linear search. ***Dec 2012***
7. Explain how time complexity is calculated. Give an example. ***Apr 2010***
8. Elaborate on asymptotic notation with example. ***Apr 2010***
9. Briefly explain the time complexity, space complexity estimation ***June 2013***
10. Write linear search algorithm and analyse its complexity. ***June 2013***
11. Show the following equalities are correct ***June 2013***
    i. $5n^2-6n = \Phi(n^2)$
    ii. $n!=O(n^n)$
    iii. $n^3+10^6n^2=\Theta(n^3)$
    iv. $2n^2 2^n + n \log n = \Theta(n^2 2^n)$
12. What are the features of an efficient algorithm? ***June 2014***
13. What is space complexity? With an example explain the components of fixed and variable part in space complexity. ***June 2014***
14. Explain towers of Hanoi problem and solve it using recursion. ***June 2014***
15. Derive the recurrence relation for Fibonacci series algorithm : also carry out time complexity analysis. ***June 2014***
16. Discuss in details about the efficiency of the algorithm with example. ***Mar 2014***

17.Explain the procedure to calculate the time complexity of binary search using non-recursive Algorithm.

18. Explain briefly the time complexity and space complexity estimation. ***Nov 2010***

19. Write a linear search algorithm and analyse its best, worst and average case time complexity.

20. Prove that for any two functions f(n) and g(n), we have f(n)-> Θ(g(n)) if and only if f(n) - > O(g(n)) and f(n) ->Ω(g(n)) ***Nov 2010***

21.Explain the Mathematical analysis for non-recursive algorithm

ANNA UNIVERSITY APRIL/MAY 2015

PART-A

1.write algorithm to find the number of binary digits in the binary representation of a positive decimal integer **Part A – Refer Q. No. 56**

2.write down the properties of asymptotic notations. **Part A – Refer Q. No. 57**

PART-B

11.(a)if you have to solve the searching problem for a list of n numbers, how can you take advantage of the fact that the list is known to be sorted? Give separate answers for

(i) List represented as arrays

(ii) List represented as linked list Compare the time complexity involved in the analysis of both the algorithms **Refer Q. No. 27**

OR

(b)(i)Derive the worst case analysis of merge sort using suitable illustration**Refer Q.No. 28**

(ii) Derive a loose bound on the following equation:

$F(x)=35\ x^8 -22x^7+14x^5 -2x^4 -4x^2+x-15$ **Q.No. 15**

ANNA UNIVERSITY NOV/DEC 2015

PART-A

1.The *(log n)*th smallest number of n unsorted numbers can be determined in O(n) average-case time (True/False) **Refer Q. No. 60**

2. Fibonacci algorithm and its recurrence relation **Refer Q. No. 61**

PART-B

11.(a)(i)write Insertion sort algorithm and estimate its running time.(8) **Refer Q. No. 12**

(ii)find the closest asymptotic tight bound by solving the recurrence equation *T(n)=8T(n/2)+n2* with *(T(1)=1)* using recursion tree method.[Assume that T(1)ЄΘ(1)]

Refer Q. No. 14

OR

(b)(i)Suppose W satisfies the following recurrence equation and base case (where c is a constant):*W(n)=c.n+W(n/2)* and *W(1)=1*.What is the asymptotic order of W(n).

Refer Q. No. 14

(ii)Show how to implement a stack using two queues. Analyze the running time of the stack Operations. **Refer Q. No. 13**

ANNA UNIVERSITY APRIL/MAY 2016

PART-A

1. Give the Euclid's algorithm for computing gcd(m, n) **Refer Q. No. 58**

2. Compare the order of growth n(n-1)/2 and n2. **Refer Q. No. 59**

PART-B

1. a.(i) Give the definition and Graphical Representation of O-Notation.(8) **Refer Q. No. 4**

(ii) Give an algorithm to check whether all the Elements in a given array of n elements are distinct. Find the worst case complexity of the same. (8) **Refer Q. No.5(2)**

OR

(b) Give the recursive algorithm which finds the number of binary digits in the binaryrepresentation of a positive decimal integer. Find the recurrence relation and complexity. (16) **Refer Q. No.6(3)**

ANNA UNIVERSITY NOV/DEC 2016

PART-A

1.Design an algorithm to compute the area and circumference of a circle **Refer Q. No. 63**

2.Define recurrence relation. **Refer Q. No. 45**

PART-B

11.(a)(i)Use the most appropriate notation to indicate the time efficiency class of sequential search algorithm in the worst case,best case and the average case. **Refer Q. No. 17**

(ii) State the general plan for analyzing the time efficiency of nonrecursive algorithm and explain with an example(8) **Refer Q. No. 5**

(b) Solve the recurrence relations **Refer Q. No. 16**

$X(n) = x(n-1) + 5$ for $n > 1$ $x(1)=0$

$X(n) = 3x(n-1)$ for $n > 1$ $x(1)=4$

$X(n) = x(n-1) + n$ for $n > 0$ $x(0)=0$

$X(n) = x(n/2) + n$ for $n > 1$ $x(1)=1$ (solve for $n= 2^k$)

$X(n) = x(n/3) + 1$ for $n > 1$ $x(1)=1$ (solve for $n= 3^k$) (16)

ANNA UNIVERSITY APRIL/MAY 2017

PART-A

1. What is an algorithm? **Refer Q. No. 1**
2. Write an algorithm to compute the greatest common divisor of two numbers **Refer Q. No. 10**

PART-B

1. Explain briefly Big oh notation , Omega notation and Theta notation give an example **Q. No. 30**

2.Briefly explain the mathematical analysis of recursive and non recursive algorithm**Q.No.35 & 40**

ANNA UNIVERSITY NOV/DEC 2017

PART-A

1. How to measure an algorithm's running time ? **Refer Q. No. 21**
2. What do you mean by "worst case efficiency: of an algorithm. **Refer Q. No. 55**

PART-B

1. Discuss the steps in Mathematical analysis for recursive algorithms. Do the same for finding Factorial of a number **Refer Q. No. 6**

2. What are the Rules of Manipulate Big-Oh Expression and about the typical growth rates of algorithms? **Refer Q.No.4**

ANNA UNIVERSITY MAY/JUNE 2018

PART-A

1. Give the Euclid's algorithm for computing gcd of two numbers. **Refer Q. No. 58**
2. What is a basic operation? **Refer Q. No. 63**

PART-B

1. a) Define Big O notation, Big Omega and Big Theta Notation. Depict the same graphically and explain. **Refer Q.No.4**

b) Give the general plan for Analyzing the time efficiency of Recursive Algorithms and use recurrence to find number of moves for Towers of Hanoi problem. **Refer Q.No.6**

ANNA UNIVERSITY NOV/DEC 2018

PART-A

1. Define algorithm. List the desirable properties of an algorithm. **Refer Q. No. 64**
2. Define best, worst, average case time complexity. **Refer Q. No. 65**

PART-B

1. (i) Prove that if g(n) is Ω(f(n)) then f(n) is O(g(n)). **Refer Q.No.18**
   (ii) Discuss various methods used for mathematical analysis of recursive algorithms. **Refer Q.No.6**
2. Write the asymptotic notations used for best case, average case and worst case analysis of algorithms. Write an algorithm for finding maximum element in an array. Give best, worst and average case complexities. **Refer Q.No.4**

ANNA UNIVERSITY APRIL/MAY 2019

PART-A

1. How do you measure the efficiency of an algorithm? - **Refer Q.No.29**
2. Prove that the of f(n)=o(g(n)) and g(n)=o(f(n)),then f(n)=θ g(n). **- Refer Q.No.66**

PART-B

1.a) (i) solve the following recurrence equation: **- Refer Q.No.8**
   1.T(n)=T(n/2)+1,where $n=2^k$ for all k>=0
   2.T(n)= T(n/3)+ T(2n/3)+cn,where 'c' is a constant and 'n' is the input size.
(ii) Explain the steps involved in problem solving. **- Refer Q.No.8**

2.(i) write an algorithm for determining the uniqueness of an array. Determine the time complexity of your algorithm. **- Refer Q.No.5**
(ii) Explain time-space trade off of the algorithm designed **- Refer Q.No.3**

ANNA UNIVERSITY NOV/DEC 2019

PART-A

1. State the transpose symmetry property of O and Ω **- Refer Q.No.66**
2. Define recursion **- Refer Q.No.67**

PART-B

1. a) i) Solve the following recurrence equations using iterative method or tree **Refer Q.No.6**
   ii) Elaborate asymptotic analysis of an algorithm with an example. **Refer Q.No.4**
2. b) write an algorithm using recursion that determines the GCD of two numbers. Determine the time and space complexity **- Refer Q.No.1.A**

ANNA UNIVERSITY NOV/DEC 2021

PART-A

1. Define algorithm with its properties. **Refer Q.No.1**
2. List the reasons for choosing an approximate algorithm. **Refer Q.No.68**

PART-B

1. a) i) Consider the problem of counting, in a given text the number of substrings that start with an Aand end with a B. For example, there are four such substrings in CABAAXBYA. Design a brute-force algorithm for this problem and determine its efficiency class. **Refer Q.No.6**

ii) "The best-case analysis is not as important as the worst-case analysis of an algorithm". Yes or No ? Justify your answer with the help of an example**. Refer Q.No.11**

2. b) (i) Solve : T(n) = 2T(n/2) + n3**. Refer Q.No.11**

(iii) Explain the importance of asymptotic analysis for running time of an algorithm with an example. **Refer Q.No.4**

ANNA UNIVERSITY NOV/DEC 2021

PART-A

1. Define the notation big-Omega. **Refer Q.No.14**
2. What is meant time complexity of an algorithm? **Refer Q.No.7**

PART-A

11. a) Outline worst case running time, best case running time and average case running time of an algorithm with an example?

b) Outline a recursive algorithm and non recursive algorithm with an example.

Refere Q.No.35 & 40