Chapter 1

Introduction

Contents

Characteristics of algorithm. Analysis of algorithm : Asymptotic analysis of complexity bounds – best, average and worst-case behaviour, Performance measurements of Algorithm, Time and space trade-offs, Analysis of recursive, algorithms through recurrence relations : Substitution method, Recursion tree method and Masters' theorem.

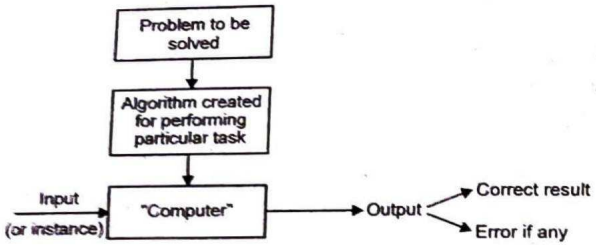
POINTS TO REMEMBER

1. 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.
2. Input, output, finiteness, definiteness and effectiveness are the properties of an algorithm.
3. Time complexity and space complexity are two types of complexities.
4. The complexity of an algorithm is a function f(n) which measure the time and space used by an algorithm interm of input size n.
5. Time complexity is the amount of time required to execute an algorithm.
6. Space complexity is the amount of memory required to execute an algorithm.
7. There are three types of running time analysis
   (a) Worst-case
   (b) Average-case
   (c) Best-case
8. Measuring the performance of an algorithm in relation with the input size n is called order of growth.
9. Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.
10. The Big Oh notation is denoted by 'O'.
11. The Big O notation define an upper bound of an algorithm running time, it bounds a function only from above.
12. Omega notation is denoted by 'Ω'.
13. The omega notation is used to represent the lower bound of algorithm's running time.
14. The theta notation is denoted by θ.
15. The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.
16. In Polynomial algorithm run time is bounded by a polynomial function.
17. In exponential algorithms run time is bounded by an exponential function, where exponent is Λ.

QUESTION-ANSWERS

Q 1. What is an algorithm?
(PTU, May 2019 ; Dec. 2019, 2016, 2015, 2013, 2008, 2005, 2004)

Ans. The algorithm is defined as a collection of unambiguous instructions occuring in some specific sequence and such an algorithm should produce output for given set of input in finite amount of time or we can also say that 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.



Notion of algorithm

Q 2. What are the properties of algorithm?

Ans. Properties of algorithm :

1. **Input :** It generally requires finite number of inputs.
2. **Output :** It must produce at least one output.
3. **Uniqueness :** Each instruction should be clear and unambiguous.
4. **Finiteness :** It must terminate after a finite number of steps.
5. **Effectiveness :** The steps of an algorithm must be basic. Basic means, the person should be able to carry out these steps using pencil and paper without applying any intelligence.

Q 3. What do you mean by complexity of an algorithm? Explain time and space complexity. **(PTU, May 2016, 2013, 2010, 2008 ; Dec. 2010, 2009)**

Ans. Complexity of an algorithm : The complexity of an algorithm is a function f(n) which measure the time and space used by an algorithm interms of input size n.

Time complexity : Time complexity is the amount of time required to execute an algorithm or we can also say that the time complexity of an algorithm is the amount of computer time required by an algorithm to run to completion.

The time complexity of an algorithm is commonly expressed using big O notation, which suppresses multiplicative constants and lower order terms. When expressed this way, the ime complexity is said to be described asymptotically, i.e., as the input size goes to infinity. For example, if the time required by an algorithm on all inputs of size n is at most $5n^3 + 3n$, the symptotic time complexity is $O(n^3)$.

It is difficult to compute the time complexity in terms of physically clocked time. Let us

take the example of multiuser system. In multiuser system, executing time depends on many factors such as :

1. System load
2. Number of other programs running
3. Instruction set used
4. Speed of underlying hardware

The time complexity is therefore given in terms of frequency count. Frequency count is a count denoting number of times of execution of statement.

Space complexity : Space complexity is the amount of memory required to execute an algorithm or we can also say that the space complexity can be defined as the 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 :

$$S(p) = C + Sp$$

Where C is a constant i.e. fixed part and it denotes the space of inputs and outputs. This space is an amount of space taken by instruction, variables and identifiers. Sp is a space dependent upon instance characteristics. This is a variable part whose space requirement depends on particular problem instance.

Space complexity is normally expressed as an order of magnitude, e.g. O(N ^ 2) means that if the size of the problem (N) doubles then four times as much working storage will be needed.

Q 4. What do you mean by "Worst case-efficiency" of an algorithm?
(PTU, Dec. 2009)

Ans. 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 algorithms runs the longest among all possible inputs of that size.

For example : If you want to sort a list of numbers in ascending order when the numbers are given in descending order. In this running time will be the longest.

Q 5. What do you mean by "Best case-efficiency" of an algorithm?

Ans. 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.

For example : If you want to sort a list of numbers in ascending order when the numbers are given in ascending order. In this running time will be the smallest.

Q 6. Define the "Average case-efficiency" of an algorithm.

Ans. The "Average case efficiency" of an algorithm is its efficiency for the input of size n, for which the algorithm runs between the best case and the worst case among all possible inputs of that size.

Q 7. How is an algorithm's time efficiency measured?

Ans. Time efficiency indicates how fast the algorithm runs. An algorithm's time efficiency

is measured as a function of its input size by counting the number of times its basic operation (running time) is executed. Basic operation is the most time consuming operation in the algorithm's innermost loop.

Q 8. Write a short note on order of Growth.

Ans. Order of Growth : Measuring the performance of an algorithm in relation with the input size n is called order of growth. For example, the order of growth for varying input size of n is as given below :

n	log n	n log n	n^2	n^3	2^n	n!
1	0	0	1	1	2	1
2	1	2	4	8	4	2
4	2	8	16	64	16	24
8	3	24	64	512	256	40320
16	4	64	256	4096	65,536	Large
32	5	160	1024	32768	4,294,967,296	Very large

Order of growth for some functions

The efficiency analysis framework concentrates on the order of growth of an algorithm's basic operation count as the principal indicator of the algorithm's efficiency. To compare and rank such orders of growth we use three notations.

(i) O (Big oh) notation

(ii) Ω (Big omega) notation

(iii) θ (Big theta) notation

Q 9. Define best-case step count.

Ans. The best case step count is the minimum number of steps that can be executed for the given parameters.

Q 10. Define worst case step count.

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

Q 11. Define average step count.

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

Q 12. What do you understand by Asymptotic notation? Also define the following notations.

(i) Big O (ii) Omega (iii) Theta

(PTU, Dec. 2016, 2014, 2013 ; May 2019, 2018, 2017, 2016, 2014, 2013)

Ans. Asymptotic notation : Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis or we can also say that asymptotic notation is a shorthand way to represent the time complexity. Using asymptotic notations we can give time complexity as "fastest possible", "slowest possible" or "average time". There are various notations such as Ω, θ and O used are called asymptotic notations. These notations are mostly used to represent time complexity of algorithms.

(i) Big O notation : The big oh notation is denoted by 'O'. The big O notation define an upper bound of an algorithm running time, it bounds a function only from above.

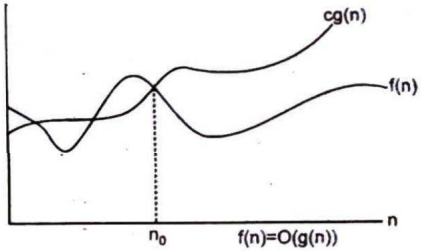
e.g. : Let us consider the case of insertion sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of insertion sort is O(n ^ 2). Note that O(N ^ 2) also cover linear time. If we use θ notation to represent time complexity of insertion sort, we have to use two statements for best and worst cases :

(a) The worst case time complexity of insertion sort is θ(n ^ 2).

(b) The best case time complexity of insertion sort is θ(n).

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

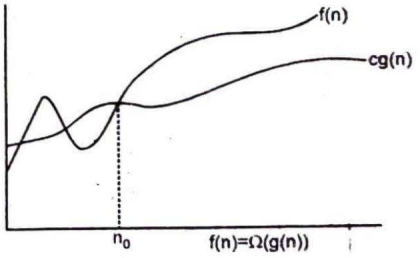
O(g(n)) = {f(n) : there exist positive constants c and n_0 such that 0<=f(n) < = cg(n) for all n > = n_0}



(ii) Omega : Omega notation is denoted by 'Ω'. This notation is used to represent the lower bound of algorithm's running time. Using omega notation we can denote shortest amount of time taken by algorithm.

Ω notation can be useful when we have lower bound on time complexity of an algorithm. For a given function g(n), we denote by Ω (g(n)) the set of functions.

Ω(g (n)) = {f(n) : there exist positive constants c and n_0 such that 0<=cg(n) < = f(n) for all n > = n_0}.



(iii) Theta : The theta notation is denoted by θ. By this method the running time is between upper bound and lower bound. We can also say that the theta notation bounds a functions from above and below, so it defines exact asymptotic behavior. A simple way to get theta notation of an expression is to drop low order term and ignore leading constants. For example let us consider the following expression

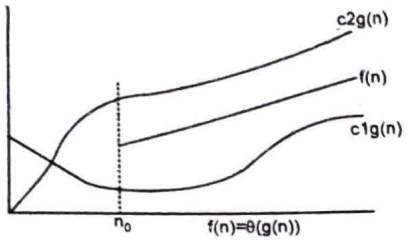
$3n^3 + 6n^2 + 6000 = \theta(n^3)$

Dropping lower order terms is always fine because there will always be a n_0 after which $\theta(n^3)$ beats $\theta(n^2)$ irrespective of the constants involved.

For a given function g(n), we denote $\theta(g(n))$ is following set of functions.

$\theta(g(n))$ = {f(n) : there exist positive constants c1, c2 and n_0 such that 0 <= c1*g(n)<= f(n)<= c2 * g(n) for all n>= n_0}

if f(n) is theta of g(n), then the value f(n) is always between c1 * g(n) and c2 * g(n) for large values of n(n>=n_0). The defination of theta also requires that f(n) must be non-negative for values of n greater than n_0.



Q 13. Write a short note on polynominal Vs exponential running time.

(PTU, Dec. 2019, 2016 ; May 2015)

Ans. Polynomial Vs. Exponential running time :

Polynomial running time : An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a given input is $O(n^k)$ for some non-negative integer k, where n is the complexity of the input. Polynomial-time algorithms are said to be "fast". Most familiar mathematical operations such as addition, subtraction, multiplication and division, as well as computing square root, powers and logarithms, can be performed in polynomial time. Computing the digits of most interesting mathematical constants, including pi and e, can also be done in polynomial time.

All basic arithmetic operations (i.e. addition, subtraction, multiplication, division). comparison operations, sort operations are considered as polynomial time algorithms.

We can also say that in polynomial algorithm run time is bounded by a polynomial function (addition, subtraction, multiplication, division, non-negative integer exponents).

- n, n^2, n^{5000}, etc.

Exponential running time : The set of problems which can be solved by an exponential time algorithms, but for which no polynomial time algorithm is known. An algorithm is said to be exponential time, if T(n) is upper bounded by $2^{poly(n)}$, where poly(n) is some polynomial in n. More formally, an algorithm is exponential time if T(n) is bounded by $O(2^{nK})$ for some constant k.

We can also say that in this run time is bounded by an exponential function, where exponent is n.

- n^n, 2^n, etc.

Q 14. What is program? **(PTU, Dec. 2004)**

Ans. A program is a set of operations that a computer follows in order for it to run properly. The sequence of steps involved in the program help the computer run smoothly and to avoid errors.

Q 15. What is set algorithm? **(PTU, May 2006)**

Ans. Set algorithms are input-specialized algorithms that deal with sets. They implement basic mathematical set operations over sets with generic element types. STL implements set containers with red-black trees. The reason for this is that operations with sets require fast and bounded search on set members. This can be achieved with binary search on red-black trees. Red-black trees are one of the ways of getting balanced binary trees and O (log N) bounded binary search time, the other alternatives being AVL trees and B-trees. AVL trees are better balanced than red-black trees ; however, they require more operations to maintain the balance. B-trees would be a better choice with huge sets. Having set elements in binary search trees assures the precondition that all set elements should be sorted. Set algorithms can be applied on container classes other than sets but in this case programmer should take care of the sorting.

Q 16. What do you understand by algorithm evaluation?

(PTU, May 2018 ; Dec. 2007)

Ans. Evaluation is same as the testing of an algorithm. It mainly refers to the finding of errors by processing an algorithm.

Q 17. Given an example of an algorithm which is infinite in nature.

(PTU, May 2009, 2007 ; Dec. 2009, 2008)

Ans. Divide and conquer is such an algorithm which is infinite in nature. In this approach, whole problem is divide into several sub problems. These sub problems are solved recursively and are smaller in size as compared to original problem.

Q 18. Give two metrics for evaluating an algorithm. **(PTU, May 2010)**

Ans. Average Bias (Accuracy) : Average bias is one of the conventional metrics that has been used in many ways as a measure of accuracy. It averages the errors in predictions made at all subsequent times after prediction starts for the *l*th UUT. This metric can be extended to average biases over all UUTs to establish overall bias.

$$B_l = \frac{1}{l}\sum_{i=1}^{l} \Delta^l(i).$$

Sample Standard Deviation (Precision) : Sampled standard deviation measures the dispersion/spread of the error with respect to the sample mean of the error. This metrics is restricted to the assumption of normal distribution of the error. It is, therefore, recommended to carry out a visual inspection of the error plots to determine the distribution characteristics before interpreting this metric

$$S = \sqrt{\frac{\sum_{i=1}^{l}(\Delta(i) - m)^2}{l-1}}.$$

Q 19. What are combinational algorithms? **(PTU, May 2011)**

Ans. Combinational algorithms are algorithms for investigating combinatorial structures.

- **Generation :** Construct all combinatorial structures of a particular type.
- **Enumeration :** Compute the number of all different structure of a particular type.
- **Search :** Find at least one example of a combinatorial structure of a particular type.
- **Optimization problems :** Can be seen as a type of search problem.

Q 20. List the various steps used in designing an algorithm. **(PTU, Dec. 2011 ; May 2013, 2008)**

Ans. Various steps used in designing an algorithm :

1. Understanding the problem
2. Decision making on
   (a) Capabilities of computational devices
   (b) Choice for either exact or approximate problem solving method
   (c) Data structures
   (d) Algorithmic strategies.
3. Specification of algorithm
4. Algorithmic verifications
5. Analysis of algorithm
6. Implementation or coding of algorithm.

Q 21. What is algorithm? Write the various performance analysis techniques of algorithm. Discuss advantages and disadvantages of each. (PTU, Dec. 2018 ; May 2008)

Ans. An algorithm is a set of rules for carrying out calculation either by hand or on a machine.

- An algorithm is a finite step-by-step procedure to achieve a required result.
- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is a sequence of operations performed on data that have to be organized in data structures.
- An algorithm is an abstraction of a program to be executed on a physical machine (model of computation).

There are basically two techniques to analyse the algorithm which are space complexity and time complexity. Time complexity of an algorithm concerns determining an expression of the number of steps needed as a function of the problem size. Since, the step count measure is somewhat coarse, one does not aim at obtaining an exact step count. Instead, one attempts only to get asymptotic bounds on the step count. Asymptotic analysis makes use of the O (Big Oh) notation. Two other notational constructs used by computer scientistis in the analysis of algorithms are Θ (Big Theta) notation and Ω (Big Omega) notation. The performance evaluation of an algorithm is obtained by totalling the number of occurrences of each operation when running the algorithm. The performance of an algorithm is evaluated as a function of the input size n and is to be considered modulo a multiplicative.

Q 22. Explain the algorithm of a non-deterministic finite automation. **(PTU, Dec. 2009, 2007)**

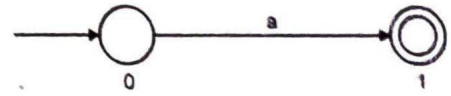
Ans. Let Q be a finite set and let Σ be a finite set of symbols. Also let δ be a function from $Q \times \Sigma$ to 2^Q, let q_0 be a state in Q and let A be a subset of Q. We call the elements of Q a state, δ the transition function, q_0 the initial state and A the set of accepting states.

Then a non-deterministic finite automation is a 5-tuple $< Q, \Sigma, q_0, \delta, A >$

Example : $Q = \{0, 1\}$, $\Sigma = \{a\}$, $A = \{1\}$ the initial state is 0 and δ is as shown in the following table :

State (q)	Input (a)	Next Stae (δ (q, a))
0	a	{1}
1	a	θ

A state transition diagram for this finite automation is given below :



If the alphabet Σ is changed to {a, b} in stead of {a}, this is still an NFA that accepts {a}.

Q 23. What is deterministic algorithm? **(PTU, May 2012, 2007)**

OR

What do you mean by deterministic and non-deterministic algorithms? Differentiate between them. Write example for each of them. **(PTU, Dec. 2018)**

Ans. A **deterministic algorithm** is an algorithm which, in informal terms, behaves predicably. Given a particular input, it will always produce the same output, and the underlying machine will always pass through the same sequence of states. Deterministic algorithms are by far the most studied and familiar kind of algorithm, as well as one of the most practical, since they can be run on real machines efficiently.

Formally, a deterministic algorithm computes a mathematical function ; a function has a unique value for any given input, and the algorithm is a process that produces this particular value as output.

Non-deterministic : A variety of factors can cause an algorithm to behave in a way which is not deterministic, or no-deterministic.

- If it uses external state other than the input, such as user input, a global variable, a hardware timer value, a random value, or stored disk data.
- If it operates in a way that is timing-sensitive, for example if it has multiple processors writing to the same data at the same time. In this case, the precise order in which each processor writes its data will affect the result.
- If a hardware error causes its state to change in an unexpected way.

Although real programs are rarely purely deterministic, it is easier for humans as well as other programs to reason about programs that are. For this reason, most programming

languages and especially functional programming languages make an effort to prevent to above events from happening except under controlled conditions.

Difference betwen deterministic and non-deterministic algorithm : Algorithm is deterministic if for a given input the output generated is same for a function. A mathematical function is deterministic. Hence, the state is known at every step of the algorithm.

Algorithm is non-deterministic if there are more than one path the algorithm can take. Due to this, one cannot detemine the next state of the machine running the algorithm. Example would be a random function.

Q 24. What is reverse polish notation? **(PTU, Dec. 2005)**

Ans. Reverse Polish Notation is a way of expressing arithmetic expressions that avoids the use of brackets to define priorities for evaluation of operators. In ordinary notation, one might write (3 + 5) * (7 – 2) and the brackets tell us that we have to add 3 to 5, then subtract 2 from 7, and multiply the two results together. In RPN, the numbers and operators are listed one after another, and an operator always acts on the most recent numbers in the list. The numbers can be thought of as forming a stack, like a pile of plates. The most recent number goes on the top of the stack. An operator takes the appropriate number of arguments from the top of the stack and replaces them by the result of the operation.

In this notation the above expression would be 3 5 + 72 – *

Q 25. What is a recursive relationship? **(PTU, Dec. 2006)**

Ans. A recursive relationship can be defined as a relationship that is expressed about multiple records within one table. As an example if we take an employee table then there are some employees who are supervisor and some who are being supervised. This is the relationship of supervisor and supervisee is called a recursive relationship.

More concrete definition of recursive relationship can be a relationship between information held in a field, group of fields, or complete record and information of the same type held in one or more other occurrences of that record, or part thereof.

Q 26. What is asymptotic time complexity? **(PTU, May 2011 ; Dec. 2007)**

Ans. In computer science, the time-complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the size of the input to the problem. The time complexity of an algorithm is commonly expressed using big O notation, which supresses multiplicative constant and lower order terms. When expressed this way, the time complexity is said to be described asymptotially, i.e., as the input size goes to infinity. For example, if the time required by an algorithm on all inputs of size n is at most $5n^3 + 3n$, the asymptotic time complexity is O (n^3).

Q 27. Define big omega notation (Ω) and little omega notation (ω).
(PTU, Dec. 2008, 2005)

Ans. Big Omega Notation (Ω) : A function t (n) is said to be in Ω (g(n)), denoted t (n) $\in$ Ω (g(n)), if t (n) is bounded below by some positive constant multiple of g (n) for all large n, i.e., if there exist some positive constant c and some non negative integer n_0 such that

$$T(n) \leq c\, g(n) \text{ for } n \geq n_0$$

Little Omega Notation (ω) : The function in ω (g) are the larger function of Ω (g). Conspiring 'g' be set of function from the non-negative integer into the positive real numbers. Then ω (g) is set of function 'f' also from the non-negative integers into the positive real numbers, such that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

Q 28. Differentiate between space complexity and time space trade off.
(PTU, May 2009)

Ans. A space-time or time-memory tradeoff is a situation where the memory use can be reduced at the cost of slower program execution (and, conversely, the computational time can be reduced at the cost of increased memory use). As the relative costs of CPU cycles, RAM space, and hard drive space change, hard drive space has for some time been getting cheaper at a much faster rater than other components of computers, the appropriate choices for space-time tradeoffs have changed radically. Often, by exploiting a space-time tradeoff, a program can be made to run much faster.

Q 29. Is 2n + 2 = O (2n + 1)? **(PTU, May 2010)**

Ans. No, 2n + 2 is not equal to O (2n + 1).

Q 30. Define recurrence relation. **(PTU, Dec. 2010, 2008 ; May 2013, 2010)**

Ans. A **recurrence relation** is an equation that recursively defines a sequence : each term of the sequence is defined as a function of the preceeding terms. The term **difference equation** sometimes (and for the purposes of this article) refers to a specific type of recurrence relation. However, 'difference equation' is frequently used to refer to any recurrence relation. An example of a recurrence relation is the logistic map :

$$x_{n+1} = \gamma x_n (1 - x_n)$$

Some simply defined recurrence relations can have very complex (chaotic) behaviours, and they are a part of the field of mathematics known as non-linear analysis. Solving a recurrence relation means obtaining a closed-form solution : a non-recursive function of n.

A recurrence relation for the sequence $\{a_n\}$ is an equation that expresses a_n is terms of one or more of the previous terms of the sequence, namely, a_0, a_1, ..., a_{n-1}, for all integers n with $n \geq n_0$, where n_0 is a non-negative integer.

A sequence is called a solution of a recurrence relation if it terms satisfy the recurrence relation. In other words, a recurrence relation is like a recursively defined sequence, but without specifying any initial values (initial conditions).

Therefore, the same recurrence relation can have (and usually has) multiple solutions. If both the initial conditions and the recurrence relation are specified, then the sequence is uniquely determined.

Example : Consider the recurrence relation

$$a_n = 2a_{n-1} - a_{n-2} \text{ for } n = 2, 3, 4, \ldots.$$

Is the sequence $\{a_n\}$ with $a_n = 3_n$ a solution of this recurrence relation.

For $n \geq 2$ we see that

$$2a_{n-1} - a_{n-2} = 2(3(n-1)) - 3(n-2) = 3n = a_n.$$

Therefore, $\{a_n\}$ with $a_n = 3n$ is a solution of the recurrence relation.

Q 31. What do you mean by term order of complexity?
(PTU, May 2012 ; Dec. 2010, 2009)

Ans. Complexity can then be characterized by lack of symmetry or 'symmetry breaking' by the fact that no part or aspect of a complex entity can provide sufficient information to actually or statistically predict the properties of the others parts. This again connects to the difficulty of modelling associated with complex systems.

Q 32. Define Big oh Notation (O) and Little oh Notation (o). **(PTU, May 2009)**

Ans. Big oh Notation (O) : f (n) is o (g(n)) if f (n) is asymptotically less than or equal to g (n).

Little oh Notation (o) : f (n) is o (g(n)) if f (n) is asymptotically strictly less than to g (n).

Q 33. What do you mean by worst case analysis? **(PTU, May 2013 ; Dec. 2010)**

Ans. The worst case time complexity is the function define by the maximum amount of time needed by an algorithm for an input of size, 'n'. Thus, it is the function defined by the maximum number of steps taken on any instance of size 'n'.

Q 34. List different notions of complexity of an algorithm. **(PTU, Dec. 2011)**

Ans. Big O notation, Omega notation, Theta notation.

Q 35. Find the big of (0) notation for the following function
(i) f (n) = 5995 (ii) f (n) = 3n + 5 (iii) f (n) = $69n^2 + 35$ (iv) f (n) = $70n^3 + 35\ n^2 + 45n$
(PTU, Dec. 2005)

Ans. (i) f (n) = 5995

f (n) ≤ 5995 * 1, where c = 5995, and $n_0 = 0$

thus, big of (0) notation is f (n) = 0 (1).

(ii) f (n) = 3n + 5

for f (n) = 3n + 5, where 'n' is at least 5, n ≥ 5

$3n + 5 \le 3n + n \le 4n$

So, f (n) = 0 (n)

(iii) $f(n) = 69n^2 + 35$ for n ≥ 35

$69n^2 + 35 \le 69n^2 + n$

Now, for $n \le n^2$

$69n^2 + n \le 69n^2 + n^2 \le 70n^2$ {c = 70, $n_0 = 1$}

So, f (n) = 0 (n^2).

(iv) $f(n) = 70n^3 + 35n^2 + 45n$

for $n^2 \ge 45n$

$70n^3 + 35n^2 + 45n \le 70n^3 + 35n^2 + n^2 \le 70n^3 + 36n^2$

Now for $n^3 \ge 36n^2$

$70n^3 + 36n^2 \le 70n^3 + n^3 \le 71n^3$, {C = 71, $n_0 = 70$)

So, f (n) = O (n^3).

Q 36. Argue on the following relation
(i) Is $2^{n+1} = 0\ (2^n)$?
(ii) Is $2^{2n} = 0\ (2^n)$

Ans. (i) $2^{n+1} = 0\ (2^n)$? **(PTU, May 2008 ; Dec. 2005)**

For O-notation we have to show that function is asymptotically bounded by upper bounds as :

$f(n) \le Cg(n)$

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

$= 2^{n+1} \le Cg(n) \Rightarrow 2^n . 2 \le Cg(n)$ ....(2)

$= 2^n . 2 \le 2^n \Rightarrow c \ge 2$

since, $f(n) = 0\ (g(n))$

$\Rightarrow$ $2^{n+1} = 0\ (2^n)$.

(ii) $2^{2n} = 0\ (2^n)$?

Show that $f(n) \le Cg(n)$

$\Rightarrow$ $2^{2n} \le C.2^n$

$\Rightarrow$ $C \le \dfrac{2^{2n}}{2^n}$

$\Rightarrow$ $C = 2^{2n-n} = 2^n$

$\Rightarrow$ $C \ge 2^n$ (not a constant)

∴ does not have any fixed value, because where n change c changed so c is not a constant.

Hence, $2^{2n} \ne 0\ (2^n)$.

Q 37. Consider the recurrence $T_n = 4T(n-1) + 2^n$ with T (0) = 6. Guess the solution and prove it by induction.
(PTU, Dec. 2005)

Ans.

$$T_n = 6 . 4^n + \sum_{i=1}^{n} 4^{n-i} . 2^i$$

$$= 6.4^n + 4^n \sum_{i=1}^{n} 4^{-i} . 2^i$$

$$= 6.4^n + 4^n \sum_{i=1}^{n} \left(\frac{1}{2}\right)^i$$

$$= 6.4^n + 4^n . \frac{1}{2} . \sum_{i=0}^{n-1} \left(\frac{1}{2}\right)^i$$

$$= 6.4^n + \left(1 - \left(\frac{1}{2}\right)^n\right) . 4^n$$

$$= 7.4^n - 2^n.$$

Q 38. Explain how to validate and analyze the algorithms. **(PTU, Dec. 2009 ; May 2008)**

OR

What do you analyze in an algorithm? What is the basis of analysis? Explain. **(PTU, May 2012)**

Ans. In order to learn more about an algorithm, we can 'analyze' it. By this we mean to study the specification of the algorithm and to draw conclusions about how the implementation of that algorithm - the program - will perform in general. But what can we analyze? We can

- Determine the running time of a program as a function of its inputs ;
- Determine the total or maximum memory space needed for program data ;
- Determine the total size of the program code ;
- Determine whether the program correctly computes the desired result ;
- Determine the complexity of the program - e.g., how easy is to read, understand, and modify ; and
- Determine the robustness of the program - e.g. how well does it deal with unexpected or erroneous inputs?

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

Once an algorithm has been devised it become necessary to show that it works it computer the correct to all possible, legal input. One simply way is to code into a program. However, converting the algorithm into program is a time consuming process. Hence, it is essential to be reasonably sure about the effectiveness of the algorithm before it is coded. This process, at the algorithm level, is called 'validation'. Several mathematicaland other empirical method of validation are available. Providing the validation of an algorithm is a fairly complex process and most often a complete theoretical validation though desirable, may not be provided. Alternatively, algorithm segment, which have been proved elsewhere may be used and the overall working algorithm may be empirically validated for several test cases. Such method, although suffice in most cases.

Q 39. Find the Big-Oh notation for the following function :

(i) $4x^2 - 5x + 3$

(ii) $f(x) = (x + 5) \log_2 (3x^2 + 7)$ is $O(x \log_2 x)$

(iii) $f(x) = \dfrac{\left(x^2 + 5 \log_2 x\right)}{(2x+1)}$ **(PTU, May 2012)**

Ans. (i) $f(x) = 4x^2 - 5x + 3$

$|f(x)| = |4x^2 - 5x + 3|$

$\le |4x^2| + |-5x| + |3|$

$\le 4x^2 + 5x + 3$, for all $x > 0$

$\le 4x^2 + 5x^2 + 3x^2$, for all $x > 1$

$\le 12x^2$, for all $x > 1$

We conclude that $f(x)$ is $O(x^2)$. Observe that $C = 12$ and $K = 1$ from the definition of big-O

(ii) $f(x) = (x + 5) \log_2 (3x^2 + 7)$

$|f(x)| = |(x + 5) \log_2 (3x^2 + 7)|$

$= (x + 5) \log_2 (3x^2 + 7)$, for all $x > -5$

$\le (x + 5x) \log_2 (3x^2 + 7x^2)$, for all $x > 1$

$\le 6x \log_2 (10x^2)$, for all $x > 1$

$\le 6x \log_2 (x^3)$, for all $x > 10$

$\le 18x \log_2 x$, for all $x > 10$.

We conclude that $f(x)$ is $O(x \log_2 x)$. Observe that $C = 18$ and $K = 10$ from the definition of big-O.

(iii) $f(x) = \dfrac{\left(x^2 + 5 \log_2 x\right)}{(2x+1)}$

since $\log_2 x < x$ for all $x > 0$, we conclude that

$5 \log_2 x < 5x < 5x^2$, for all $x > 1$

Since $2x + 1 > 2x$, we conclude that

$$\frac{1}{2x+1} < \frac{1}{2x} \text{ for all } x > 0$$

$$\therefore \quad |f(x)| = \left|\frac{x^2 + 5\log_2 x}{2x+1}\right|$$

$$= \frac{x^2 + 5\log_2 x}{2x+1}, \text{ for all } x > 1$$

$$\le \frac{x^2 + 5x^2}{2x}, \text{ for all } x > 1$$

$\le 3x$, for all $x > 1$

We conclude that $f(x)$ is $O(x)$. Obsèrve that $C = 3$ and $K = 1$ from the definition of big-O.

Q 40. Consider the recurrence :

$$T(n) \le 4T(n/2) + n^2$$

$$T(1) = 1$$

Find the solution. **(PTU, Dec. 2004)**

Ans. Let $T(n) = R(n) . n^\alpha$, substitute this value into the original recurrence.

$R(n) . n^\alpha \le 4 (n/2)^\alpha R(n/2) + n^2$

Put $\alpha = 2$, so as to cancle out the multiplicative factor of 4.

$n^2 R(n) \leq 4 (n^2/4) R(n/2) + n^2$

$n^2 R(n) \leq n^2 R(n/2) + n^2$

Dividing through by $n^\alpha = n^2$, we have

$R(n) \leq n^2 R(n/2) + 1$

Since $R(n) \in 0$ (log n), thus,

$T(n) = R(n). n^\alpha \{\alpha = 2\}$

$T(n) = R(n). n^2$

$T(n) \in 0 (n^2 \log n)$.

Q 41. What is re-entrant program? **(PTU, Dec. 2009, 2008 ; May 2009)**

Ans. Re-entrant is an adjective that describes a computer program or routine that is written so that the same copy in memory can be shared by multiple users. Reentrant code is commonly required in operating systems and in applications intended to be shared in multi-use systems. A programmer writes a reentrant program by making sure that no instructions modify the contents of variable values in other instructions within the program. Each time th e program is entered for a user, a data area is obtained in which to keep all the variable values for that user. The data area is in another part of memory from the program itself. When the program is interrupted to give another user a turn to use the program, information about the data area associated with that user is saved. When the interrupted user of the program is once again given control of the program, information in the saved data area is recovered and the program can be reentered without concern that the previous user has changed some instruction within the program.

Q 42. Define non-deterministic algorithm. **(PTU, May 2013)**

Ans. Non-deterministic algorithm : A non-deterministic algorithm is one in which for given input instance each intermediate step has one or more possibilities. This means that there may be more than one path from which the algorithm may arbitrarily choose one. Not all paths terminate successfully to give the desired output. The non-deterministic algorithm works in such a way so as to always choose a path that terminates successfully, thus always giving the correct result. We can also say that algorithm is non-deterministic if there are more than one path the algorithm can take. Due to this, one can not determine the next state of the machine running the algorithm. Example would be a random function.

Q 43. Explain the tradeoff between time and space while analyzing an algorithm. **(PTU, May 2013)**

Ans. A space-time tradeoff refers to a choice between algorithmic solutions of a data processing problem that allow one to decrease the running time of an algorithmic solution by increasing the space to store the data and vice versa. The computation time can be reduced at the cost of increased memory use. As the relative costs of CPU cycles, RAM space, and hard drive space change hard drive space has for some time been getting cheaper at a much faster rate than other components of computers, the appropriate choice for space-time tradeoffs have changed radically. Often, by exploiting a space-time tradeoff, a program can be made to run much faster.

Q 44. What are the criteria that an algorithm should follow? **(PTU, May 2015 ; Dec. 2013)**

Ans. Every algorithm must satisfy the following criteria :

1. Input : There are zero or more quantities which are externally supplied.

2. Output : At least one quantity is produced.

3. Definiteness : Each instruction must be clear and unambiguous.

4. Finiteness : If we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps.

5. Effectiveness : Every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

Q 45. Arrange the following growth order in the increasing order $O(n^3)$, $O(1)$, $O(n \log n)$, $O(n)$, $O(n^2 \log n)$ **(PTU, Dec. 2013)**

Ans. $O(1)$, $O(n)$, $O(n \log n)$, $O(n^2 \log n)$, $O(n^3)$

Q 46. Show that for any real constants a and b, Where b > 0, $(n+a)^b = \theta(n^b)$. **(PTU, Dec. 2017 ; May 2014)**

Ans. $(n+a)^b \leq (n+|a|)^b$, where $n>0$

$\leq (n+n)^b$ for $n \geq |a|$

$= (2n)^b$

$= C_1.n^b$, where $C_1=2^b$

Thus $(n+a)^b = \Omega (n^b)$ ....(1)

$(n+a)b \geq (n-|a|)^b$, where $n>0$

$\geq (C'_2 n)^b$ for $C'_2 = \frac{1}{2}$ where $n \geq 2|a|$

as $n/2 \leq n-|a|$, for $n \geq 2|a|$

Thus $(n+a)b = O(n^b)$ ....(2)

The result follows from 1 and 2 with $C_1 = 2^b$, $C_2 = 2^{-b}$, and no. $\geq 2|a|$.

Q 47. What is difference between an algorithm and a program? **(PTU, May 2014)**

Ans. An algorithm is a step by step outline or flowchart how to solve a problem whereas a program is an implemented coding of a solution to a problem based on the algorithm.

Q 48. Define algorithm validation. **(PTU, May 2016 ; Dec. 2014)**

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

Algorithm validation is the process of computing the correct answer for all possible legal inputs after the algorithm is created or devised. The purpose of the validation is to assure that the algorithm will work correctly independently of the programming languages. Once the validation is done the program can be written and the second phase of the validation, which is referred to as program proving or program verification, begins.

Q 49. What is a Recurrence Equation ? **(PTU, May 2017 ; Dec. 2014)**

Ans. The recurrence equation is an equation that defines a sequence recursively. It is normally in following form :

$$T(n) = T(n-1) + n \text{ for } n>0 \quad \ldots(1)$$

$$T(0) = 0 \quad \ldots(2)$$

Here equation 1 is called recurrence relation and equation 2 is called initial condition.

The recurrence equation can have infinite number of sequences.

The general solution to the recursive function specifies some formula.

For ex : Consider a recurrence relation

$$f(n) = 2f(n-1) + 1 \text{ for } n>1$$

$$f(1) = 1$$

Then by solving this recurrence relation we get $f(n) = 2^n - 1$.

When n = 1, 2, 3 and 4.

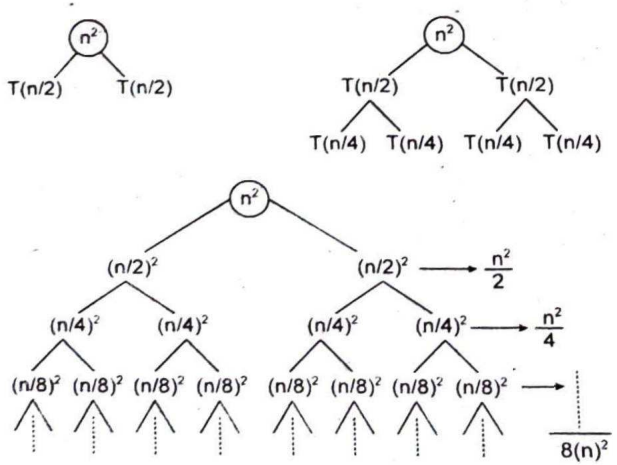
Q 50. Explain recursion tree method with the help of an example.

Ans. Recursion Tree method is a pictorial representation of an iteration method which is in the form of a tree where at each level nodes are expanded. In general, we consider the second term in recurrence as root. It is useful when the divide and congues algorithm is used. It is sometimes difficult to come up with a good guess. In recursion tree, each root and child represents the cost of a single subproblem. We sum the costs within each of the levels of the tree to obtain a set of pre-level costs and then sum all pre-level costs to determine the total cost of all levels of the recursion. A recursion Tree is best used to generate a good guess, which can be verified by the substitution method.

Example : Consider $T(n) = 2T\left(\frac{n}{2}\right) + n^2$

We have to obtain the asymptotic bound using recursion tree method.

Solution :



$$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \ldots \text{ logn times}$$

$$\leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right)$$

$$\leq n^2 \left(\frac{1}{1-\frac{1}{2}}\right) \leq 2n^2$$

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

Q 51. Explain analysis of algorithm through various recurrence relations.

Ans. Many algorithms are recursive in nature. When we analyse them, we get a recurrence relation for time Complex ion. We get running time on an input of size n as a function of n and the running time on input of smaller sizes e.g. in Merge sort, to sort a given array, we divide it in two halves and recurrsively repeat the process for the two halves finally, we merge the results. Time complexity of merge sort can be written as $T(n) = 2T(n/2) + Cn$. There are many other algorithms like search, Tower of Hanoi etc. There are mainly three ways for solving recurrences.

1. Substitution method : We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

e.g. Consider the recurence $T(n) = 2T(n/2) + n$.

We guess the solution as $T(n) = 0(n \log n)$. Now we use induction to prove our guess.

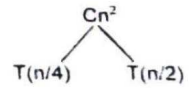
We need to prove that $T(n) <= Cn \log n$. We can assume that it is true for values smaller than n

$$T(n) = 2T(n/2) + n$$
$$= Cn/2 \log Cn/2) + n$$
$$= Cn\log n - Cn\log 2 + n$$
$$= Cn \log n - Cn + n$$
$$<= Cn \log n$$

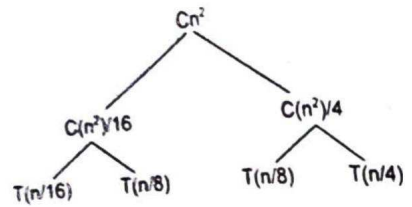
2. Recurrence Tree method : In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically arithmetic or geometric series.

e.g. Consider the recurrence relation

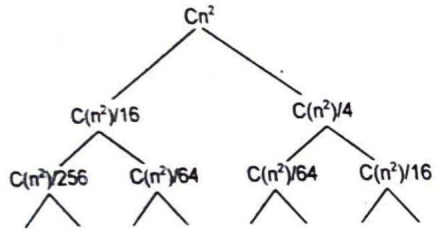
$$T(n) = T(n/4) + T(n/2) + Cn^2$$



If we further break down the express ion T(n/4) and T(n/2), we get following recursion tree.



Breaking down further gives us following :



To know the value of T(n), we need to calculate sum of tree nodes level by level. If we sum the above tree level by level, we get the following series

$$T(n) = C(n^2 + 5(n^2)/16 + 25(n^2)/256) + ...$$

The above series is geometrical progression with ratio 5/16. To get an upper bound we can sum the infinite series we get the sum as $(n^2)/(1 - 5/16)$ which is 0 (n^2)

3. Master Method : Master method is a direct way to get the solution. The faster method works only for following type of recurrences or for occurrences that can be transformed to following type.

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

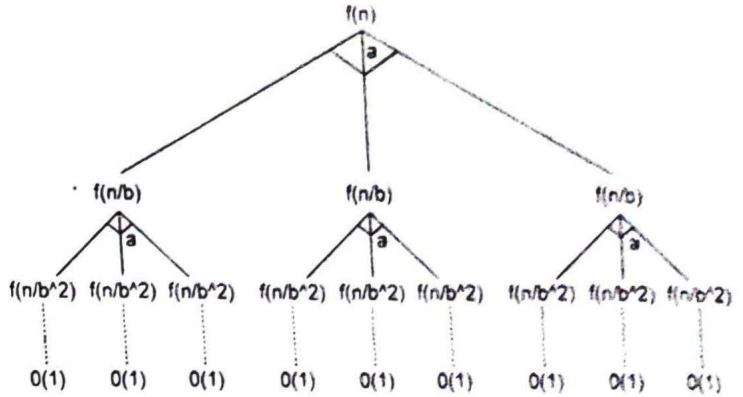
where a > = 1 and b >1

There are following three cases :

1. If $f(n) = Q(n^c)$ Where $C < \log a$ then $T(n) = Q(_n\log_b a)$
2. If $f(n) = Q(n^c)$ Where $C = \log_b a$ then $T(n) = Q(n^c \log n)$
3. If $f(n) = Q(n^c)$ Where $C > \log_b a$ then $T(n) = Q(f(n))$

Master method is mainly derived from recurrance tree method. If we draw recurrence tree of

$T(n) = aT(n/b) + f(n)$, we can see that the work done at root is f(n) and work done at all leaves is $Q(n^c)$ where C is $\log_b a$. And the height of recurrence tree is $\log_b n$.



In recurrence tree method, we calculate total work done. If the work done at leaves is polynomically more, then leaves are the dominant part, and our result becomes the workdone at leaves (case 1). If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (case 2). If work done at root is asymptotically more, then our result becomes work done at root (case 3).

Q 52. What is the time complexity of Conventional matrix multiplication method and Strassen's matrix multiplication method ? (PTU, May 2018 ; Dec. 2017)

Ans. Conventional matrix multiplication method : The time complexity of conventional matrix multiplication method is 0 (n ^ 3).

Strassen's matrix multiplication method : Complexity of strassen's matrix multiplication method is $O(n^{2.81})$.

Q 53. Prove that if $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$. (PTU, May 2018 ; Dec. 2018, 2017)

Ans. Let $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$

This means that there exist constants $C_1, C_2 > 0$ such that $f_1(n) \le C_1 g_1(n)$ and $f_2(n) \le C_2 g_2(n)$

for all n > 0 integers. To prove the claim, we must find some constant C_3 that causes

$$f_1(n) + f_2(n) \le C_3 [g_1(n) + g_2(n)] \text{ for all } n > 0 \text{ integers,}$$

$$f_1(n) + f_2(n) \le C_1 g_1(n) + C_2 g_2(n)$$

$$\le \max(C_1, C_2) g_1(n) + \max(C_1, C_2) g_2(n)$$

$$\le \max(C_1, C_2) [g_1(n) + g_2(n)]$$

$$= C_3 [g_1(n) + g_2(n)]$$

We have found a $C_3 = \max(C_1, C_2)$ that satisfies the definition of big oh proving the claim.

Q 54. What are explicit and implicit constraints ? (PTU, Dec. 2015)

Ans. Explicit constraints : These are rules that restrict each xi to take values only from a given set.

e.g., xi > = 0
xi = 0 or 1

Implicit constraints : These are rules which describes the way in which the xi must relate to each other.

Q 55. What is average case analysis ? **(PTU, Dec. 2015)**

Ans. Average case analysis : Analyze average running time over some distribution of inputs. Other words, Average case analysis requires a notion of an "average" input to an algorithm, which leads to the problem of devising a probability distribution over inputs.

Q 56. Describe an algorithm to perform selection in worst case linear time. **(PTU, Dec. 2017)**

Ans. Following is the algorithm :

1. Divide arr [ ] into [n /5] groups where size of each group is 5 except possibly the last group which may have less than 5 elements.
2. Sort the above created [n/5] groups and find median of all groups. Create an auxilliary array media [ ] and store medians of all [n/5] groups in this median array.
3. Med of Med = K th smallest (median [0 .. [n/5] – 1] [n/10])
4. Position arr [ ] around med of Med and obtain its position.
   pos = partition (arr, n, med or Med)
5. It pos = = k return med of Med
6. It pos < K return K th smallest (arr [1 ... pos 1], k)
7. It pos > k return kth smallest (arr [pos + 1...r], k – pos + l – 1)

Q 57. Differentiate Time complexity from Space complexity. **(PTU, Dec. 2019, 2015)**

Ans.

Time complexity	Space Complexity
1. Time complexity refers to the amount of time spent by the processor for the completion of the task.	1. Space complexity refers to the amount of memory occupied by a specific process or task.
2. The total number of steps involved in a solution to solve a problem is the function of the size of the problem, which is the measure of that problem's time complexity.	2. Space complexity is measured by using polynomial amounts of memory, with an infinite amount of time.
3. Time complexity estimation is based on execution time.	3. Space complexity estimation is based on memory space.

Q 58. State the principle of Substitution method. **(PTU, Dec. 2015)**

Ans. In substitution method, we guess a bound and then use the mathematical induction to prove our guess correct. There are two steps for solving the recurrences by the substitution method :

1. Guess the form of the solution.

2. Using the mathematical induction to find the constants and show that the solution works.

The substitution method can be used to establish either upper or lower bounds on a recurrences. This method is powerful, but it can be applied only in cases, when it is easy to guess the form of the answer.

Q 59. Algorithm A performs $10n^2$ basic operations, and algorithm B performs 300 logn basic operations. For what value of N does algorithm B start to show its better performance ? **(PTU, Dec. 2016)**

Ans. $10n^2 > 300 \log n$

$$n^2 > \frac{300}{10} \log n \qquad n \geq \sqrt{30}$$

$$n^2 > 30 \log n \qquad n \geq 5$$

$$n^2 \geq 30$$

Q 60. State valid shift with reference to string matching. **(PTU, May 2017)**

Ans. We formalize the string matching problem as follows : We assume that the text is an array T[1..n] of length n and that the pattern is an array P [1..m]. We further assume that the elements of P and T are characters drawn from a finite alphabet Σ For example, we may have $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b, ...z\}$. The character arrays P and T are often called string of characters. We say that pattern P occurs with shift s in text T. If $0 \leq S \leq n - m$ and T [S+1. S+m] = P[1..m]. If P occurs with shift S in T, then we call s as valid shift otherwise we call s as an invalid shift.

Q 61. What do you mean by integer arithmetic? **(PTU, Dec. 2018)**

Ans. Integer arithmetic means arithmetic without fractions. A computer performing integer arithmetic ignores any fractions that are derived. For example, 8 divided by 3 would yield the whole number 2.

Q 62. What is order statistics ? **(PTU, May 2019)**

Ans. Order statistics are sample values placed in ascending order. The study of order statistic deals with the applications of there order values and their functions.

Q 63. Take the following list of functions and arrange them in ascending order of growth rate. That is, if function g(n) immediately follows function f(n) in your list, then it should be the case that f(n) is O(g (n)).

$f_1(n) = n^{2.5}$, $f_2(n) = \sqrt{2n}$, $f_3(n) = n + 10$, $f_4(n) = 10^n$, $f_5(n) = 100^n$ and $f_6(n) = n^2 \log n$. **(PTU, May 2019)**

Ans. We can start approching this problem by putting f_4 and f_5 at the end of the list, because these functions are exponential and will grow the fastest. $f_4 < f_5$ because $10 < 1000$.

Other four functions are polynomial and will grow slower than exponential. We can represnt f_1 and f_2 as $n^{2.5} = n^2 \times \sqrt{2n}$; and $\sqrt{2n} = 2n^{0.5}$. Now, we can say that out of all polynomial function f_2 will be the slowest because it has the smallest degree. Moreover, and

will be bounded by f_3 because it has a higher degree of 1. Furthermore, f_1 and f_6 will be between exponent f_4 and f_5 and polynomial f_2 and f_3, because polynomial function grow slower and both f_4 and f_5 and have the highest degree of 2 out of all other polynomial functions and f_6 will be bounded by f_1 because $f_6 = n^2 \log(n)$ and $f_1 = n^2\sqrt{2n}$ and $\log(n) = O(\sqrt{2n})$.

Therefore the final order will be :

$$f_2(n) < f_3(n) < f_6(n) < f_1(n) < f_4(n) < f_5(n)$$

Q 64. If f(n)=n! and g(n)=2n, indicate whether f=0(g), or f=Ω(g), or both (f=θ(g)). **(PTU, May 2019)**

Ans. *If* $f(n) = n!$ *and* $g(n) = 2n$.

Then $f = \Omega(g)$ *because.*

$n! > Z_n$ *for* μ $(n \geq 4)$.

Hence, $f(n) = n!$ *and* $g(n) = 2n$ *indicates* $f = \Omega(g)$.

Q 65. Use the substitution method to prove a tight asymptotic lower bound (Ω - notation) on the solution to the recurrence. **(PTU, Dec. 2019)**

$T(n) = 4T(n/2) + n^2$

Ans. *Let* $T(n) = R(n).\ n^a$, substitute this value into the original recurrence

$$R(n).n^{\alpha} = 4(n/2)^a\ R(n/2) + n^2$$

Put $\alpha = 2$, *so as to cancel out the multiplicative factor of 4*

$$n^2\ R(n) = 4(n^2/4)\ R\ (n/2) + n^2$$

$$n^2 R(n) = n^2\ R(n/2) + n^2$$

Dividing through by $n^{\alpha} = n^2$, *we have*

$$R(n) = n^2\ R(n/2) + 1$$

Since $R(n)\ \Sigma 0\ (\log n)$, *thus*

$$T(n) = R(n).n^{\alpha}\ \{\alpha = 2\}$$

$$T(n) = R(n).\ n^2$$

$T(n)\ \Sigma 0(n^2 \log n)$.

❑❑❑

Chapter 2

Fundamental Algorithmic Strategies

Contents

Brute-Force, Greedy, Dynamic Programming, Branch and Bound and Backtracking methodologies for the design of algorithms; Illustrations of these techniques for Problem-Solving : Bin Packing, Knap Sack, TSP.

POINTS TO REMEMBER

1. Divide and conquer is a top-down technique for designing algorithms.
2. Divide and conquer solve the sub-problem recursively (successively and independently).
3. The divide and conquer paradigm consists of three steps at each level of recursion :
   (i) Divide
   (ii) Conquer
   (iii) Combine
4. Mergesort and quick sort are examples of divide and conquer technique.
5. The merge sort splits the list to be sorted into 2 equal halves and placed them in separated array.
6. Quick sort is divide and conquer strategy that works by partitioning its input elements according to their value relative to some preselected element (pivot).
7. The idea of the dynamic programming developed by Richard Bellman.
8. Dynamic programming is a technique for solving problems with overlapping subproblems.
9. Dynamic programming is an algorithm design method that can be used when a solution to the problem is viewed as the result of sequence of decisions.
10. Knapsack problem is an example of dynamic programming.
11. Dynamic programming is an Bottom-up approach.
12. Greedy algorithm is an Top-down approach.
13. A greedy algorithm is a method for finding a optimal solution to some problem involving large, homogeneous data structure (array, tree, graph).
14. Time complexity of traveling salesman problem is O(n2 2n).
15. Space Complexity of traveling salesman problem is O(n 2n).
16. The searching problem deals with finding a given value, called a search key, in a given set.

17. The sorting problem asks us to rearrange the items of a given list in ascending order (or descending order).
18. Binary search is a technique for searching an ordered list in which we first check the middle item and based on that comparison - "discard" half the data. The same procedure is then applied to the remaining half until a match is found or there are no more item left.
19. The worst-case complexity of binary search is O(lg n).
20. The average-case complexity of binary search is O(lg n).
21. A bubble sort compares two values next to each other and exchange them if necessary to put them in the right order.
22. Bubble sort complexity is $O(n^2)$ and only suitable to sort array with small size of data.
23. Complexity of merge sort is O(n* log (n)).
24. Redix sort is also known as postal sort, bin sort.
25. A heap is a complete binary tree in which each node satisfies the heap condition.
26. There are two types of heap or heap tree. These are :
    (i) Maxheap
    (ii) Minheap
27. Maxheap is also called descending heap.
28. Minheap is also called ascending heap.
29. The complexity of heap sort is O(n* log (n)).
30. Quicksort is similar to mergesort : divide-and-conquer recursive algorithm.
31. Quick sort executes in O(n log n) on average, and $O(n^2)$ in the worst-case.
32. A lower bound of a problem is the least time complexity required for any algorithm which can be used to solve this problem.
    - ❑ Worst case lower bound
    - ❑ Average case lower bound.
33. The selection problem can be solved in O(n log n) time.

QUESTION-ANSWERS

Q 1. What are algorithm design techniques?

Ans. Algorithm design techniques (or strategies or paradigm) are general approaches to solving problems algorithmatically, applicable to a variety of problems from different areas of computing.

General design techniques are :

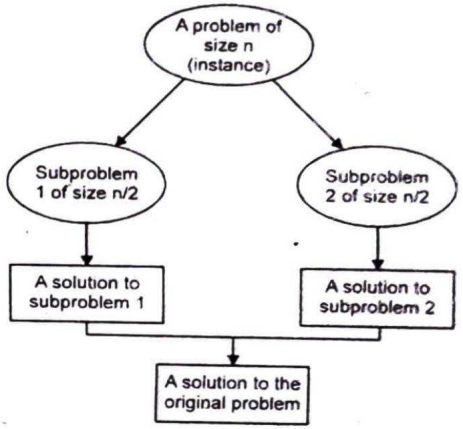
1. Brute force
2. Divide and conquer
3. Decrease and conquer
4. Transform and conquer
5. Greedy technique
6. Dynamic programming
7. Backtracking
8. Branch and bound

Q 2. Give brief concept of divide and conquer.

(PTU, Dec. 2016 ; May 2019, 2018, 2013)

Ans. Divide and conquer is an important general technique for designing algorithms :

1. Divide instance of problem into two or more smaller instances.
2. Solve smaller instances recursively.
3. Obtain solution to original (larger) instance by combining these solutions.



Divide and Conquer Examples

Sorting : Merge sort and quicksort

Q 3. Explain how analysis of linear search is done with a suitable illustration.

(PTU, Dec. 2014)

Ans. Count how many times the key is compared to an array element.

Best case : The key is the first element in the array. Number of comparisons of an array element to the key : 1 = O(1)

Worst case : The key is the last element in the array or the key is not in the array. Number of comparisons : n = O(n)

Average case : The key is equally likely to be in any position in the array.

If the key is in the first array position : 1 comparison

If the key is in the second array position : 2 comparison

. . .

If the key is in the ith position : i comparisons

. . .

So average all these possibilities : (1+2+3+. . . +n)/n = [n(n+1)/2]/n = (n+1)/2 comparisons

The average number of comparisons is (n+1)/2 = O(n).

Q 4. Define merge sort.

Ans. Merge sort is a perfect example of a successful application of the divide and

conquer technique. It sorts a given array A[0 ..... n–1] by dividing it into two halves A [0 ..... [n/2] – 1] and A[[n/2] ..... n – 1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

Q 5. What is the working principle of Mergesort?

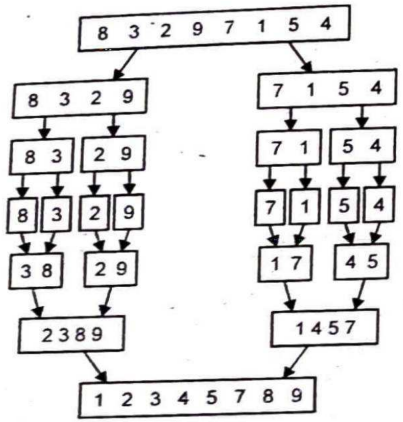
Ans. Using divide-and-conquer, we can obtain a merge sort algorithm.

Divide : Divide the n-elements into two subsequences of n/2 elements each.

Conquer : Sort the two subsequences recursively.

Combine : Merge the two sorted subsequences to produce the sorted answer.

Example :



Q 6. What is quick sort?

Ans. Quick sort is divide and conquer strategy that works by partitioning it's input elements according to their value relative to some preselected element (pivot). It uses recursion and the method is also called partition-exchange sort.

Q 7. Define dynamic programming.

Ans. Dynamic programming : Dynamic programming is an algorithm design method that can be used when a solution to the problem is viewed as the result of sequence of decisions. It is technique for solving problems with overlapping subproblems.

Q 8. What are the features of dynamic programming?

Ans. 1. Optimal solutions to sub problems are retained so as to avoid recomputing of their values.

2. Decision sequences containing subsequences that are sub optimal are not considered.

3. It definitely gives the optimal solution always.

Q 9. Write the general procedure of dynamic programming.

Ans. The development of dynamic programming algorithm can be broken into a sequence of four steps.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of the optimal solution.

3. Compute the value of an optimal solution in the bottom-up fashion.
4. Construct an optimal solution from the computed information.

Q 10. What are the drawbacks of dynamic programming?

Ans. 1. Time and space requirements are high, since storage is needed for all level.

2. Optimality should be checked at all levels.

Q 11. Define principle of optimality. **(PTU, Dec. 2016)**

Ans. Principle of optimality : It states that an optimal sequence of decisions has the property that whenever the initial stage or decisions must constitute an optimal sequence with regard to stage resulting from the first decision.

Q 12. Give an example of dynamic programming.

(PTU, May 2018 ; Dec. 2018, 2007)

Ans. An example of dynamic programming is knapsack problem. The solution to the knapsack problem can be viewed as a result of sequence of decisions. We have to decide the value of Xi for $1 < i < n$. First we make a decision on X1 and then on X2 and so on. An optimal sequence of decisions maximizes the object function Spixi.

Q 13. Explain optimal binary search trees.

Ans. One of the principal application of binary search tree is to implement the operation of searching. If probabilities of searching for elements of a set are known, it is natural to pose a question about an optimal binary search tree for which the average number of comparisons in a search is the smallest possible.

Q 14. Define "0–1 knapsack problem."

Ans. Items are indivisible, you either take an item or not. Given a knapsack with maximum capacity W, and a set S consisting of n items. Each item i has some weight w_i and benefit value b_i (all w_i and W are integer values). Problem is to find max $\sum_{i \in T} b_i$ subject to $\sum_{i \in T} w_i \leq W$

The problem is called a "0–1" problem because each item must be entirely accepted or rejected.

Q 15. How to find actual knapsack item?

Ans. Firstly all of the information we need is in the table.

V[n, W] is the maximum value of items that can be placed in the knapsack.

Let i = n and K = W

if V[i, K] ≠ V [i – 1, K] then

mark the i^{th} item as in the knapsack i = i – 1, K = K –wi

else i = i – 1

Q 16. Write the 0–1 knapsack algorithm and also write the running time of this algorithm.

Ans. For w = 0 to W

V[0, w] = 0

for i = 1 to n

V[i, 0] = 0

for i = 1 to n

for w = 0 to W

if wi < = w // **item i can be part of the solution**

if bi + V[i – 1, w – wi] > V[i – 1, w]

V[i, w] = bi + V[i – 1, w – wi]

else

V[i, w] = V[i – 1, w]

else

V[i, w] = V[i – 1, w] // wi > w

Running time = 0 (n * W)

Q 17. Solve following knapsack 01 problem

n = 4 (number of elements)

W = 5 (max weight)

Elements(weight, benefit) : (2, 3), (3, 4), (4, 5), (5, 6) **(PTU, Dec. 2019, 2014)**

Ans.

Step 1.

i/W	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

for w = 0 to W

V [0,w] = 0

Step 2.

i/W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

for 1 = 1 to n

V [i, 0] = 0

i/W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

i = 1
bi = 3
wi = 2
w = 1
w–wi = –1

i/W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					

i = 1
bi = 3
wi = 2
w = 2
w–wi = 0

i/W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3		
2	0					
3	0					
4	0					

i = 1
bi = 3
wi = 2
w = 3
w–wi = 1

i/W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0					
3	0					
4	0					

i = 1
bi = 3
wi = 2
w = 4
w–wi = 2

i/W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

i = 1
bi = 3
wi = 2
w = 5
w–wi = 3

i/W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0				
3	0					
4	0					

i = 2
bi = 4
wi = 3
w = 1
w–wi = –2

i/W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3			
3	0					
4	0					

i = 2
bi = 4
wi = 3
w = 2
w–wi = –1

i/W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4		
3	0					
4	0					

i = 2
bi = 4
wi = 3
w = 3
w–wi = 1

i/W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	
3	0					
4	0					

i = 2
bi = 4
wi = 3
w = 4
w–wi = 1

i/W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

i = 2
bi = 4
wi = 3
w = 5
w–wi = 2

i/W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4		
4	0					

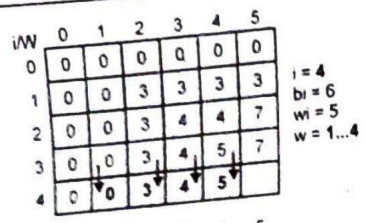
i = 3
bi = 5
wi = 4
w = 1 ... 3

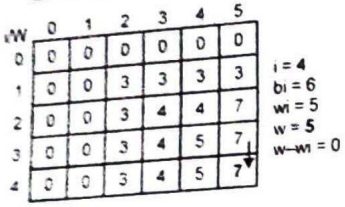
i/W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	
4	0					

i = 3
bi = 5
wi = 4
w = 4
w–wi = 0

i/W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

i = 3
bi = 5
wi = 4
w = 5
w–wi = 1

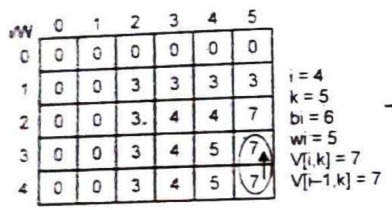


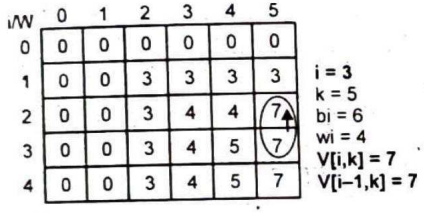


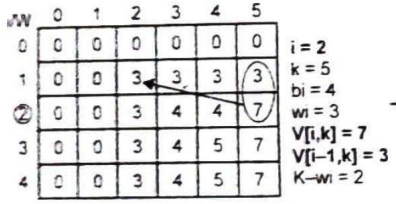
Now we find the item

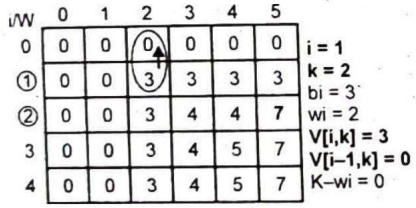
So, **i=4, k=5**, bi=6, wi=5, V[i,k]=7, V[i–1, k]=7

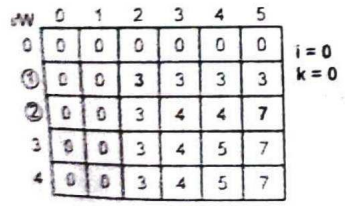
Then

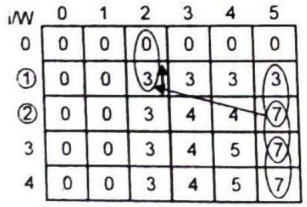












Now the optimal knapsack should contain {1,2}

The optimal knapsack should contain {1,2}

Q 18. What is greedy method? **(PTU, Dec. 2016 ; May 2019, 2013)**

Ans. Greedy method : Greedy method is the most important design technique, which makes a choice that looks best at that moment. A given 'n' inputs are required us to obtain a subset that satisfies some constraints that is the feasible solution. A greedy method suggests that one can device an algorithm that works in stages considering one input at a time.

Q 19. Differentiate between dynamic programming and Greedy algorithms. **(PTU, Dec. 2015)**

Ans.

	Dynamic Programming		Greedy Algorithms
1.	At each step, the choice is determined based on solutions of subproblems.	1.	At each step, we quickly make a choice that currently looks best. It is a local optimal (greedy) choice.
2.	Sub-problems are solved first.	2.	Greedy choice can be made first before solving further sub-problems.
3.	Bottom-up approach.	3.	Top-down approach.
4.	Can be slower, more complex.	4.	Usually faster, simpler.

Q 20. What are the steps for a developing a greedy algorithm? **(PTU, May 2019 ; Dec. 2016)**

Ans. Steps for a developing a greedy algorithm are :

1. Feasible : Here we check whether it satisfies are possible constraints, not to obtain atleast one solution to our problems.

2. Local optimal choice : In this, the choice should be optimum which is selected from the currently available.

3. Unalterable : Once the decision is made at any subsequence step that option is not altered.

Q 21. Define feasible and optimal solution. **(PTU, Dec. 2014)**

Ans. Feasible solution : Given n inputs and we are required to form a subset such that it satisfies some given constraints then such a subset is called feasible solution.

Optimal solution : A feasible solution either maximizes or minimizes the given objective function is called as optimal solution.

Q 22. Write the control abstraction for greedy method.

Ans. Algorithm Greedy(a, n)

```
{
solution = 0 ;
for i = 1 to n do
{
x = select(a) ;
if feasible(solution, x) then
solution = Union(solution, x) ;
}
return solution ;
}
```

Q 23. What is the greedy choice property?

Ans. A globally optimal solution can arrive at by making a locally optimal choice. The choice made by greedy algorithm depends on choices made so far but it can not depend on any future choices or on solution to the sub problem. It progresses in top down fashion.

Q 24. Give the general characteristics of greedy algorithm. **(PTU, Dec. 2009)**

Ans. To solve a problem in an optimal way construct the solution from given set of candidates. As the algorithm proceeds, two other sets get accumulated among this one set contains the candidates that have been already considered and chosen while the other set contains the candidates that have been considered but rejected.

Q 25. Let S = {a, b, c, d, e, f, g} denote a set of objects with weights and benefits as given in the table below. What is an optimal solution to the fractional knapsack problem for S assuming that we have a sack that can hold objects with total weight 18?

Item	A	B	C	D	E	F	G
benefits	12	10	8	11	14	7	9
Weight (Kg)	4	6	5	7	3	1	6

Carrying capacity W = 18 Kg.

Ans. First we must calculate the "value" for the each items, which is defined as value = benefits/weights.

So,

Item	A	B	C	D	E	F	G
benefits	12	10	8	11	14	7	9
Weight	4	6	5	7	3	1	6
Value	3	1.67	1.6	1.57	4.67	7	1.5

Now sort this table according to the decreasing value

Item	F	E	A	B	C	D	G
benefits	7	14	12	10	8	11	9
Weight	1	3	4	6	5	7	6
Value	7	4.67	3	1.67	1.6	1.57	1.5

W=18

Initially

Knapsack ⌊ ⌋ W=18

Weight = 0

benefit = 0

Item	F	E	A	B	C	D	G
benefits	7	14	12	10	8	11	9
Weight	1	3	4	6	5	7	6
Value	7	4.67	3	1.67	1.6	1.57	1.5

W = 18

Now select maximum valued item 'F', Here (Weight + w[F]) < W

Put whole item 'F' into knapsack. Add weight[F] with weight and benefit [F] with benefit.

So, Knapsack ⌊ F ⌋ W = 18

Weight = 1

benefits = 7

Item	F	E	A	B	C	D	G
benefits	7	14	12	10	8	11	9
Weight	1	3	4	6	5	7	6
Value	7	4.67	3	1.67	1.6	1.57	1.5

W = 18

Now select next maximum valued item 'E', Here (Weight + w[E]) < W

Put whole item 'E' into knapsack. Add weight[E] with weight and benefit[E] with benefit.

So, Knapsack ⌊ F,E ⌋ W = 18

Weight = 1 + 3 = 4

benefits = 7 + 14 = 21

Item	F	E	A	B	C	D	G
benefits	7	14	12	10	8	11	9
Weight	1	3	4	6	5	7	6
Value	7	4.67	3	1.67	1.6	1.57	1.5

W = 18

Now select next maximum value item 'A', Here (weight + w[A])<= W

Put whole item 'A' into knapsack. Add weight[A] with weight and benefit[A] with benefit.

So, Knapsack ⌊ F,E,A ⌋ W = 18

Weight = 1 + 3 + 4 = 8

benefits = 7 + 14 + 12 = 33

Item	F	E	A	B	C	D	G
benefits	7	14	12	10	8	11	9
Weight	1	3	4	6	5	7	6
Value	7	4.67	3	1.67	1.6	1.57	1.5

W = 18

Now select next maximum valued item 'B', here (weight + w[B]) <= W.

Put whole item 'B' into knapsack. Add weight[B] with weight and benefit[B] with bene

So, Knapsack ⌊ F,E,A,B ⌋ W = 18

Weight = 1 + 3 + 4 + 6 = 14

benefits = 7 + 14 + 12 + 10 = 43

Item	F	E	A	B	C	D	G
benefits	7	14	12	10	8	11	9
Weight	1	3	4	6	5	7	6
Value	7	4.67	3	1.67	1.6	1.57	1.5

W=18

Now select next maximum valued item 'C', Here (Weight + w[C]) $\not<=$ W. Put whole item 'B' into knapsack. And calculate weight and benefit as follows :

needed weight= W – Weight = 18 – 14 = 4

Put whole item 'B' into knapsack

Knapsack [F,E,A,B,C] W = 18

Weight = W = 18

benefits = 7 + 14 + 12 + 10 + (needed weight)*

Value [C] = 43 + (4 *1.6) = 43 + 6.4 = 49.4

Item	F	E	A	B	C	D	G
benefits	7	14	12	10	8	11	9
Weight	1	3	4	6	5	7	6
Value	7	4.67	3	1.67	1.6	1.57	1.5

W=18

Remaining items D,G could not put into knapsack (bag) because bag is full i.e. weight = W

Knapsack = [F,E,A,B,C]

Weight in Bag = W= 18

Benefits = Rs 49.4

Q 26. Write the algorithm for fractional knapsack problem.

Ans. Fractional knapsack problem :

```
Greedy-fractional-knapsack(Item[n], w[ ] , b[ ], W)
{
Knap = 0
Weight = 0
Benefit = 0
for each item i
v[i] = b[i] / w[i]
while(weight < = W)
{
i = Extract item of maximum value from list
if(weight + w[i] ≤ W)
{
knap = knap ∪ item[i]
weight = weight + w[i]
benefit = benefit + v[i]
}
else
{
knap = knap ∪ item[i]
weight = W
```



```
benefit = (W – weight) * v[i] / w[i]
}
}
return x
}
```

Q 27. Explain traveling salesman problem.

Ans. In traveling salesman problem a salesman has to travel n cities starting from any one of the cities and visit the remaining cities exactly once and come back to the city where he started his journey in such a manner that either the distance is minimum or cost is minimum. This is known as traveling salesman problem.

Q 28. Write some applications of traveling salesman problem.

Ans. 1. Routing a postal van to pick up mail from boxes located at n different sites.

2. Using a robot arm to tighten the nuts on some piece of machinery on an assembly line.

3. Production environment in which several commodities are manufactured on the same set of machines.

Q 29. Give the time and space complexity of traveling salesman problem.

Ans. Time Complexity : O(n2 2n)

Space Complexity : O(n 2n)

Q 30. What is Greedy method? State and write algorithm for Knapsack problem using Greedy method. **(PTU, Dec. 2018, 2011 ; May 2017, 2008)**

Ans. Greedy method is a method of choosing a subset of the dataset as the solution set that results in some profit. Consider a problem having n injputs, we are required to obtain the solution which is a series of subsets that satisfy some constraints or conditions. Any subset, which satisfies these constraints, is called a feasible solution. It is required to obtain the feasible solution that maximizes or minimizes the objective function. This feasible solution finally obtained is called optimal solution.

If one can devise an algorithm that works in stages, considering one input at a time and at each stage, a decision is taken on whether the data chosen results with an optimal solution or not. If the inclusion of a particular data results with an optimal solution, then the data is added into the partial solution set. On the other hand, if the inclusion of that data results with infeasible solution then the data is eliminated from the solution set.

Knapsack problem

- Input : n objects and a knapsack
- Each object i has a weight **wi** and the knapsack has a capacity **m**
- A fraction of an object xi, $0 \le xi \le 1$ yields a profit of pi.xi
- Objective is to obtain a filling that maximizes the profit, under the weight constraint of **m**
- Formally,

Maximize $\sum_{i=1}^{n} p_i . x_i$

subject to $\sum_{i=1}^{n} w_i . x_i \leq m$

and $0 \geq x_i \leq 1$, $1 \leq i \leq n$

and Each $p_i > 0$ and $w_i > 0$

- Problems instance : n = 3, m = 20, P = (25, 25, 15) and W = (18, 15, 10).
- Greedy strategy 1 : Pick items with maximum profit per item.
  Solution. (1, 2/15, 0). Profit : 28.2
- Greedy strategy 2 : Pick as many items as possible (smallest weight items first).
  Solution. (0, 2/3, 1). Profit : 31
- Greedy strategy 3 : Pick items with maximum profit per unit weight.
  Solution. (0, 1, 1/2). Profit : 31.5
- Items considered in the objective function : total profit, capacity used, and ratio of accumulated profit to capacity used

Algorithm

```
void greedy_knapsack (m, n)
{
// Solution vector is x [i], 0 <= i < n
for (i = 0 ; i < n l i ++)
x [i] = 0.0 ;
U = m ;//Unused capacity
for (i = 0 ; (i < n) && (w [i] <= U) ; i++)
{
x [i] = 1.0 ;
U = U – w [i] ;
}
if (i < n)
x [i] = U/w [i] ;
}
```

Q 31. Does greedy algorithm always give an optimal solution? Give arguments to support your answer. **(PTU, May 2013, 2009)**

Ans. A greedy algorithm always makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. Greedy algorithms do not always yield optimal solutions, but for some problems they are very efficient.

Greedy algorithms are typically used to solve optimization problem. Most of these problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We are required to find a feasible solution that either minimizes or maximizes a given objective function. In the most common situation we have :

- C : A set (or list) of candidates ;
- S : The set of candidates that have already been used
- feasible ( ) : A function that checks if a candidate can contribute to a solution
- solution ( ) : A function that checks if a set provides a solution
- select ( ) : A function for choosing most promising candidate
- An objective function that we are trying to optimize.
  Example : Coin change
- We want to give change to a customer using the smallest number of coins (of units 1, 5, 10, and 25, resp.).
- Greedy algorithm will always find the optimal solution in this case.
- If 12-unit coins are added, it will not necessarily find the optimal solution
  e.g. = 15 = (12, 1, 1), (10, 5) is optimal.
- Greedy method might even fail to find a solution despite the fact that one exists (Consider coins of 2, 3 and 5 units).
  e.g. 6 = 5 + ? (3, 3) is optimal

Q 32. Let n = 4 (P_1, P_2, P_3, P_4) = (100, 10, 15, 27) and (d_1, d_2, d_3, d_4) = (2, 1, 2, 1) where Pi and profits on processes or job and di are deadline of completion. Find out the optimal schedule. **(PTU, Dec. 2005)**

Ans.

S.No.	Feasible Soluton	Processing Sequence	Value
1.	(1, 2)	(2, 1)	110
2.	(1, 3)	(1, 3) or (3, 1)	115
3.	(1, 4)	(4, 1)	127
4.	(2, 3)	(2, 3)	25
5.	(3, 4)	(4, 3)	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

Thus, the optimal solution s solution 3 (i.e. 127) with schedule (4, 1) and profit 127.

Q 33. Consider four items along with their respective weights and values
I = <i1, i2, i3, i4>
w = <7, 3, 4, 5>
v = <49, 12, 42, 30>
The capacity of the knapsack W = 10. Find the solution for the fractional knapsack problem using greedy method. **(PTU, May 2012)**

Ans. This is fractional knapsack problem. The item can be selected fractionally. First of all we will obtain value to weight ratio and arrange the item in non increasing order.

Item	Weight	Value	Value to weight
3	4	#42	10.5
1	7	$49	7
4	5	$30	6
2	3	$12	4

To fulfill the capacity W = 10 we will have

- add item of weight 4
- select item of weight 7 and take its fractional 6/7 weight

∴ Weights of selected items

$\therefore 4 + 7 \times \frac{6}{7} = 10$ which fits into the knapsack

Hence, the profit obtained will be

$$42 + 49 \times \frac{6}{7} = 42 + 42 = \$84$$

This is an optimal solution to given instance of knapsack.

Q 34. Differentiate between dynamic programming and divide and conquer technique. **(PTU, Dec. 2005)**

Ans. 1. Both solve a problem through combining the solutions of the subproblems.

2. Subproblem independence (YES for Divide and Conquer, NO for Dynamic Programming).

3. Divide and conquer does more work than necessary by solving the common subproblems, while DP solves each subproblem just once and saves the solution in a table.

Q 35. What is the working principle of quicksort? **(PTU, May 2012)**

Ans. Quicksort also called partition exchange sort, designed to improve and resolve the deficiencies of the selection sort. The quicksort is based on three main strategies :

(a) Split (divide) the array into small subarrays.

(b) Sort the subarrays.

(c) Merge (join/concatenate) the sorted subarrays.

Q 36. Differentiate between top down and bottom up approaches. **(PTU, May 2010 ; Dec. 2008)**

Ans. A top-down approach (also known as step-wise design) is essentially the breaking down of a system to gain insight into its compositional sub-systems. In a top-down approach an overview of the system is formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. A top-down model is often specified with the assistance of 'black boxes', these make it easier to manipulate. However, black boxes may fail to elucidate elementary mechanisms or be detailed enough to realistically validate the model.

A **bottom-up** approach is the piecing together of systems to give rise to grander systems, thus making the original systems sub-systems of the emergent system. In a bottom-up

approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. This strategy often resembles a 'seed' model, whereby the beginnings are small but eventually grow in complexity and completeness. However, 'organic strategies' may result in a tangle of elements and subsystems, developed in isolation and subject to local optimization as opposed to meeting a global purpose.

Q 37. What are important characteristics of dynamic programming? **(PTU, May 2012)**

Ans. Important characteristics of dynamic programming :

1. The problem can be divided into stages with a decision required at each stage.
2. Each stage has a number of states associated with it.
3. The decision at one stage transforms one state into a state in the next stage.
4. Given the current state, the optimal decision for each of the remaining states does not depend on the previous stages or decisions.
5. There exists a recursive relationship that identifies the optimal decision for stage j, given that stage j + 1 has already been solved.
6. The final stage must be solvable by itself.

Q 38. Describe the matrix multiplication algorithm for multiplying A and B matrix in dynamic programming

where $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 0 \end{bmatrix}_{2\times3}$ **and** $\begin{bmatrix} 7 & 10 \\ 8 & 11 \\ 9 & 12 \end{bmatrix}_{3\times2}$. **(PTU, Dec. 2007)**

Ans. The multiplication of A and B is defined as

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} * \begin{bmatrix} 7 & 10 \\ 8 & 11 \\ 9 & 12 \end{bmatrix} = \begin{bmatrix} 1\times7+2\times8+3\times9 & 1\times10+2\times11+3\times12 \\ 4\times7+5\times8+6\times9 & 4\times10+5\times11+6\times12 \end{bmatrix}$$

$$A \times B = C$$

$$2 \times 3 * 3 \times 2 = 2 \times 2$$

Therefore when we multiply a matrix whose order is 2 × 3 to matrix 3 × 2 then we get order 2 × 2 of the resultant matrix.

Algorithm : Matrix multiplication (A, B)

1. if cal [A] ≠ row [B]
2. then error "can't be multiply"
3. else for i ← 1 to row [A] (P)
4. do for j ← 1 to cal [B] (r)
5. do C [i, j] ← 0

6. for K ← 1 to cal [A] [q]
7. do C [i, j] ← C [i, j] + A [i, K] * B [K, j]
8. return C [i, j]

Therefore, $A_{i \times K} * B_{K \times j} = C_{i \times j}$

Q 39. What are the advantages of dynamic programming over the greedy method? (PTU, May 2007)

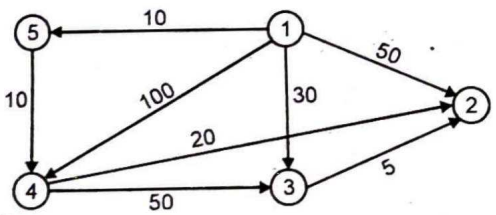
OR

***Differentiate between greedy and dynamic programming method of problem solving.* (PTU, Dec. 2016, 2011 ; May 2011)**

Ans. Greedy vs. Dynamic Programming :

- Both techniques are optimization techniques, and both build solutions from a collection of choices of individual elements.
- The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices.
- Dynamic programming computes its solution bottom up by synthesizing them from smaller subsolutions, and by trying many possibilities and choices before it arrives at the optimal set of choices.
- There is no a prior litmus test by which one can tell if the Greedy method will lead to an optimal solution.
- By contrast, there is a litmus test for Dynamic Programming, called The Principle of Optimality.

Q 40. Find the shortest path from node 1 to all vertices of the graph given below. Show all the intermediate steps. The numbers on the edges are the weights. (PTU, Dec. 2008)



Ans. Given a weighted connected graph (undirected or directed), the all pairs shortest paths problem asks to find the distances (the lengths of the shortest path) from each vertex to all other vertices.

Q 41. Describe the dynamic programming algorithm for computing the minimum cost. (PTU, May 2010)

Ans. A Dynamic Programming Algorithm : To begin, let's assume that all we really want to know is the minimum cost, or minimum number of arithmetic operations, needed to multiply out of the matrices. If we're only multiplying two matrices, there's only one way to multiply them, so the minimum cost is the cost of doing this. In general, we can find the

minimum cost using the following recursive algorithm :

- Take the sequence of matrices and separate it into two subsequences.
- Find the minimum cost of multiplying out each subsequence.
- Add these costs together, and add in the cost of multiplying the two result matrices.
- Do this for each possible position at which the sequence of matrices can be split, and take the minimum over all of them.

For example, if we have four matrices ABCD, we compute the cost required to find each of (A) (BCD), (AB) (CD), and (ABC) (D), making recursive calls to find the minimum cost to compute ABC, AB, CD, and BCD. We then choose the best one. Better still, this yields not only the minimum cost, but also demonstrates the best way of doing the multiplication : just group it the way that yields the lowest total cost, and do the same for each factor.

Unfortunately, if we implement this algorithm we discover that it's just as slow as the naive way of trying all permutations! What went wrong? The answer is that we're doing a log of redundant work. For example, above we made a recursive call to find the best cost for computing both ABC and AB. But finding the best cost for computing ABC also requires finding the best cost for AB. As the recursion grows deeper, more and more of this type of unnecessary repetition occurs.

One simple solution is called memorization : each time we compute the minimum cost needed to multiply out a specific subsequence, we save it. If we are ever asked to compute it again, we simply give the saved answer, and do not recomputed it. Since, there are about $n^2/2$ different subsequences, where n is the number of matrices, the space required to do this is reasonable. It can be shown that this simple trick brings the runtime down to O (n^3) from O (2^n), which is more than efficient enough for real applications. This is top-down dynamic programming.

Q 42. Solve all pair shortest path problem by using dynamic programming. (PTU, May 2011)

Ans. When a weighted graph, represented by its weight matrix W the objective is to find the distance between every pair of nodes.

We will apply dynamic programming to solve all pairs shortest path.

Step 1. We will decompose the given problem into subproblems. Let $A^K_{(i,j)}$ be the length of shortest path from node i to node f such that the label for every intermediate node will be < = K. We will compute A^K for K = 1 .... n for n nodes.

Step 2. For solving all pair shortest path, the principle of optimality is used. That means any subpath of shortest. Path is a shortest path between the end nodes. Divide the path from i node to j node for every intermediate node, say 'K'. Then there arises two cases.

(i) Path going from i to j via K.

(ii) Path which is not going via K. Select only shortest path from two cases.

Step 3. The shortest path can be computed using bottom up computation method following is recursion method.

Initially : $A^0 = W [i, j]$

Next computations :

$$A^k_{(i,j)} = \min\left\{A^{K-1}_{(i,j)}, A^{K-1}_{(i,k)} + A^{K-1}_{(K,j)}\right\}.$$

Q 43. What do you mean by dynamic programming? Explain all pair shortest path problem with example. **(PTU, May 2007)**

Ans. In mathematics and computer science, **dynamic programming** is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable to problems exhibiting the properties of overlapping subproblems which are only slightly smaller and optimal substructure (described below). When applicable, the method takes far less time than naive methods.

The key idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution. Often, many of these subproblems are really the same. The dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations. This is especially useful when the number of repeating subproblems is exponentially large.

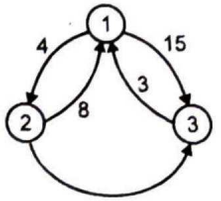
Top-down dynamic programming simply means storing the results of certain calculations, which are later used again since the completed calculation is a sub-problem of a larger calculation. Bottom-up dynamic programming involves formulating a complex calculation as a recursive series of simpler calculations.

All-pairs shortest path problem : The all-pairs shortest path problem can be considered the mother of all routing problems. It aims to compute the shortet path from each vertex v to every other u. Using standard single-source algorithms, you can expect to get a naive implementation of O (n ^ 3) if you use Dijkstra for example – i.e. running a O (n ^ 2) process n times. Likewise, if you use the Bellman-Ford-Moore algorithm on a dense graph, it'll take about O (n ^ 4), but handle negative arc-lengths too.

Storing all the paths explicity can be very memory expensive indeed, as you need one spanning tree for each vertex. This is often impractical in terms of memory consumption, so these are usually considered as all-pairs shortest distance problems, which aim to find just the distance from each to each node to another.

The result of this operation is an **n*n** matrix, which stores estimated distances to the each node. This has many problems when the matrix get too big, as the algorithm will scale very poorly.

Q 44. Compute all pair shortest path for the following graph. (PTU, Dec. 2004)



Ans.

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array}\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 15 \\ 8 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array}\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 15 \\ 8 & 0 & 2 \\ 3 & (7) & 0 \end{bmatrix} \rightarrow \min(\infty, (3+4))$$

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array}\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & (6) \\ 8 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \rightarrow \min(15, (4+2))$$

$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array}\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 6 \\ (5) & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

A^3 gives shortest distances between any pair of vertices.

Q 45. What do you mean by dynamic programming? Explain assignment problem with example. **(PTU, Dec. 2015 ; May 2009)**

Ans. Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable to problems exhibiting the properties of overlapping subproblems which are only slightly smaller and optimal substructure (described below). When applicable, the method takes far less time than naive methods.

The key idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solutions. Often, many of these subproblems are really the same. The dynamic programming approach seeks to solve each sub problem only once, thus reducing the number of computations. This is especially useful when the number of repeating subproblems is exponentially large.

Top-down dynamic programming simply means storing the results of certain calculations, which are later again since the completed calculation is a sub-problem of a larger calculation. Bottom-up dynamic programming involves formulating a complex calculation as a recursive series of simpler calculations.

Assignment Problem : The assignment problem can be stated as a problem where different jobs are to be assigned to different machines on the basis of the cost of doing these jobs. The objective is to minimize the total cost of doing all the jobs on different machines. The peculiarity of the assignment problem is only one job can be assigned to one machine i.e., it should be a one-to-one assignment. The cost data is given as a matrix where rows correspond to jobs and columns to machines and there are as many rows as the number of columns i.e. the number of jobs and number of machines should be equal.

Example : Four persons A, B, C, and D are to be assigned four jobs I, II, III and IV. The cost matrix is given as under, find the proper assignment.

Man/jobs	A	B	C	D
I	8	10	17	9
II	3	8	5	6
III	10	12	11	9
IV	6	13	9	7

Solution : In order to find the proper assignment we apply the Hungarian algorithm as follows :

I (A) Row reduction

Man/jobs	A	B	C	D
I	0	2	9	1
II	0	5	2	3
III	1	3	2	0
IV	0	7	3	1

I (B) Column reduction

Man/jobs	A	B	C	D
I	0	0	7	1
II	0	3	0	3
III	1	1	0	0
IV	0	5	1	1

II (A) and (B) zero assignment

Man/jobs	A	B	C	D
I	~~0~~	[0]	7	1
II	~~0~~	3	[0]	3
III	1	1	~~0~~	[0]
IV	[0]	5	1	1

In this way all the zero's are either crossed out or assigned. Also total assigned zero's = 4 (i.e. number of rows or columns). Thus, the assignment is optimal.

From the table we get I → B ; II → C : III → D and IV → A.

Q 46. What do you mean by dynamic programming? Explain with the help of suitable examples. **(PTU, May 2019, 2015, 2013 ; Dec. 2011, 2010)**

OR

What is dynamic programming? How is this approach different from recursion? Explain. **(PTU, Dec. 2018, 2013 ; May 2012)**

Ans. Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable to problems exhibiting the properties of overlapping subproblems which are only slightly smaller and optimal substructure (described below). When applicable, the method takes far less time than naive methods.

The key idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solutions. Often, many of these subproblems are really the same. The dynamic programming approach seeks to solve each sub problem only once, thus reducing the number of computations. This is especially useful when the number of repeating subproblems is exponentially large.

Top-down dynamic programming simply means storing the results of certain calculations, which are later again since the completed calculation is a sub-problem of a larger calculation. Bottom-up dynamic programming involves formulating a complex calculation as a recursive series of simpler calculations.

Difference Between DP and Recursion : The essential difference is the DP keeps its intermediate results where as recursion does not. This makes a huge difference to performance when a recursion function is called repeatedly with the same arguments. In fact dynamic programming is nothing more than recursion with the addition of a caching strategy. For the sequence comparison algorithm the caching strategy was to use a 2D array. In other situations sparse arrays and hashing are more appropriate.

Q 47. What is swapping? Explain. **(PTU, Dec. 2004)**

Ans. To replace pages or segments of data in memory. Swapping is a useful technique that enables a computer to execute programs and manipulate data files larger than main memory. The operating system copies as much data as possible into main memory and leaves the rest on the disk. When the operating system needs data from the disk, it exchanges a portion of data (called a page or segment) in main memory with a portion of data on the disk. DOS does not perform swapping, but most other operating system, including OS/2, Windows, and UNIX, do. Swapping is often called paging. In UNIX systems, swapping refers to moving entire processes in and out of main memory.

Q 48. What is recursion? What are its drawbacks? **(PTU, May 2007)**

Ans. Recursion : Divide-and-conquer algorithms are naturally implemented as recursive procedures. In that case, the partial sub-problems leading to the one currently being solved are automatically stored in the procedure call stack.

Q 49. State greedy strategy. **(PTU, May 2015)**

Ans. ❑ The greedy strategy is an algorithm design technique Like divide & Conquer

The greedy algorithms are used to solve optimization problems.

❑ The goal is to find the best solution.

❑ Works when the problem has the greedy choice property

A global optimum can be reached by making locally optimun choices.

We can also say that Greedy is a strategy that works well on optimization problems with the following characteristics :

1. Greedy-choice property : A global optimum can be arrived at by selecting a local optimum.

2. Optimal substructure : An optical solution to the problem contains an optimal solution to subproblems.

Q 50. Discuss the use of D and C in quicksort algorithm. **(PTU, May 2011)**

Ans. Quicksort is a divide and conquer sorting algorithm in which division is dynamically carried out (as opposed to static division in merge sort).

The three steps of quicksort are as follow :

Divide : Rearrange the elements and split the array into two sub-arrays and an element in between such that each element in the left subarray is less than or equal the middle element and each element in the right subarray is greater than the middle element.

Conquer : Recursively sort the two subarrays.

Combine : None.

Algorithm

Quicksort (A, n)

1 : Quicksort' (A, 1, n)

Quicksort' (A, P, r)

1 : if P ≥ r then return

2 : q = partition (A, P, r)

3 : Quicksort' (A, P, q – 1)

4 : Quicksort' (A, q + 1, r).

Q 51. Write algorithm for travelling sales person problems using dynamic programming. **(PTU, Dec. 2006)**

Ans. Algorithm for travelling sales person problem :

Problem Description : Let G be directed graph denoted by (V, E) and where V denotes set of vertices and E denotes set of edges. The edges are given along with their cost C_{ij}. The cost $C_{ij} > 0$ for all i and j. If there is no edge between i and j then $C_{ij} = \infty$.

A tour for the graph should be such that all the vertices should be visited only once and cost of the tour is sum of cost of edges on the tour. The traveling sales person problem is to find the tour of minimum cost.

Dynamic programming is used to solve this problem.

Step 1. Let the function C (1, V – {1}) is the total length of the tour terminating at 1. The objective of TSP problem is that the cost of this tour should be minimum. Let d [i, j] be the shortest path between two vertices i and j.

Step 2. Let $V_1, V_2 \ldots V_n$ be the sequence of vertices followed in optimal tour. Then (V_1, V_2, ... V_n) must be a shortest path from V_1 to V_n which passes through each vertex exactly once.

Here the principle of optimality is used. The path V_i, V_{i+1} ..... V_j must be optimal for all paths beginning at V (i), ending at V (j), and passing through all the intermediate vertices $\{V_{i+1} \ldots V_{j-1}\}$ once.

Step 3. Following formula can be used to obtain the optimum cost tour.

Cost (i, j) = min {d [i, j] + cost (j, s – {j}) } where $j \in S$ and $i \in S$

Q 52. Write algorithm for quick sort using divide and conquer. What is divide and conquer algorithm? Use this algorithm to find the maximum and minimum from a given array. **(PTU, Dec. 2018 ; May 2007)**

Ans. Divide-and-conquer is a top-down technique for designing algorithms that consists of dividing the problem into smaller subproblems hoping that the solutions of the subproblems are easier to find and then composing the partial solutions into the solution of the original problem.

Little more formally, divide-and-conquer paradigm consists of following major phases:

❑ Breaking the problem into several sub-problems that are similar to the original problem but smaller in size,

❑ Solve the sub-problem recursively (successively and independently), and then

❑ Combine these solutions to subproblems to create a solution to the original problem.

Binary Search (Simplest application of divide-and-conquer)

Binary search is an extremely well-known instance of divide-and-conquer paradigm. Given an ordered array of n elements, the basic idea of binary search is that for a given element we "probe" the middle element of the array. We continue in either the lower or upper segment of the array, depending on the outcome of the probe until we reached the required (given) element.

Quick sort is a divide and conquer algorithm. Quick sort first divides a large list into two smaller sub-lists : the low elements and the high elements. Quick sort can then recursively sort the sub-lists.

The steps are :

1. Pick an element, called a pivot, from the list.

2. Reorder the list so that all element with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either away). After this partitioning, the pivot is in its final position. This is called the partition operation.

3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

The base case of the recursion are lists of size zero or one, which never need to be sorted.

Q 53. Explain the role of randomization in designing an algorithm.
(PTU, May 2019, 2017 ; Dec. 2016, 2013)

Ans. A randomized algorithm is an algorithm that employs a degree of randomness as part of its logic. Randomization has played an important role in the design of both sequential and parallel algorithm.

Q 54. What are the drawbacks of Greedy Algorithm ? **(PTU, Dec. 2014)**

Ans. The greedy approach does not always work because greedy algorithms only make locally optimal choices. This will often lead to a local maximum in the solution space that isn't the best solution. In certain cases, you can give an upper bound for the worst solution the greedy algorithm will return (e.g. O(k log n) for set cover), but in other cases, no bound may be found. In many of these instances, dynamic programming will be a more robust approach that can return the optimal solution.

Q 55. What are the applications of dynamic programming ? **(PTU, Dec. 2014)**

Ans.
- Bioinformatics
- Control theory
- Information theory
- Operations research
- Computer Science : Theory, graphics, AI, systems, ...

Q 56. Define Brute force approach ? **(PTU, Dec. 2014)**

Ans. Brute force is a straight forward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved. The "force" implied by the strategy's definition is that of a computer and not that of one's intellect, "just do it!" would be another way to describe the prescription of the brute-force approach. And often, the brute force strategy is indeed the one that is easiest to apply.

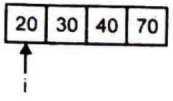
Q 57. Write a pseudo code for divide & conquer algorithm for merging two sorted arrays in to a single sorted one. Explain with example. **(PTU, Dec. 2014)**

Ans. Algorithm MergeSort(int A[0....n–1],low,high)
//Problem Description : This algorithm is for sorting the elements using mergesort
//Input : Array A of unsorted elements, low as beginning pointer of array A and high as end pointer of array A
//Output : Sorted array A[0....n–1]
if(low < high) then
{
mid ← (low + high)/2 //split the list at mid
MergeSort(A, low, mid) //first sublist
MergeSort(A,mid+1,high)//second sublist
Combine(A,low,mid,high)//merging of two sublists
}
Algorithm Combine (A[0....n–1], low, mid, high)
{
k ←low; //k as index for array temp

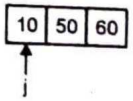
i ←low; //i as index for left sublist of array A
j ←mid +1 //j as index for right sublist of array A.
while (i< = mid and j < = high) do
{
if(A [i] < = A [j]) then
//if smaller element is present in left sublist
{
//copy that smaller elemet to temp array
temp [k] ←A[i]
i ←i+1
k ←k+1
}
else //smaller element is present in right sublist
{
//copy that smaller element to temp array
temp[k] ←A[j]
j ←j+1
k ←k+1
}
}
//copy remaining elements of left sublist to temp
while (i<=mid) do
{
temp[k] ←A[i]
i ←i+1
k ←k+1
}
//copy remaining elements of right sublist to temp
while(j < = high)do
{
temp[k] ←A[j]
j ←j+1
k ←k+1
}

Consider that at some instance we have got two sublist 20,30,40,70 and 10,50,60, then

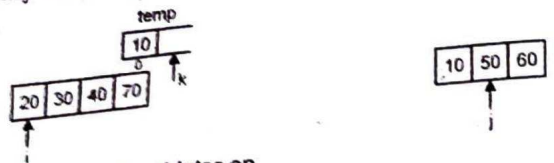
Array A (left sublist)



Array A (right sublist)

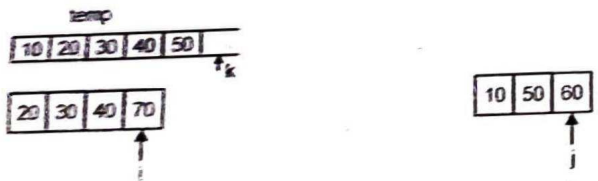


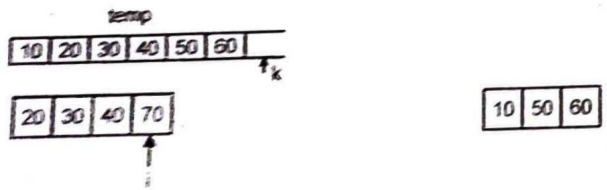
Initially k=0, Then k will be incremented

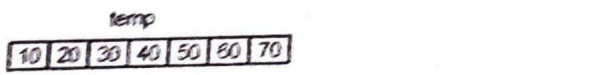


K = 1, It is advanced later on

Now K=2







Q 58. Differentiate between optimization problem and decision problem.
(PTU, May 2015)

Ans. Optimization problem : The problem of finding the best solution from all feasible solutions is an optimization problem.

- Optimization problem involves the identification of an optimal solution i.e. either maximum or minimum.
- Similarly, an algorithm which is used to solve an optimization problem is called optimization algorithm.
- Optimization problems have corresponding decision problems meaning that many optimization problems can be recast into decision problems that ask whether there is a feasible solution.
- However, a decision problem can be solved in polynomial time if the optimization problem can.
- Typically optimization problems can be solved using branch and bound.

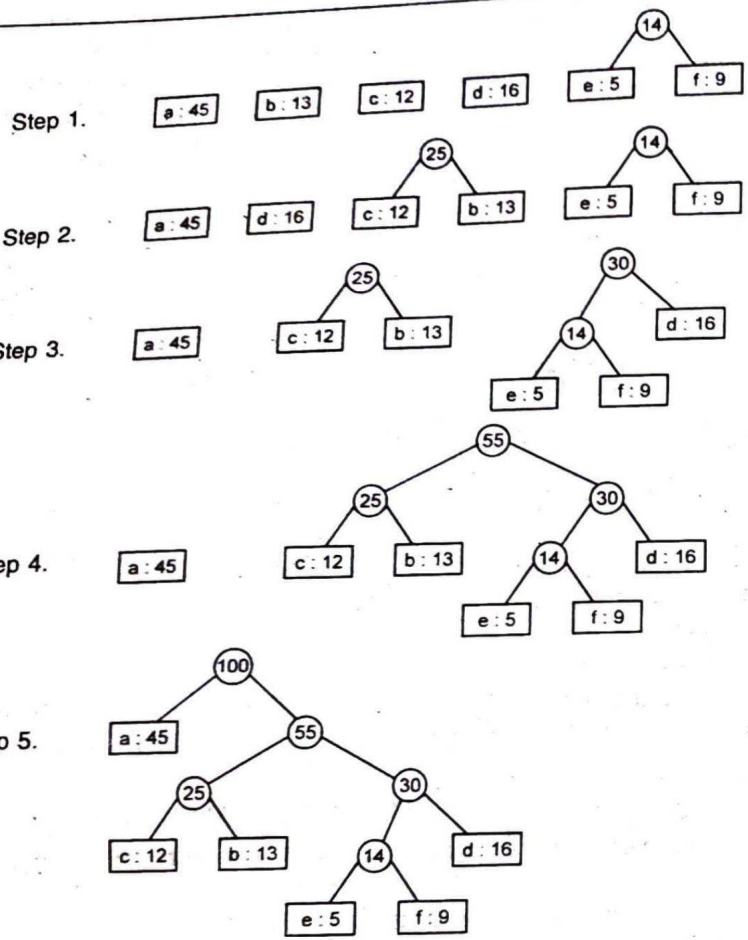
Decision problems :

- Any problem for which the answer is either yes or no depending on the values of some input parameters is called a decision problem.
- These input parameters can be natural numbers as well as strings of a formal language. Instead of yes/no sometimes is also uses I/O, success/failure, or true/false as output.
- Outputs of decision problems are Boolean.
- An algorithm for solving a decision problem is termed as decision algorithm or procedure for that problem and the problem is then called decidable or effectively solvable.
- Typically decision problems can be solved using backtracking.

Q 59. Explain how you can use greedy technique for Huffman coding.
(PTU, Dec. 2014)

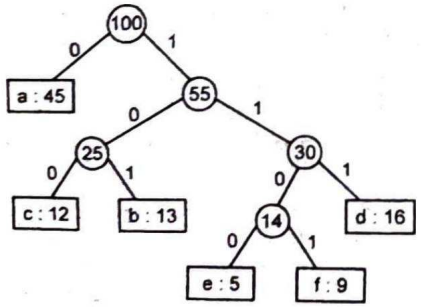
Ans. According to Huffman algorithm, a bottom up tree is built starting from the leaves. Initially, there are n singleton trees in the forest, as each tree is a leaf. The greedy strategy first finds two trees having minimum frequency of occurrences. Then these two trees are merged in a single tree where the frequency of this tree is the total sum of two merged trees. The whole process is repeated, until there is only one tree in the forest.

Let us consider a set of characters S = <a,b,c,d,e,f> with the frequency of occurrences P = <45,13,12,16,5,9>. Initially, these six characters with their frequencies are considered six singleton trees in the forest. The stepwise merging of these trees to a single tree is shown in figure below. The merging is done by selecting two trees with minimum frequencies, till there is only tree in the forest.



Stepwise merging of the singleton trees

Now, the left branch is assigned a code '0', and right branch is assigned a code '1'. Then,



The binary code word for a character is interpreted as path from the root to that character. Hence, the codes for the characters are as follows :

a = 0
b = 101
c = 100
d = 111
e = 1100
f = 1101

Therefore, it is seen that no code is the prefix of other code. Suppose we have a code 01111001101, to decode the binary code word for a character. We traverse the tree. The first character is 0, and the character at which the tree traversal terminates is a. Then, the next bit is 1 for which the tree is traversed right. Since, it has not reached at the leaf node, the tree is next traversed right for the next bit 1. Similarly, the tree is traversed for all the bits of the code string. When the tree traversal terminates at a leaf node, it again starts from the root for the next bit of the code string. The character string after decoding is 'adcf'.

Q 60. What is dynamic programming technique ? How does it differ from divide and conquer technique ? **(PTU, Dec. 2015)**

Ans. Dynamic Programming : *Refer to Q.No. 46*

Dynamic Programming	Divide and conquer
1. In dynamic programming many decision sequences are generated and all the overlapping subinstances are considered.	1. The problem is divided into small subproblems. These subproblems are solved independently. Finally all the solutions of subproblems are collected together to get the solution to the given problem.
2. In dynamic computing duplications in solutions is avoided totally.	2. In this method duplications in subsolutions are neglected, i.e. duplicate subsolutions may be obtained.
3. Dynamic programming is efficient than divide and conquer strategy.	3. Divide and conquer is less efficient because of rework on solutions.
4. Dynamic programming uses bottom up approach of problem solving (Iterative method)	4. The divide and conquer uses top down approach of problem solving (recursive methods).
5. Dynamic programming splits its input at every possible split points rather than at a particular point. After trying all split points it determines which split point is optimal.	5. Divide and conquer splits its input at specific deterministic points usually in the middle.

Q 61. What are the advantages of brute force technique ? (PTU, Dec. 2015)

Ans. The various advantages of brute force technique are

1. Brute force applicable to a very wide variety of problems. It is used for many elementary but important algorithmic tasks.
2. For some important problems this approach yields reasonable algorithms of at least some practical value with no limitation on instance size.
3. The expense to design a more efficient algorithm may be unjustifiable if only a few instances of problems need to be solved and a brute force algorithm can solve those instances with acceptable speed.
4. Even if inefficient in general it can still be used for solving small size instances of a problem.
5. It can serve as a yardstick with which to judge more efficient alternatives for solving a problem.

Q 62. Distinguish between decision, counting and optimization problems and give examples. (PTU, Dec. 2015)

Ans. Decision Problems : In this class of problems, the output is either 'yes' or 'no'. For example, whether a given number is prime is a decision problem.

Counting problems : The output of this class of algorithms is a natural number. For example, given a number how many distinct factorization of the number are there.

Optimization problems : This class of algorithms optimizes some objective function based on the problem instance. For example, given a weighted connected graph, finding a minimal spanning tree is an optimization problem.

Q 63. Describe divide and conquer strategy for multiplying two n-bit numbers. Derive its time complexity. (PTU, Dec. 2015)

Ans. Multiplying two n-bit numbers :

$$xy = (2^{n/2} x_L + x_R)(2^{n/2} y_L + Y_R)$$
$$= 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

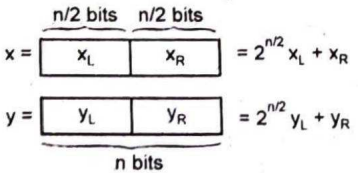
So# n/2 – bit products : 4

bit shifts (by n or n/2 bits) : 2

additions (at most 2n bits long) : 3

we can compute the n/2 - bit products recursively.

Let T(n) be the overall running time for n-bit inputs.Then



$$T(n) = \begin{cases} 0(1) & \text{if } n = 1 \\ 4T\left(\frac{n}{2}\right) + 0(n) & \text{otherwise} \end{cases} = 0(n^2)$$

Q 64. What do you mean by binary search? Explain with an appropriate example and procedure. (PTU, May 2014 ; Dec. 2004)

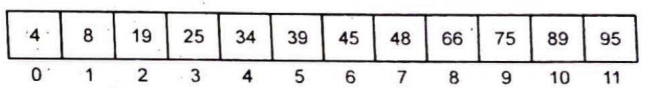
Ans. Binary search : Binary search is a technique for searching an ordered list in which we first check the middle item and based on that comparison "discard" half the data.

The same procedure is then applied to the remaining half until a match is found or there are no more items left.

In binary search the array list is divided into two equal parts (approximately). If the elements of array list are in ascending order then if desired element is less than the middle element of array list, it will never be present in the second half of the array. Similarly, if the desired element is greater than the middle element of array, it will never be present in the first half of the array list.

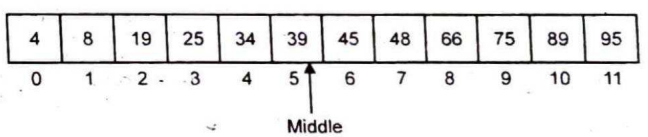
Thus we can focus our attention on one half of the array list. The process is repeated and in the next stage we have to search for the element only is one quarter of the array list. This process reduce the search length and search time.

Example of Binary Search :



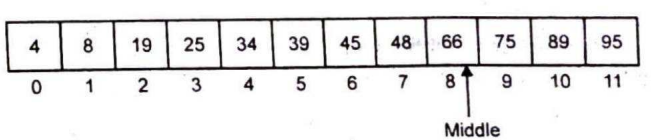
Suppose we want to search 66, then in first pass :

$$\text{Middle} = \frac{(0+11)}{2} = \frac{11}{2} = 5$$



66 is greater than the middle element, then process is repeated in second half

$$\text{Middle} = \frac{(6+11)}{2} = \frac{17}{2} = 8$$



Now the position is found that is 8.

Q 65. Write down the algorithm of binary search. (PTU, May 2014 ; Dec. 2004)

Ans. Algorithm of Binary Search :

Step 1. [Initialize]
Low = 0
High = n – 1
[Here, Low = represent the lower limit
High = represent the upper limit]

Step 2. [Perform search]
Repeat thru step 4 while Low < = High

Step 3. [Obtain index of midpoint of interval]
Middle= INT((Low + High)/2)

Step 4. [Compare]
if Element < List [Middle]
[Element = given element, List = array]
then High = Middle – 1
else
if Element > List [Middle]
then
low = Middle + 1
else
write ('Successful Search')
pos = Middle
[when pos is the position of given element]

Step 5. [Unsuccessful search]
Write ('Unsuccessful search')
pos = null
Step 6. Exit.

Q 66. Write the worst and average case complexity of the binary search.

Ans. Worst case : The worst case complexity of binary search is O(log n).

Average case : The average case complexity of binary search is O(log n).

Q 67. What do you mean by 'Sorting' problem?

Ans. Sorting problem : The sorting problem asks us to rearrange the items of a given list in ascending order (or descending order).

Q 68. What do you mean by 'Searching' problem?

Ans. Searching problem : The searching problem deals with finding a given value, called a search key, in a given set.

Q 69. Explain bubble sort with example.

Ans. Bubble Sort : A bubble sort compares two values next to each other and exchange them if necessary to put them in the right order.

It keeps passing through the array [a_0, ..., a_{N-1}] exchanging each pair of adjacent elements (a_{j-1}, a_j) which are out of order ($a_{j-1} > a_j$).

Why does it works?

- During the first pass the largest element is exchanged with each of the elements to its right and gets into position a_{N-1}.
- After the second pass the second largest gets into position a_{N-2}, ...
- After step K, the sub-array [a_{N-k}, ..., a_{N-1}] is ordered, we need to continue on the interval [0, N – k – 1].
- When no more exchanges are required; the array is sorted.

Sorting Activities for Bubble :

- Go through multiple passes over the array.
- In every pass :
  (a) Compare adjacent elements in the list.
  (b) Exchange the elements if they are out of order.
  (c) Each pass moves the largest (or smallest) elements to the end of the array.
- Repeating this process in several passes eventually sorts the array into ascending order.

Example of Bubble Sort : 5 3 1 9 8 2 4 7

5 3 1 9 8 2 4 7	
3 5 1 9 8 2 4 7	
3 1 5 9 8 2 4 7	Pass 1
3 1 5 9 8 2 4 7	
3 1 5 8 9 2 4 7	
3 1 5 8 2 9 4 7	
3 1 5 8 2 4 9 7	
3 1 5 8 2 4 7 **9**	
1 3 5 8 2 4 7	
1 3 5 8 2 4 7	Pass 2
1 3 5 8 2 4 7	
1 3 5 2 8 4 7	
1 3 5 2 4 8 7	
1 3 5 2 4 7 **8**	
1 3 5 2 4 7	
1 3 5 2 4 7	Pass 3
1 3 2 5 4 7	
1 3 2 4 5 7	
1 3 2 4 5 **7**	
1 3 2 4 5	Pass 4
1 2 3 4 5	
1 2 3 4 5	
1 2 3 4 **5**	
1 2 3 4	Pass 5
1 2 3 4	
1 2 3 **4**	
1 2 3	Pass 6
1 2 **3**	Pass 7
1 **2**	

Bubble sort complexity is $O(n^2)$ and only suitable to sort array with small size of data.

Q 70. Explain Bubble sort time complexity in average, best and worst case.

Ans. Bubble sort average time complexity in number of comparisons : The average number of comparisons is N(N – 1)/2. We count the number of comparisons needed by the algorithm.

- At the first step, we need N–1 comparisons to put the largest element at position N–1.
- At the second step we only need N–2 comparisons. We avoid comparing elements with the last one.

Summing up : (N – 1) + (N – 2) + ... + 1 = N(N – 1)/2.

Bubble sort time complexity in best case : We count the number of comparisons needed by the algorithm.

Best Case : Bubble sort on an already sorted array :

- It does like for the average case N(N – 1)/2 comparisons.
- During the iterations on the array : 0 exchange.

Bubble Sort Time Complexity in Worst Case : We count the number of comparisons needed by the algorithm.

Worst Case : Array already sorted in reverse order :

- It does like for the average case N(N – 1)/2 comparisons.
- It does a exchange each time it does a comparison N(N – 1)/2 exchanges.

Q 71. Write down the algorithm of Bubble Sort for fixed number of passes.

Ans. BubbleSort(x, n)

Where x = Represents the list of elements
n = Represents the number of elements in the list

Step 1. [Initialize]
i=0

Step 2. Repeat through step 7 while (i < n–1).

Step 3. j = 0

Step 4. Repeat through step 6 while (j < n–i–1).

Step 5. If(x[j] > x[j + 1]
temp = x[j]
x[j] = x [j + 1]
[x j + 1] = temp

Step 6. j++

Step 7. i++

Step 8. Exit.

Q 72. Explain mergesort with the help of example.

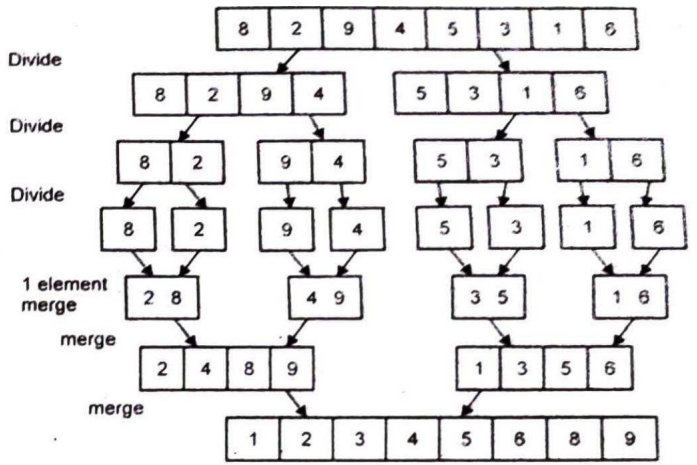
Ans. Merge Sort is an O(n log n) comparison based sorting algorithm. Merge sort is the first example we see of a divide-and-conquer algorithm. To sort an array, merge sort first chops it up into two halves, sorts these recursively and then merges together the result.

Basic Plan :

1. Divide the data elements into two sections with equal number of elements.

2. Sort the two sections separately.
3. Merge the two-sorted sections into a single sorted collection.

Example :



Complexity of mergesort is O(n* log (n))

Q 73. Explain the analysis for the mergesort.

Ans. Algorithm :

```
Mergesort(int [ ] a, int left, int right)
{
if(right > left)
{
middle = left + (right – left)/ 2;
mergesort (a, left, middle) ;
mergesort (a, middle + 1, right) ;
merge (a, left, middle, right) ;
}
}
```

Assumption : N is a power of two.

For N = 1 : time is a constant (denoted by 1)

Otherwise : time to mergesort N elements = time to mergesort N/2 elements plus time to merge two arrays each N/2 elements.

Time to merge two arrays each N/2 elements is linear, i.e. N

Thus we have :

1. T (1) = 1
2. T (N) = 2T (N/2) + N

Next we will solve this recurrence relation, first we divide (2) by N :

3. $\frac{T(N)}{N} = T(N/2)/(N/2) + 1$

N is a power of two, so we can write

4. $T(N/2)/(N/2) = T(N/4)/(N/4) + 1$
5. $T(N/4)/(N/4) = T(N/8)/(N/8) + 1$
6. $T(N/8)/(N/8) = T(N/16)/(N/16) + 1$
7. .....
8. $T(2)/2 = T(1)/1 + 1$

Now we add equations (3) through (8) : the sum of their left-hand sides will be equal to the sum of their right-hand sides :

$$T(N)/N + T(N/2)/(N/2) + T(N/4)/(N/4) + ... + T(2)/2 = T(N/2)/(N/2) + T(N/4)/(N/4) + ... + T(2)/2 + T(1)/1 + \log N$$

(Log N is the sum of 1s in the right-hand sides)

After crossing the equal term, we get

9. $T(N)/N = T(1)/1 + \log N$

T (1) is 1, hence we obtain.

10. $T(N) = N + N \log N = O(N \log N)$

Hence the complexity of the mergesort algorithm is O(N log N).

Q 74. Define Radix sort and also explain Radix sort algorithm.

Ans. Radix sort : Radix sort puts the elements in order by comparing the digits of the numbers. Sort objects based on some key value found with in the object. Most often used when keys are strings of the same length, or positive integers with the same number of digits. Also known as postal sort, bin sort.

Radix sort algorithm :

- Let us suppose keys are K-digit integers.
- Radix sort uses an array of 10 queues, one for each digit 0 through 9.
- Each object is placed into the queue whose index is the least significant digit (the 1's digit) of the object key.
- Objects are then dequeued from these 10 queues, in order 0 through 9, and put back in the original queue/list/array container ; they are sorted by the last digit of the key.
- Process is repeated, this time using the 10's digit instead of the 1's digit ; values are now sorted by last two digits of the key.
- Keep repeating using the 100's digit, then the 1000's digit, then the 10,000's digit, ...
- Stop after using the most significant (10^{n-1}'s) digit.
- Objects are now in order in original container.

Q 75. Give an example of Radix sort.

Ans. Let us consider the following 9 numbers :

493 812 715 710 195 437 582 340 385

We should start sorting by comparing and ordering the one's digits :

	Digit	Sublist
4 9 **3**	0	710 340
8 1 **2**	1	
7 1 **5**	2	812 582
7 1 **0**	3	493
1 9 **5**	4	
4 3 **7**	5	715 195 385
5 8 **2**	6	
3 4 **0**	7	437
3 8 **5**	8	
	9	

Now, we gather the sublist into the main list again :

710, 340, 812, 582, 493, 715, 195, 385, 437

Now the sublist are created again, this time based on the ten's digit

	Digit	Sublist
7 **1** 0	0	
3 **4** 0	1	710 812 715
8 **1** 2	2	
5 **8** 2	3	437
4 **9** 3	4	340
7 **1** 5	5	
1 **9** 5	6	
3 **8** 5	7	
4 **3** 7	8	582 385
	9	493 195

Now the sublists are gathered in order from 0 to 9 :

710 812 715 437 340 582 385 493 195

Finally, the sublists are created according to the hundred's digit.

	Digit	Sublist
7 1 0	0	
8 1 2	1	195
7 1 5	2	
4 3 7	3	340
3 4 0	4	437 385 493
5 8 2	5	582
3 8 5	6	
4 9 3	7	710 715
1 9 5	8	812
	9	

At last, the list is gathered up again :

195, 340, 385, 437, 493, 582, 710, 715, 812

And now we have a fully sorted array.

Q 76. What is the running time of Radix sort?

Ans. Let there be d digits in input integers. Radix sort takes O(d*(n + b)) time where b is the base for representing numbers, for example, for decimal system, b is 10. What is the value of d? If K is the maximum possible value, then d would be $O(\log_b(R))$. So overall time complexity is $O((n + b) * \log_b (K))$.

Which looks more than the time complexity of comparison based sorting algorithm for a large k. Let us first limit k. Let $k <= n^c$. Where c is a constant.

In that case ; the complexity becomes $O(n \log_b (n))$. But it still does not beat comparison based sorting algorithms. What if we make value of b larger? What should be the value of b to make the time complexity linear? If we set b as n, we get the time complexity as O(n). In other words, we can sort an array of integers with range from 1 to n^c if the numbers are represented in base n (or every digit takes $\log_2(n)$ bits).

Q 77. What is a heap? Define maxheap or minheap. **(PTU, May 2015, 2014)**

Ans. Heap : A heap is a complete binary tree in which each node satisfies the heap condition. There are two types of heaps or heap tree.

1. Maxheap
2. Minheap

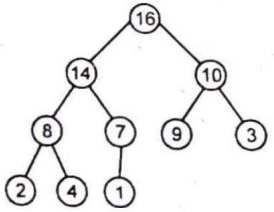
Maxheap : Maxheap is also called descending heap. It is a complete binary tree in which every node has a value greater than or equal to value of every child of that node.

Minheap : Minheap is also called ascending heap. It is a complete binary tree in which every node has a value less than or equal to value of its every child of that node.

Q 78. Give an appropriate example of heap sort.

Ans. Replace that root with last node of heap tree then keep the root at proper position i.e. always keep nodes value should be equal to or greater than all its children.

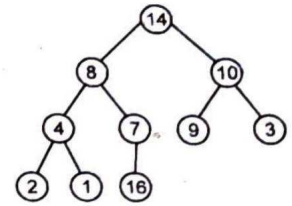
Now, let us consider the following example :



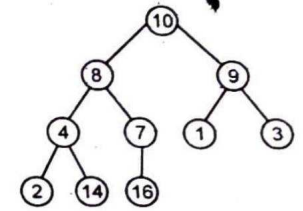
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Now the root node is 16 and last node is 1. Delete the highest item 16 from its position and place it at the end of the array in place of item 1. Now promote next lower item 14 in place of item 16 and higher sub tree 8 of item 14 in place of item 14. Fill the place of sub tree 8 with item 7.

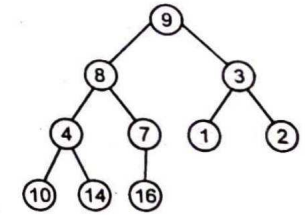
This process continue untill all the items are sorted.



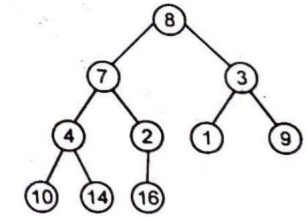
1	2	3	4	5	6	7	8	9	10
14	8	10	4	7	9	3	2	1	16



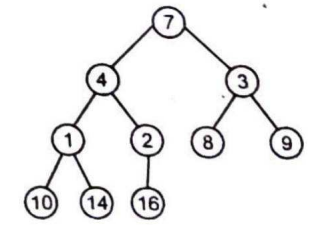
1	2	3	4	5	6	7	8	9	10
10	8	9	4	7	1	3	2	14	16



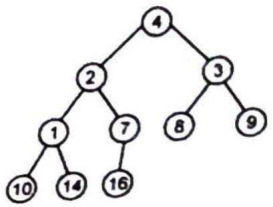
1	2	3	4	5	6	7	8	9	10
9	8	3	4	7	1	2	10	14	16



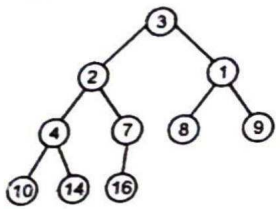
1	2	3	4	5	6	7	8	9	10
8	7	3	4	2	1	9	10	14	16



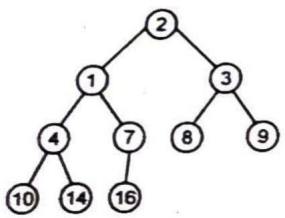
1	2	3	4	5	6	7	8	9	10
7	4	3	1	2	8	9	10	14	16



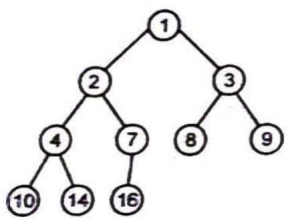
1	2	3	4	5	6	7	8	9	10
4	2	3	1	7	8	9	10	14	16



1	2	3	4	5	6	7	8	9	10
3	2	1	4	7	8	9	10	14	16



1	2	3	4	5	6	7	8	9	10
2	1	3	4	7	8	9	10	14	16



1	2	3	4	5	6	7	8	9	10
1	2	3	4	7	8	9	10	14	16

Q 79. Write down the algorithm of heap sort. **(PTU, May 2015)**

Ans. Heat_sort(data,. n)

where data = Represents the list of elements.

n = Represents number of elements in the list.

Step 1. [Create initial heap]
(call Heap_Creation(data, n)

Step 2. [Start sort]
Repeat through step 10 for k = n, n – 1, ..., 2

Step 3. [Interchange elements]
data[1] = data[k]

Step 4. temp = data[1]
i = 1
j = 2

Step 5. [Find index of largest child of new element]
if j + 1 < k then
if data[j + 1] > data[j] then
j = j + 1

Step 6. [Recreate the new heap]
Repeat through step 10 while j <= k–1 and data[j] > temp

Step 7. [Interchange element]
data[i] = data[j]

Step 8. [Obtain left child]
i = j
j = 2*i

Step 9. [Obtain index of next largest child]
if j + 1 < k
if data[j + 1] > data[j] then j = j + 1 else if j > n
then j = 1

Step 10. [Copy element into its proper place]
data[j] = temp

Step 11. Exit.

Q 80. Write down the complexity of heap sort.

Ans. O(n * log (n))

Q 81. Briefly describe the basic idea of quicksort.

Ans. Quicksort is similar to mergesort; divide-and-conquer recursive algorithm. It is the one of the fastest sorting algorithms Quick sort executes in O(n log n) on average, and $O(n^2)$ in the worst-case.

Basic Idea :

1. Pick one element in the array, which will be the pivot.
2. Make one pass through the array, called a partition step, re-arranging the entries so that :
   (a) The pivot is in its proper place.
   (b) Entries smaller than the pivot are to the left of the pivot.
   (c) Entries larger than the pivot are to its right.
3. Recursively apply quicksort to the part of the array that is to the left of the pivot, and to the right part of the array.

Here we don't have the merge step, at the end all the elements are in the proper order.

Q 82. Write the average, best and worst case complexity for the quick sort.

Ans. Average Case : O(N* log (N))

Best Case : O(N* log (N))

The best case is when the pivot is the median of the array, and then the left and the right part will have same size.

There are logN partitions, and to obtain each partitions we do N comparisons (and not more than N/2 swaps). Hence the complexity is O(N*log (N)).

Worst-case : $O(N^2)$

This happens when the pivot is the smallest (or the largest) element. Then one of the partitions is empty, and we repeat recursively the procedure for n–1 elements.

Q 83. Explain the worst, best and average case analysis for the quick sort.

Ans. Analysis :

$$T(N) = T(i) + T(N - i - 1) + cN$$

- The time to sort the left partition with i elements, plus
- The time to sort the right partition with N – i – 1 elements, plus
- The time to build the partitions.

Worst case analysis : The pivot is the smallest element

$$T(N) = T(N - 1) + cN, N > 1$$

Telescoping :

$$T(N - 1) = T(N - 2) + C(N - 1)$$
$$T(N - 2) = T(N - 3) + C(N - 2)$$
$$T(N - 3) = T(N - 4) + C(N - 3)$$
$$T(2) = T(1) + C.2$$

Add all equations

$$T(N) + T(N-1) + T(N-2) + \ldots + T(2) = T(N-1) + T(N-2) + \ldots T(2) + T(1) + C(N) + C(N-1) + C(N-2) + \ldots + C.2$$

$$T(N) = T(1) + C(2 + 3 + \ldots + N)$$
$$T(N) = 1 + C(N(N+1)/2 - 1)$$

Therefore $T(N) = O(N^2)$

Best case analysis : The pivot is in the middle

$$T(N) = 2T(N/2) + cN$$

Divide by N :

$$T(N)/N = T(N/2) / (N/2) + C$$

Telescoping :

$$T(N/2) / (N/2) = T(N/4) / (N/4) + C$$
$$T(N/4) / (N/4) = T(N/8) / (N/8) + C$$

.....

$$T(2)/2 = T(1) / (1) + C$$

Add all equations :

$$T(N)/N + T(N/2) / (N/2) + T(N/4) / (N/4) + \ldots + T(2)/2 = (N/2) / (N/2) + T(N/4) / (N/4) + \ldots + T(1) / (1) + c \log N$$

After crossing the equal terms :

$$T(N)/N = T(1) + c \text{ Log } N = 1 + c \text{ Log } N$$
$$T(N) = N + Nc\log N$$

Therefore $T(N) = 0(N \log N)$

Average Case Analysis :

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

The average value of T(1) is 1/N times the sum of T(0) through T(N – 1)

$$\frac{1}{N}\sum T(j), j = 0 \text{ thru } N-1$$

$$T(N) = \frac{2}{N}\left(\sum T(j)\right) + cN$$

Multiply by N

$$NT(N) = 2\left(\sum T(j)\right) + CN * N$$

To remove the summation, we rewrite the equation for N – 1 :

$$(N-1)T(N-1) = 2\left(\sum T(j)\right) + C(N-1)^2, j = 0 \text{ thru } N-2 \text{ and subtract.}$$

$$NT(N) - (N-1)T(N-1) = 2T(N-1) + 2cN - c$$

Prepare for telescoping

Rearrange terms, drop the insignificant C :

$$NT(N) = (N+1)T(N-1) + 2cN$$

Divide by N(N + 1) :

$$\frac{T(N)}{(N+1)} = T(N-1)/N + 2c/(N+1)$$

Telescope :

$$\frac{T(N)}{(N+1)} = T(N-1)/N + 2c/(N+1)$$

$$\frac{T(N-1)}{(N)} = T(N-2)/(N-1) + 2c/(N)$$

$$\frac{T(N-2)}{(N-1)} = T(N-3)/(N-2) + 2c/(N-1)$$

.....

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

Add the equations and cross equal terms :

$$\frac{T(N)}{(N+1)} = \frac{T(1)}{2} + 2c\sum\left(\frac{1}{j}\right), j = 3 \text{ to } N+1$$

$$T(N) = (N+1)\left(\frac{1}{2} + 2c\sum\left(\frac{1}{j}\right)\right)$$

The sum $\sum\left(\frac{1}{j}\right), j = 3$ to $N - 1$ is about log N.

Thus $T(N) = O(N \log N)$.

Q 84. What are the advantages and disadvantages of quicksort?

Ans. Advantages of Quicksort :

1. One of the fastest algorithms on average.
2. Does not need additional memory (the sorting takes place in the array – this is called in place processing). Compare with mergesort, mergesort needs additional memory for merging.

Disadvantages of Quicksort :

1. The worst-case complexity is $O(N^2)$.
2. Very slow in the worst case.
3. In the worst case, could cause a stack overflow.

Q 85. Write down the algorithm of quick sort. (PTU, May 2013 ; Dec. 2010, 2008)

Ans. Algorithm of quick sort :

QuickSort(list, first, last)

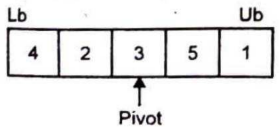
Where list = Represents the list of elements
first = Represents the position of the first element in the list.
last = Represents the position of the last element in the list.

Step 1. Initialize
low = first
high = last
pivot = list[(low + high)/2]
Step 2. Repeat through step 7 while(low < = high).
Step 3. while (list[low] < pivot) repeat step 4.
Step 4. low = low + 1.
Step 5. while(list [high] > pivot) repeat step 6.
Step 6. high = high – 1
Step 7. if(low <= high)
temp = list[low]
list[low] = list[high]
list[high] = temp
low = low + 1
high = high – 1
Step 8. if(first < high)
Call QuickSort(list, first, high)
Step 9. if(low < last)
call QuickSort (list, low, last)
Step 10. Exit

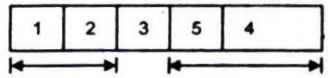
The Quick sort algorithm uses the O (N log 2N) comparisons on average case.

Q 86. Give an appropriate example of quicksort.

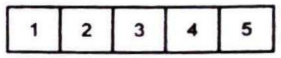
Ans.



The pivot selected is 3. Indices are run starting at both ends of the array. One index starts on the left and selects an element that is larger than the pivot, while another index starts on the right and selects an element that is smaller than the pivot. In this case, number 4 and 1 are selected. These elements are then exchanged so,



This process repeats untill all elements to the left of the pivot <= the pivot, and elements to the right of the pivot are >= the pivot. QuickSort recursively sort the two sub array, so



Q 87. Explain the concept of lower bound on sorting with suitable example.

Ans. Lower bound : A lower bound of a problem is the least time complexity required for any algorithm which can be used to solve this problem.

- Worst case lower bound.
- Average case lower bound.

The lower bound for a problem is not unique. e.g. Ω (1), Ω (n), Ω (n log n) are all lower bounds for sorting.

(Ω (1), Ω (n) are trivial).

A present, if the highest lower bound of a problem is Ω (n log n) and the time complexity of the best algorithm is $O(n^2)$.]

- We may try to find a higher lower bound.
- We may try to find a better algorithm.
- Both of the lower bound and the algorithm may be improved.

If the present lower bound is Ω(n log n) and there is an algorithm with time complexity O(n log n), then the algorithm is optimal.

The worst case lower bound of sorting : 6 permutations for 3 data elements

a_1	a_2	a_3
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

Example : Straight Insertion Sort :

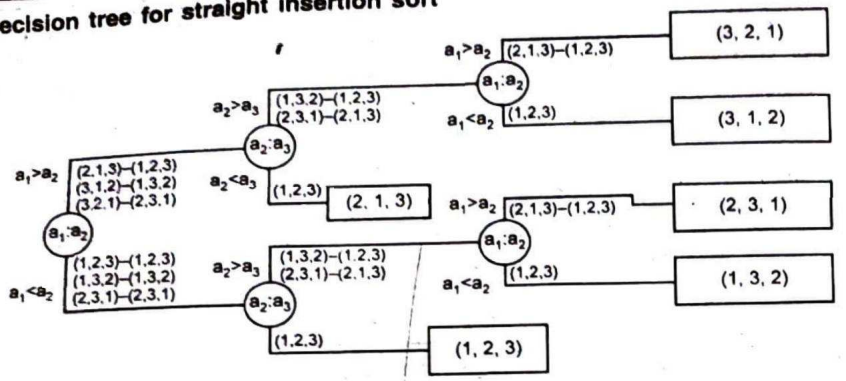
Input data : (2, 3, 1)

1. $a_1 : a_2$
2. $a_2 : a_3$, $a_2 \leftrightarrow a_3$
3. $a_1 : a_2$, $a_1 \leftrightarrow a_2$

Input data : (2, 1, 3)

1. $a_1 : a_2$, $a_1 \leftrightarrow a_2$
2. $a_2 : a_3$

Decision tree for straight insertion sort



Lower bound of sorting : To find the lower bound, we have to find the smallest depth of a binary tree. n! distinct permutations. n! leaf nodes in the binary decision tree.

Balanced tree has the smallest depth :

$$\lceil \log(n!) \rceil = \Omega(n \log n)$$

lower bound for sorting : $\Omega(n \log n)$

Method :

$$\log(n!) = \log(n(n-1) \ldots 1)$$

$$= \log 2 + \log 3 + \ldots + \log n > \int_1^n \log x\,dx$$

$$= \log e \int_1^n \ln x\,dx$$

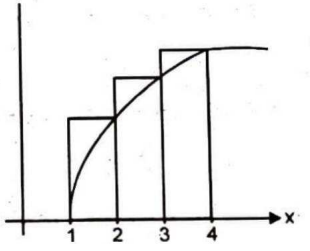
$$= \log e \left[x \ln x - x\right]_1^n$$

$$= \log e\,(n \ln n - n + 1)$$

$$= n \log n - n \log e + 1.44$$

$$\geq n \log n - 1.44\,n$$

$$= \Omega(n \log n)$$



Q 88. Write a short note on selection problem.

Ans. Selection problem :

Input : A set A of n elements or numbers and an integer i, with $1 \leq i \leq n$.

Output : The element $x \in A$ that is larger than exactly $i - 1$ other elements of A.

The selection problem can be solved in O(n log n) time, since we can sort the numbers using heapsort or mergesort and then simply index the ith element in the output array.

Q 89. Define median and order statistic.

Ans. The ith order statistic of a set of n elements is the ith smallest element.

For e.g. : The minimum of a set of elements is the first order statistic (i = 1), and the maximum is the nth order statistic (i = n).

A median, informally, is the "halfway point" of the set.

When n is odd, the median is unique, occuring at $i = (n + 1)/2$. When n is even, there are two median, occuring at $i = n/2$ and $i = n/2 + 1$. Thus, regardless of the parity of n, medians occur at $i = \lfloor (n + 1) / 2 \rfloor$ and $i = \lceil (n + 1)/2 \rceil$.

Q 90. How many comparisons are necessary and sufficient for computing both the minimum and maximum?

Ans. We can easily obtain an upper bound of n – 1 comparisons for finding the minimum of a set of n elements. Examine each element in turn and keep track of the smallest one. The algorithm is optimal, because each element, except the minimum, must be compared to a smaller element at least once.

```
Minimum(A)
1. min ← A[1]
2. for i ← 2 to length[A]
3. do if min > A[i]
4. then min ← A[i]
5. return min
```

Here minimum(A) has worst-case optimal number of comparisons. Well, to compute the minimum n – 1 comparisons are necessary and sufficient. The same is the true for the maximum. So, the number should be 2n – 2 for computing both.

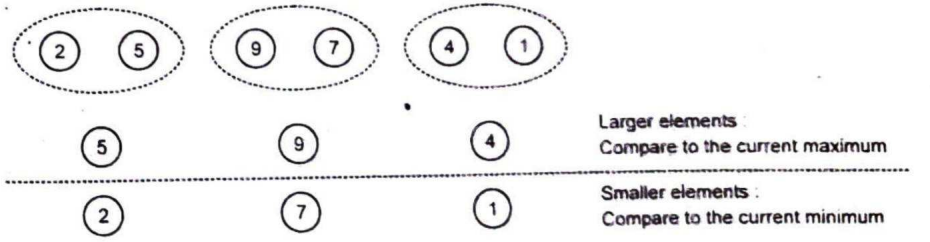
Q 91. Write a note on simultaneous minimum and maximum.

Ans. Simultaneous minimum and maximum : Some applications need to determine both the maximum and minimum of a set of elements. For example : graphics program trying to fit a set of points on to a rectangular display. Independent determination of maximum and minimum requires 2n – 2 comparisons. In fact, at most $3\lfloor n/2 \rfloor$ comparisons are needed.

We maintain the minimum and maximum of elements seen so far. We process elements in pairs. Then we compare with each other, and then compare the larger element to the maximum so far, and compare the smaller element to the minimum so far.

This leads to only 3 comparisons for every 2 elements. If we compare the elements of a pair to each other, the larger can't be the minimum and the smaller can't be the maximum. So we just need to compare the larger to the current maximum and the smaller to the current minimum. It costs 3 comparisons for every 2 elements.

The previous method costs 2 comparisons for each element.



Setting up the initial values for the min and max depends on whether n is odd or even. If n is even, compare the first two elements and assign the larger to max and the smaller to min. If n is odd, set both min and max to the first element.

If n is even, number of comparisons = $\frac{3(n-2)}{2}+1$ (for the initial comparison) = $\frac{3n}{2}-2$

$< 3\lfloor n/2 \rfloor$.

If n is odd, number of comparisons = $\frac{3(n-1)}{2} = 3\lfloor n/2 \rfloor$

Thus total number of comparison is $\leq 3\lfloor n/2 \rfloor$

Q 92. Explain selection problem in expected linear time.

Ans. Selection in expected linear time : Modeled after randomized quicksort and exploits the abilities of Randomized-Partition (RP). Randomized-partition returns the index K in the sorted order of a randomly chosen element (pivot). If the order statistic we are interested in, i, equals k, then we are done. Else, reduce the problem size using its other ability. RP rearranges the other elements around the random pivot. If i < K, selection can be narrowed down to A[1 .... K – 1]. Else, select the (i – K)th element from A[K + 1 ..... n].

(Assuming RP operates on A[1 ..... n] For A[p ..... r], change K appropriately.

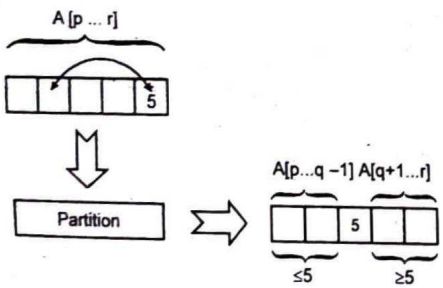
Randomized Quicksort

Quicksort(A,p,r)

```
if   p < r then
     q : = Rnd-Partition(A, p, r) ;
     Quicksort(A, p, q – 1) ;
     Quicksort(A, q + 1, r)
fi
```

Rnd-partition (A, p, r)

```
i : = Random (p, r) ;
A[r] ↔ A[i] ;
x, i : = A[r], p – 1 ;
for j : = p to r – 1 do
    if A[j] ≤ x then
       i : = i + 1 ;
       A[i] ↔ A[j]
    fi
od ;
 A[i + 1] ↔ A[r] ;
   return i + 1
```

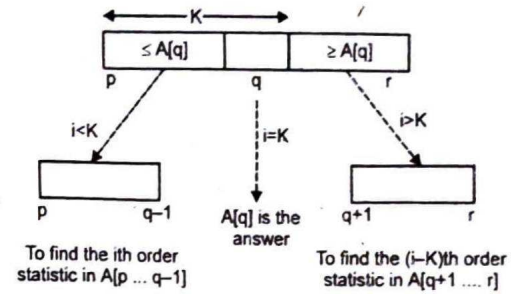


Randomized-select

Randomized-select(A, p, r, i) // select ith order statistic

1. if p = r
2. then return A[p]
3. q ← Randomized-Partition (A, p, r)
4. K ← q – p + 1
5. if i = K

6. then return A[q]
7. elseif i < k
8. then return Randomized-Select (A, p, q – 1, i)
9. else return Randomized-Select (A, q + 1, r, i – K)



Algorithm analysis :

The worst case : Always recurse on a subarray that is only 1 element smaller than the previous subarray.

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

The bese case : Always recurse on a subarray that has half of the elements smaller than the previous subarray.

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

The average case : We will show that T(n) = O(n).

For $1 \leq K \leq n$, the probability that the subarray A[p .. q] has K elements is 1/n. To obtain an upper bound, we assume that T(n) is monotonically increasing and that the ith smallest element is always in the larger subarray. So we have

$$T(n) \leq \frac{1}{n}\sum_{K=1}^{n}\left(T(\max)(K-1, n-K)+O(n)\right)$$

$$T(n) \leq \frac{1}{n}\sum_{K=1}^{n}\left(T(\max)(K-1, n-K)+O(n)\right)$$

$$= \frac{1}{n}\sum_{K=1}^{n}\left(T(\max)(K-1, n-K)\right)+O(n)$$

$$\leq \frac{2}{n}\sum_{k=\lfloor n/2 \rfloor}^{n-1} T(K)+O(n)$$

$\because$ max (K – 1, n – K) = $\begin{cases} K-1 \text{ if } K > \lceil n/2 \rceil \\ n-K \text{ if } K \le \lceil n/2 \rceil \end{cases}$

K	1	2	...	$\lceil n/2 \rceil$	$\lceil n/2 \rceil + 1$	...	n–1	n
max (K–1,n–k)	n–1	n–1	...	n–$\lceil n/2 \rceil$	$\lceil n/2 \rceil$	...	n–2	n–1

if n is even each term from T ($\lceil n/2 \rceil$) to T(n – 1) appears exactly twice.

If n is odd, each term from T ($\lceil n/2 \rceil$) to T(n – 1) appears exactly twice and T($\lfloor n/2 \rfloor$) appears once.

Because $K = \lceil n/2 \rceil$, $K - 1 = n - K = \lfloor n/2 \rfloor$.

Solve this recurrence by substitution : Let us assume T(n) ≤ cn for sufficiently large c. The function described by the O(n) term is bounded by an for all n > 0.

Then, we have

$$T(n) \le \frac{2}{n}\sum_{K=\lceil n/2 \rceil}^{n-1} T(K) + 0(n) \le \frac{2}{n}\sum_{K=\lfloor n/2 \rfloor}^{n-1} cK + an$$

$$= \frac{2c}{n}\left(\sum_{K=1}^{n-1} K - \sum_{K=1}^{\lceil n/2 \rceil - 1} K\right) + an = \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2}\right) + an$$

$$\le \frac{2c}{n}\left[\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2}\right] + an$$

$$= \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2} - 2\right) + an = c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}\right) + an$$

$$\le \frac{3cn}{4} + \frac{c}{2} + an = cn - \left(\frac{cn}{4} - \frac{c}{2} - an\right)$$

Then, if we assume that T(n) = 0(1) for n < 2c/(c – 4a),

We have T(n) = 0(n).

Q 93. Explain selection problem in worst-case linear time.

Ans. Section in worst-case linear time : Select the ith smallest element of S = {a_1, a_2, ..., a_n}.

Use so called **prune and search** technique :

Let x∈S, and partition S into three subsets

$S_1 = \{a_j \mid a_j < x\}$

$S_2 = \{a_j \mid a_j = x\}$

$S_3 = \{a_j \mid a_j > x\}$

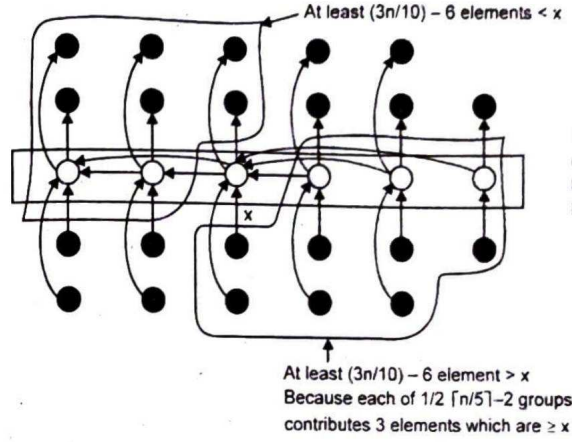
if $|S_1| > i$, search ith smallest element in S_1 recursively, (prune S_2 and S_3 away).

Else if $|S_1| + |S_2| > i$, then return x (the ith smallest element).

Else search $(i - (|S_1| + |S_2|))$ th in S_3 recursively, (prune S_1 and S_2 away).

Now how to select x such that S_1 and S_3 are nearly equal.

The way to select x



Select ith element in n elements :

1. Divide n elements into $\lceil n/5 \rceil$ groups of 5 elements.
2. Find the median of each group.
3. Use SELECT recursively to find the median x of the above $\lceil n/5 \rceil$ medians.
4. Partition n elements around x into S1, S2 and S3.
5. If $|S_1| > i$, search ith smallest element in S1 recursively.
   Else if $|S_1| + |S_2| > i$, then return x (the ith smallest element).
   Else search $(i - (|S_1| + |S_2|))$ th in S_3 recursively.

Analysis of Select :

Step 1, 2, 4 take O(n), step 3 takes T($\lceil n/5 \rceil$) .

Let us see step 5 :

At least half of medians in step 2 are ≥ x, thus at least $1/2\lceil n/5 \rceil - 2$ groups contribute 3 elements which are ≥ x. i.e., $3(\lceil 1/2 \lceil n/5 \rceil \rceil - 2) \ge (3n/10) - 6$.

Similarly the number of elements ≤ x is also at least (3n/10) – 6. Thus, $|S_1|$ is at most (7n/10) + 6, similarly for |S3|. Thus SELECT in step 5 is called recursively on at most (7n/10) + 6 elements.

Recurrence is :

$$T(n) = \begin{cases} 0(1) & \text{if } n < \text{some value (i.e. 140)} \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + 0(n) & \text{if } n \ge \text{the value (i.e., 140)} \end{cases}$$

Solve recurrence by substitution

Suppose $T(n) \le cn$, for some c.

$$T(n) \le c\lceil n/5 \rceil + c(7n/10 + 6) + an$$
$$\le cn/5 + c + 7/10\, cn + 6c + an$$
$$= \frac{9}{10} cn + an + 7c$$
$$= cn + (-cn/10 + an + 7c)$$

Which is at most cn if $-cn/10 + an + 7c < 0$

i.e., $c \ge 10a\,(n/(n-70))$ when $n > 70$.

So select $n = 140$, and then $c \ge 20a$.

Note that n may not be 140, any integer > 70 is OK.

Q 94. The order of complexity of binary search (successful case) in best case is ..... in average case is ..... and in worst case is ..... (PTU, May 2007)

Ans. The order of complexity of binary search (successful case) in best case is O (1) in average case is O (log n) and in worst case is O (log n).

Q 95. What is stable sorting? (PTU, May 2013, Dec. 2010, 2009, 2008)

Ans. Stable sorting : A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array. **For example,** in the following input the two 4's are indistinguishable.

1, 4a, 3, 4b, 2

And so the output of a stable sorting algorithm must be :

1, 2, 3, 4a, 4b

Some sorting algorithms are stable by nature like insertion sort, merge sort, bubble sort, etc. And some sorting algorithm are not, like heap sort, quick sort etc.

Q 96. What is the time complexity of binary search? Explain.

(PTU, Dec. 2014, 2013)

Ans. Time complexity of binary search : The best case complexity is 0(1) i.e. if the element to search is the middle element. The average and worst case time complexity are O(log n).

Q 97. Write a code for maximum heap. (PTU, Dec. 2005)

Ans. Max Heap : Suppose H is a complete binary tree with n elements. Then H is called a heap or max heap, if the value at N is greater than or equal to the value at any of the children of N.

Q 98. Consider a set of elements {12, 34, 56, 73, 24, 11, 34, 56, 78, 91, 34, 91, 45}. Sketch the heapsort algorithm and use it to sort this set. Obtain a derivation for the time complexity of heapsort, both the worst case and average case behaviour.

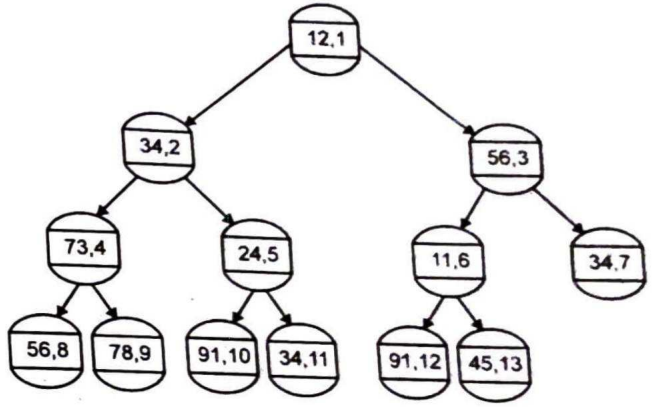
(PTU, Dec. 2011)

Ans. Consider a set of elements {12, 34, 56, 73, 24, 11, 34, 56, 78, 91, 34, 91, 45}. Sketch the heapsort algorithm and use it to sort this set. Obtain a derivation for the time complexity of heapsort, both the worst case and average case behaviour.

Simulation of Heapify

Initial

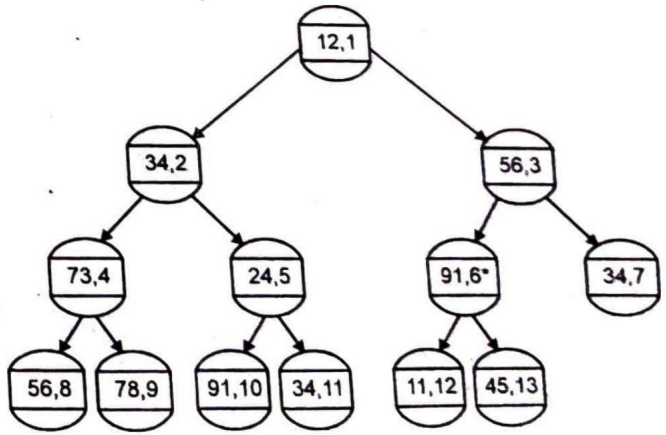
12	34	56	73	24	11	34	56	78	91	34	94	45
1	2	3	4	5	6	7	8	9	10	11	12	13



Number of elements = 13 = n ; l = Floor (n/2) = 6

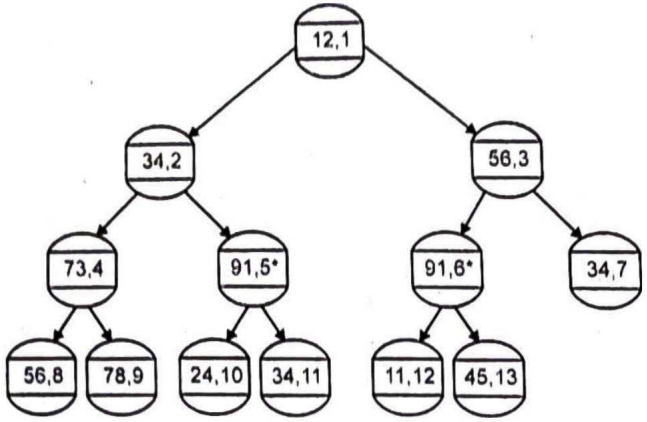
So consider the heap with 6 as root, the left subtree is a one element heap, the right subtree is a one element heap, and the root may be violating the heap property. So 11 comes down and 91 becomes the 6th node.

12	34	56	73	24	91	34	56	78	91	34	11	45
1	2	3	4	5	6	7	8	9	10	11	12	13



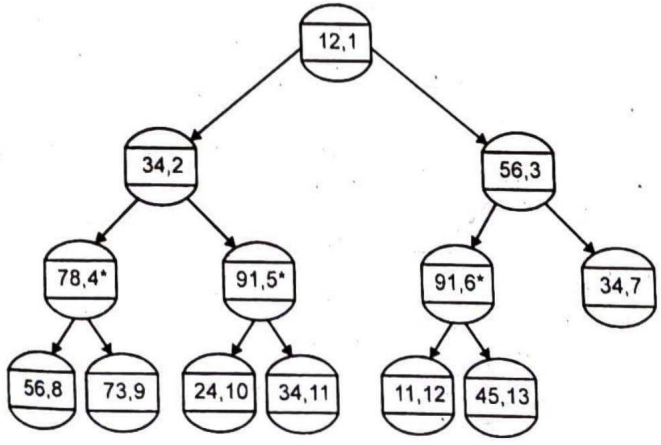
Now let I = 5. We have the heap with 5 as root and left subtree is a one element heap and the right subtree is a one element heap.The element at 5 is violating the heap property so let 91 come up to position 5, and 24 go down the position 10.

12	34	56	73	91	91	34	56	78	24	34	11	45
1	2	3	4	5	6	7	8	9	10	11	12	13



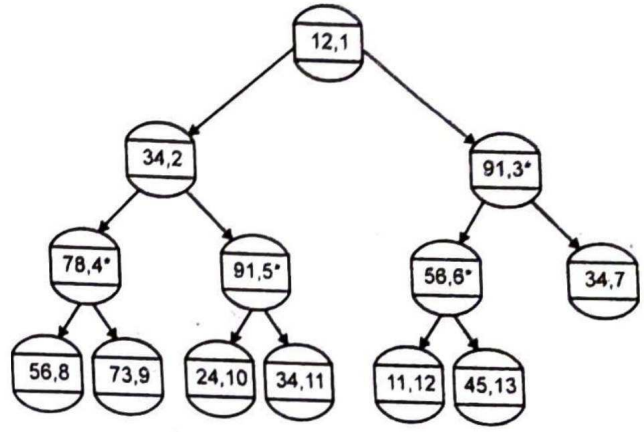
Now let I = 4. We have the heap with 4 as root and left sub tree is a one element heap and the right sub tree is a one element heap. The element at 4 is violating the heap property, so let 78 come up to position 4, and 73 go down to position 9.

12	34	56	78	91	91	34	56	73	24	34	11	45
1	2	3	4	5	6	7	8	9	10	11	12	13



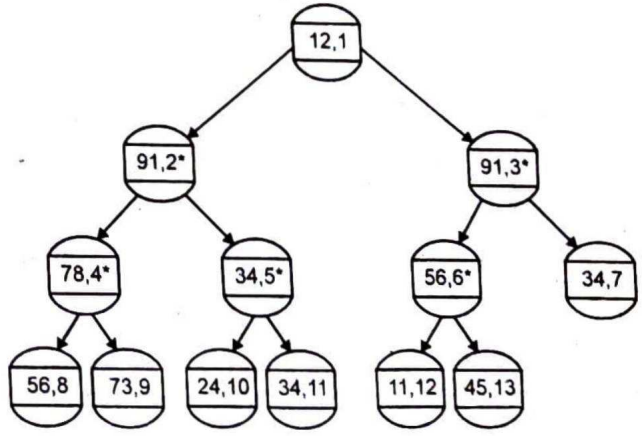
Now let I = 3. We have the heap with 3 as root and left sub tree is a heap and the right sub tree is a one element heap. The element at 3 is violating the heap property, so let 91 come up to position 3, 56 go down to position 6.

12	34	91	78	91	56	34	56	73	24	34	11	45
1	2	3	4	5	6	7	8	9	10	11	12	13



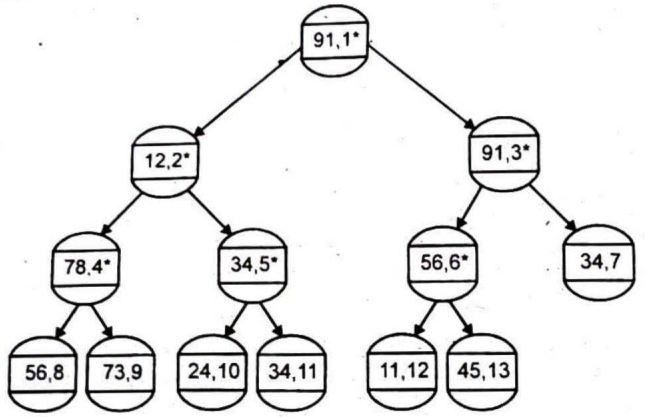
Now let I = 2. We have the heap with 2 as root and left sub tree is a heap and the right sub tree is a heap. The element at 2 is violating the heap property, so let 91 come up to position 2, and 34 go down to position 5.

12	91	91	78	34	56	34	56	73	24	34	11	45
1	2	3	4	5	6	7	8	9	10	11	12	13

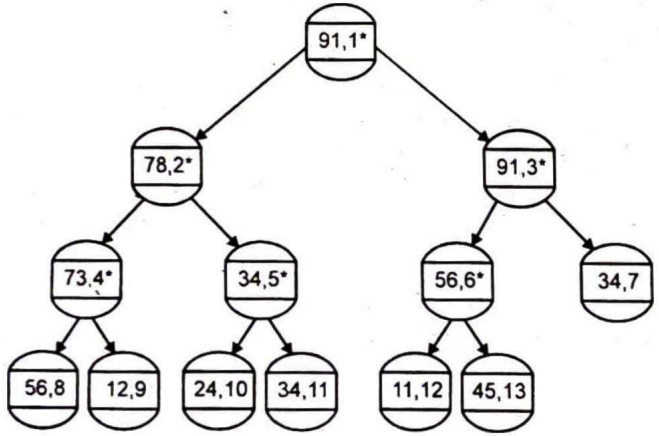


Now let I = 1. We have the heap with 1 as root and left subtree is a heap and the right subtree is a heap. The element at 1 is violating the heap property, so let 91 come up to position 1, and 12 go down to position 2.

91	12	91	78	34	56	34	56	73	24	34	11	45
1	2	3	4	5	6	7	8	9	10	11	12	13



Now 12 is compared with its two children, 73 moves up and 12 moves down.

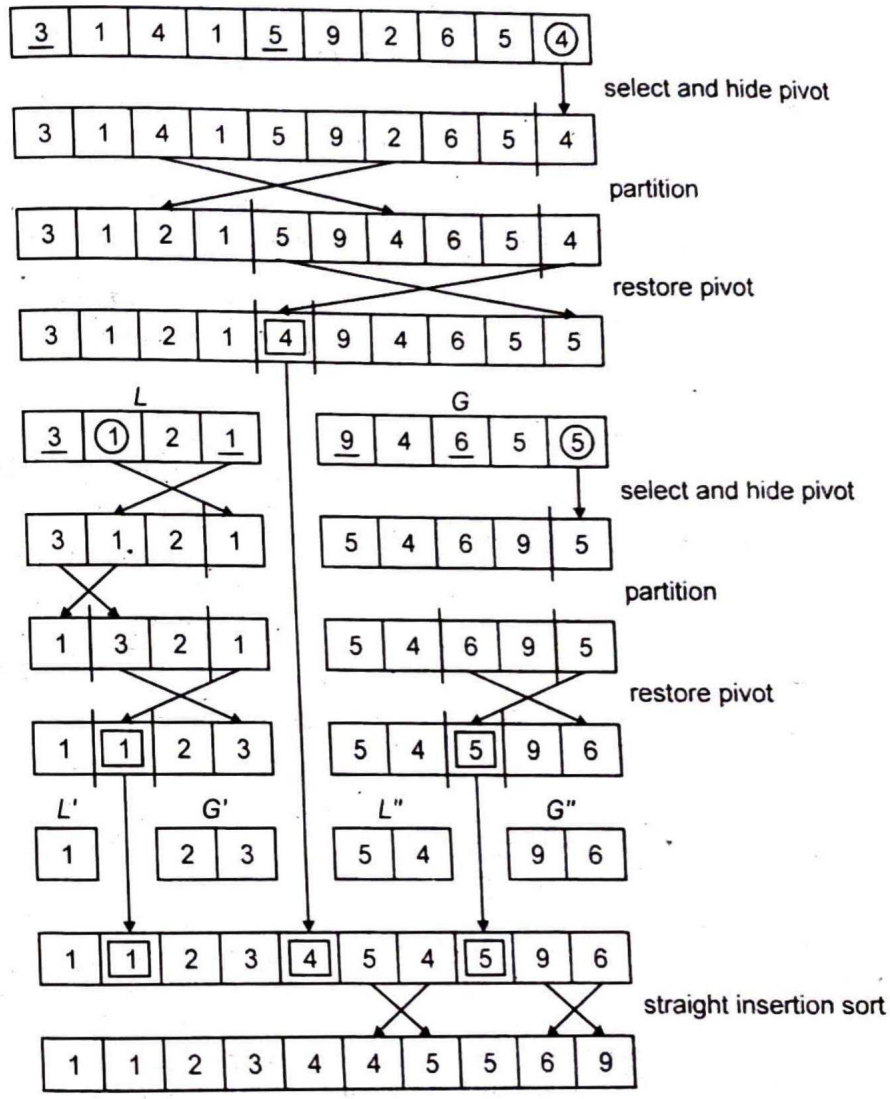


Q 99. What is binary searching? **(PTU, Dec. 2004)**

Ans. A **binary search** is a technique for quickly locating an item in a sequential list. A sequential search is a procedure for searching a table that consists of starting at some table position (usually the beginning) and comparing the file-record key in hand with each table-record key, one at a time, until either a match is found or all sequential positions have been searched.

Q 100. Apply the quicksort technique on the following list :
L = <3, 1, 4, 1, 5, 9, 2, 6, 5, 4> **(PTU, Dec. 2004)**

Ans. Quick sort technique for L = <3, 1, 4, 1, 5, 9, 2, 6, 5, 4> is as follow :



Q 101. What is the worst case running time of quick sort? **(PTU, May 2008)**

Ans. In the worst case, recursion may be n levels deep (for an array of size n)

- But the partitioning work done at each level is still n
- $O(n)^* O(n) = O(n^2)$

- So worst case for Quicksort is O (n^2)
- When does this happen?
- There are many arrangements that could make this happen
- Here are two common cases :
- When the array is already sorted
- When the array is inversely sorted (sorted in the opposite order).

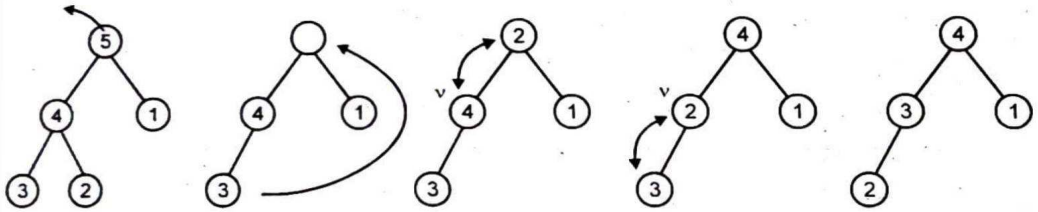
Q 102. What is the order of bubble sort? **(PTU, May 2008)**

Ans. Bubble Sort : A sorting technique that is typically used for sequencing small lists. It starts by comparing the first item to the second, the second to the third and so on until it finds one item out of order. It then swaps the two items and starts over. The sort may alternate from the top of the list to the bottom and then from the bottom to the top. The name comes from the notion that items are raised or "bubbled up" to the top.

Q 103. Write a Heapsort algorithm and analyse the same. **(PTU, Dec. 2004)**

Ans. Heapsort : The data structure of the heapsort algorithm is a heap. The data sequence to be sorted is stored as the labels of the binary tree. As shown later, in the implementation no pointer structures are necessary to represent the tree, since an almosts complete binary tree can be efficiently stored in an array.

Heapsort Algorithm : The following description of heapsort refers to fig. 2 (a) – (e).



(a) (b) (c) (d) (e)

Retrieving the maximum element and restoring the heap

If the sequence to be sorted in arranged as a heap, the greatest element of the sequence can be retrieved immediately from the root (a). In order to get the next-greatest element, the rest of the elements have to be rearranged as a heap.

The rearrangement is done in the following way : Let b be a leaf of maximum depth. Write the label of b to the root and delete leaf b (b). Now the tree is a semi-heap, since the root possibly has lost its heap property.

Making a heap from a semi-heap is simple : Do nothing if the root already has the heap property, otherwise exchange its label with the maximum label of its direct descendants (c). Let this descendant be v, i.e. make a heap from the semi-heap rooted at v (d). This process stops when a vertex is reached that has the heap property (e). Eventually this is the case at a leaf.

Q 104. Explain in detail quick sorting method. Provide a complete analysis of quick sort. **(PTU, May 2012 ; Dec. 2005)**

Ans. Quick Sort : This is the most widely used internal sorting algorithm. It is based on divide-and-conquer type i.e. Divide the problem into sub-problems, until solved sub problems are found.

Algorithm :

This algorithm sorts an array A with N elements

1. [Initialize] TOP : = NULL
2. If N > 1, then TOP : = TOP + 1, LOWER [1] : = 1, UPPER [1] : = N
3. Repeat steps 4 to 7 while TOP ≠ NULL
4. Set BEG : = LOWER [TOP], END : = UPPER [TOP], TOP := TOP – 1
5. Call QUICK (A, N, BEG, END, LOC)
6. If BEG < LOC – 1 then
   TOP : = TOP + 1, LOWER [TOP] : = BEG
   UPPER [TOP] = LOC – 1
   End If
7. If LOC + 1 < END then
   TOP : = TOP + 1, LOWER [TOP] := LOC + 1
   UPPER [TOP] : = END
   End If
8. Exit

The Quick sort algorithm uses the O (N $\log_2$ N) comparisons on average.

Q 105. Write a recursive algorithm for binary search tree and complexity. **(PTU, May 2009)**

Ans.
```
Binary search (A [0...N – 1], value, low, high) {
if (high < low)
return –1 // not found
mid = low + (high – low) / 2
if (A [mid] > value)
return BinarySearch (A, value, low, mid – 1)
else if (A[mid] < value)
return BinarySearch (A, value, mid + 1, high)
else
return mid // found
}
```

It is invoked with initial low and high values of 0 and N – 1.

Q 106. Write Heapify algorithm. **(PTU, Dec. 2006)**

Ans. The bottom up insertion algorithm gives a good way to build a heap, but "Rober Floyd" found a better way, using a merge procedure called heapify.

Given, two heaps and a fresh element, they can be merged into one by making the new one the root and trickling down.

```
Build-heap (A)
n = |A|
for i = ⌈n/2⌉ to 1 do
Heapify (A, i)
Heapify (A, i)
left = 2i
right = 2i + 1
if (left ≤ n) and (A [left] > A [i]) then
max = left
else
max = i
if (right ≤ n) and (A [right] > A [max]) then
max = right
if (max = i) then
swap (A. [z], A [max])
Heapify (A, max)
```

Q 107. What is the time complexity of merge sort? **(PTU, Dec. 2007)**

OR

Write the worst case and best case running time of merge sort.

(PTU, Dec. 2011)

Ans. The time complexity of merge sort is always O (n log n) in all the cases such as best case or worst case.

Q 108. Name three conditions under which sequential search of a list is preferable to binary search. **(PTU, May 2009 ; Dec. 2008)**

Ans. 1. In linear search there is no need that list must be sorted.

2. For binary search one must have direct access to the middle element in any sub list.

3. In binary search keeping data in a sorted array is normally very expensive when there are many insertion and deletions, operation are applied.

Q 109. What is the time complexity of selection sort? **(PTU, May 2010)**

Ans. Selection sort has no end conditions built in, so it will always compare every element with every other element. This gives it a best-worst-and average-case complexity of $O(n^2)$.

Q 110. Analyze the bubble sort algorithm. Argue on its best case, average case and worst case time complexity. **(PTU, Dec. 2005)**

Ans. Bubble Sort : In this sorting algorithm, multiple swapping take place in one iteration. Smaller elements move or 'bubble' up to the top of the list. In this method, we compare the adjacent members of the list to be sorted, if the item on top is greater than the item immediately below it, ithey are swapped.

Algorithm : BUBBLE (DATA, N)

Here DATA is an array with N elements. This algorithm sorts the elements in DATA.

1. Repeat Steps 2 and 3 for K = 1 to N – 1

2. Set PTR : = 1 [Initialize pass pointer PTR]
3. Repeat while PTR < = N – K : [Execute Pass]
   (a) If DATA [PTR] > DATA [PTR + 1], then
   Interchange DATA [PTR] and DATA [PTR + 1]
   End if
   (b) Set PTR : = PTR + 1
   [End of inner loop]
   [End of step 1 outer loop]
4. Exit.

The total numbers of comparissons in Bubble sort are :

$$= (N - 1) + (N - 2) .... + 2 + 1$$

$$= (N - 1) * \frac{N}{2} = O(N_2)$$

The time required to execute the bubble sort algorithm is proportional to n_2, where n is the number of input items. The Bubble sort algorithm uses the $O(n^2)$ comparisons on average.

The **worst case** is that you will have the smallest value in the last space in the array. This means that is will move exactly once each pass towards the first space in the array. It wil take n –1 passes to do this, doing n comparisons on each pass : $O(n^2)$

The **best case** is that the data comes to us already sorted. Assuming that you have a smart implementation (which you should, because it's easy) which stops itself once a pas makes no changes, then we only need to do n comparisons over a single pass : O (n).

Q 111. Give the recurrence relation for the time complexity of merge sort algorithm. **(PTU, May 2015 ; Dec. 2013)**

Ans. Merge sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

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

Q 112. Use the master method to show that the solution to the binary-search recurrence relation $T(n) = T(\frac{n}{2}) + \theta(1)$ is $T(n) = \theta(\lg n)$. **(PTU, May 2014)**

Ans. We can use the Master theorem case 2 because from a=1 and b=2. We have $n^{\log_b a} = n^0 = 1$, so for k=0 $f(n) = \theta(1) = (n^{\log_b a} \lg^k n)$.

This gives that

$$T(n) = \theta(n^{\log_b a} \lg^{k+1} n) = \theta(\lg n)$$

Q 113. List out two drawbacks of binary search algorithm. **(PTU, Dec. 2014)**

Ans. (1) In binary search the elements have to be arranged either in ascending or descending order.

(2) Each time the mid element has to be computed in order to partition the list in two sublists.

Q 114. Sort the following list using merge sort technique **(PTU, Dec. 2005)**

L = <5, 8, 3, 9, 2, 10, 1, 40>

Ans. Pass 1 : After merging each pair of elements following list of sorted pairs as obtain :

5 8 3 9 2 10 1 40

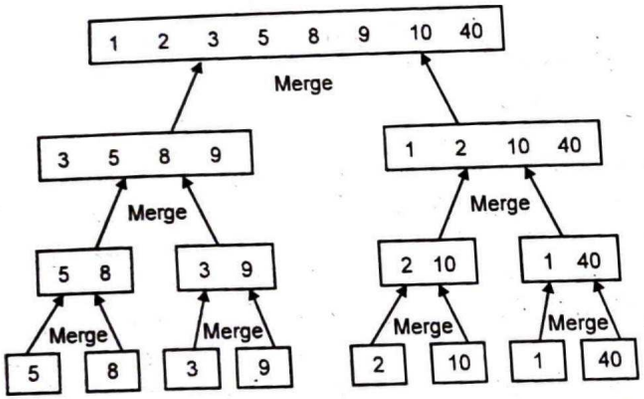
Pass 2 : After merging each pair of elements following sorted quadraplets are obtained

3 5 8 9 1 2 10 40

Pass 3 : Following sorted list is obtained after merging each sorted quadraplets.

1 2 3 5 8 9 10 40

Whole process of sorting



Q 115. Amongst the various sorting techniques as Merge sort, insertion sort and bubble sort, which is best in worst case. Support your arguments with analysis. **(PTU, Dec. 2010 ; May 2013, 2010, 2009)**

Ans. Insertion Sort : Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly-sorted lists, and often is used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. The insertion sort works just like its name suggests – it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures – the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceeding item until it is in place. Shell ort (see below) is a variant of insertion sort that is more efficient for larger lists. This method is much more efficient than the bubble sort, though it has more constraints.

Bubble Sort : Bubble sort is a straightforward and simplistic method of sorting data

that is used in computer science education. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. While simple, this algorithm is highly inefficient and is rarely used except in education. A slightly better variant, cocktail sort, works by inverting the ordering criteria and the pass direction on alternating passes. Its average case and worst case are both O (n^2).

Merge Sort : Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e., 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on ; until at last two lists are merged into the final sorted list. Of the algorithms described here, this is the firs that scales well to very large lists, because its worst-case running time is O (n log n).

Q 116. How binary tree can be used for searching an element? Explain. **(PTU, Dec. 2007)**

Ans. Binary Search : The binary search is the standard method for searching through a sorted array. It is much more efficient than a linear search, where we pass through the array elements in turn until the target is found. It does not require that the elements be in order.

The binary search repeatedly divides the array in two, each time restricting the search to the half that should contain the target element.

In this example, we search for the integer 5 in the 10-element array below :

2	5	6	8	10	12	15	18	20	21

Loop 1-Look at whole array

Low index = 0, high index = 9

Choose element with index (0 + 9) / 2 = 4

2	5	6	8	10	12	15	18	20	21

Compare value (10) to target

10 is greater than 5, so the target must be in the lower half of the array

Set high index = (4 – 1) = 3

Loop 2

Low index = 0, high index = 3

Choose element with index (0 + 3) /2 = 1

2	5	6	8	10	12	15	18	20	21

Compare value (5) to target

5 is equal to target

Target was found, index = 1

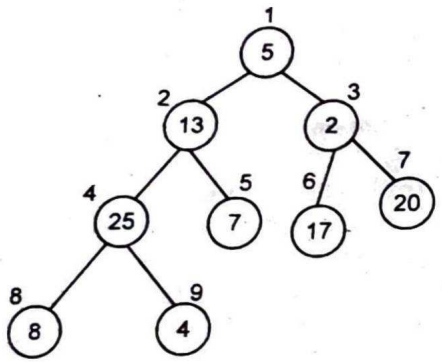
Q 117. Sort the following using Heapsort technique **(PTU, May 2009)**

L = <5, 13, 2, 25, 7, 17, 20, 8, 4>.

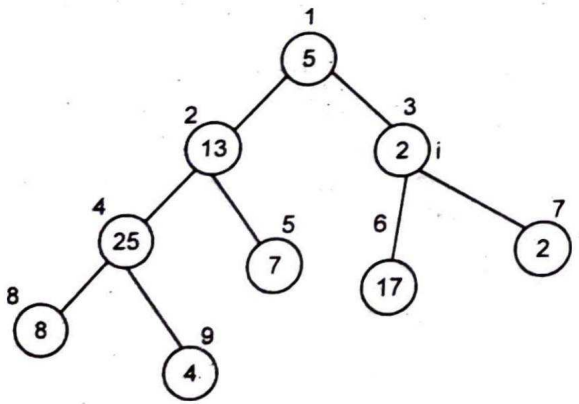
Ans. First we call BUILD-MAX-HEAP heap size [A] = 9

$$\frac{i}{2} = \frac{9}{2} = 4.5$$

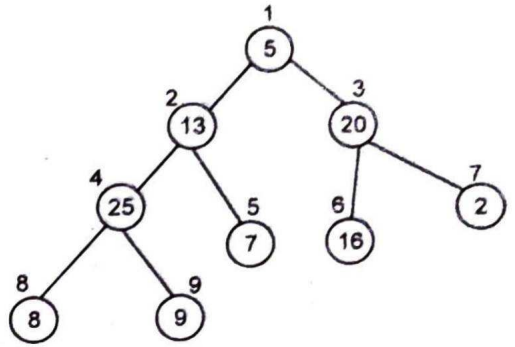
so, $i = 4, 3, 2, 1$



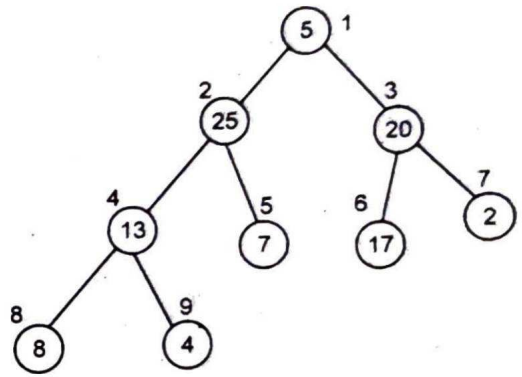
Now we call MAX-HEAPIFY (A, 4)



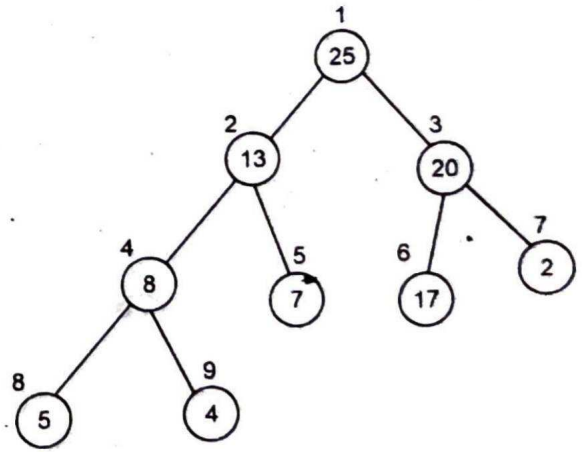
Now we call MAX-HEAPIFY (A, 3)



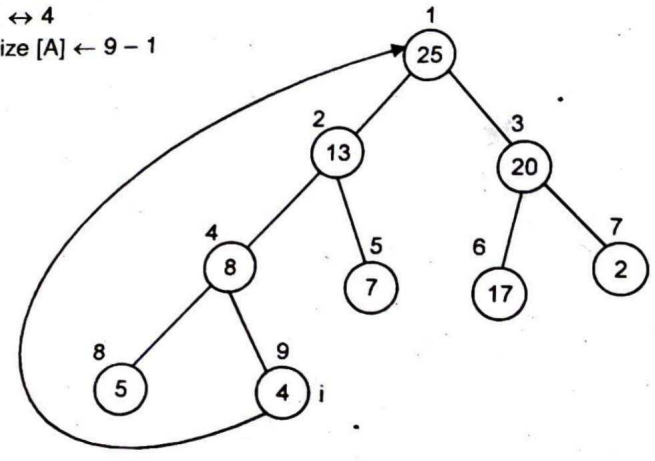
Now call MAX-HEAPIFY (A, 2)



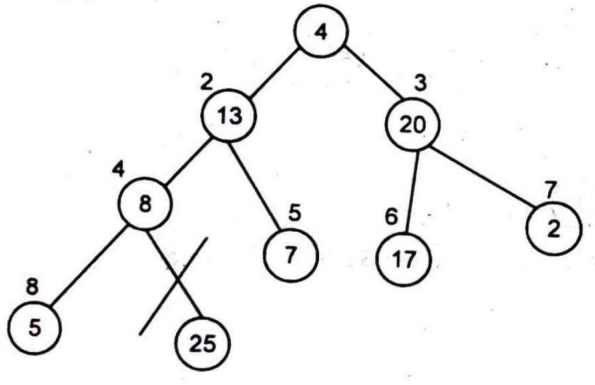
We call MAX-HEAP IFY (A, I), we get BUILD-MAX HEAP



Now for i ← 9 down to 2 when i = 9
do exchange A [1] ↔ A [9]
i.e., 25 ↔ 4
heap size [A] ← 9 – 1

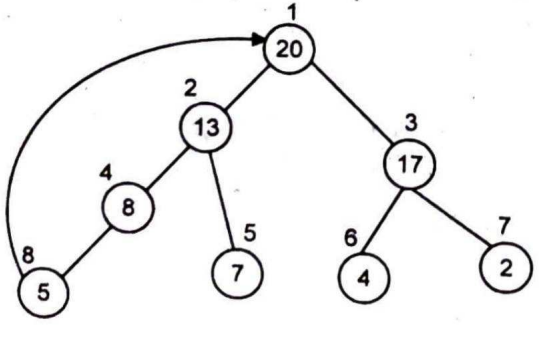


i.e., heap size [A] ← 8

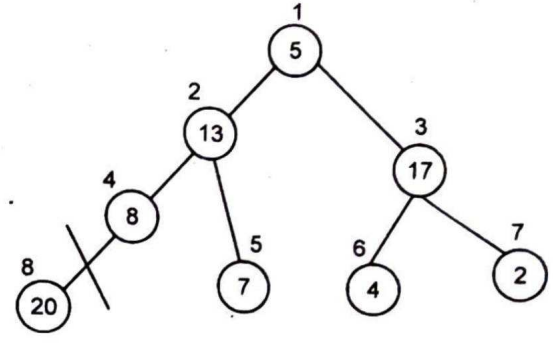


25

Now call MAX-HEAPIFY (A, 1), we get

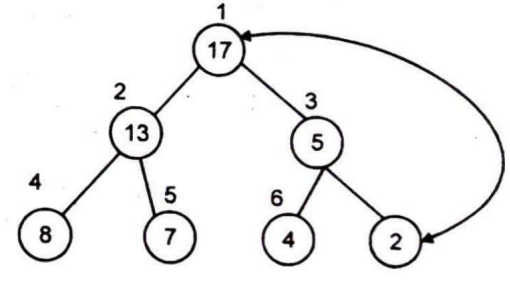


When i = 8, do exchange A [1] ↔ A [8] and heap size
[A] ← 8 – 1
i.e. heap size [A] ← 7

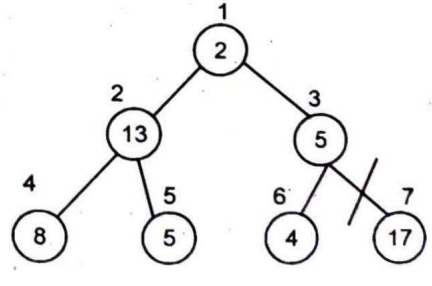


20	25

Call MAX-HEAPIFY (A, 1), we get

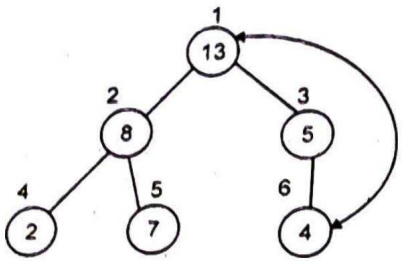


When i = 7, do exchange A [1] ↔ A [7] and heap size
[A] ← 7 – 1
i.e. heap size [A] ↔ 6

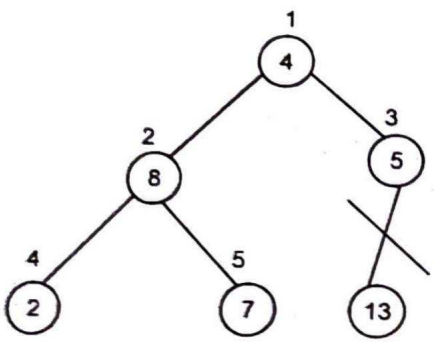


17	20	25

We call MAX.HEAPIFY (A, 1)

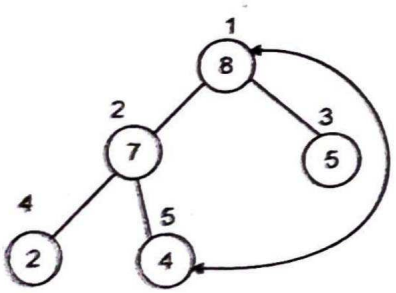


When i = 6, do exchange A [1] ↔ A [6] and heap size [A] ← 6 – 1
i.e. heap size [A] ← 5

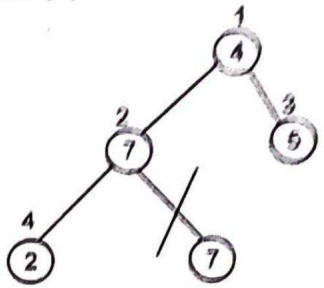


13	17	20	25

Now, we call MAX-HEAPIFY (A, 1)

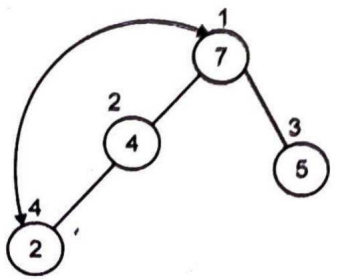


When i = 5, do exchange A [1] ↔ A [5] and heap size [A] ← 5 – 1
i.e. heap size [A] ← 4

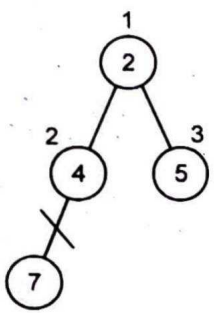


8	13	17	20	25

Again call MAX-HEAPIFY (A, 1)

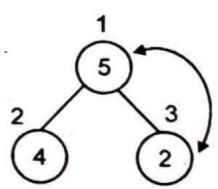


When i = 4, do exchange A [1] ↔ A [4] and heap size [A] ← 3

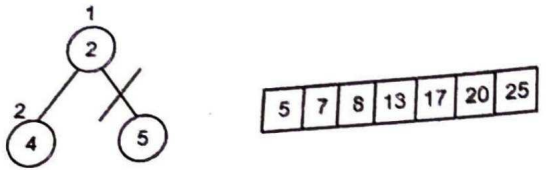


7	8	13	17	20	25

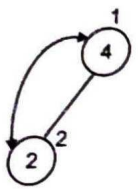
Again, we call MAX-HEAPIFY (A, 1)



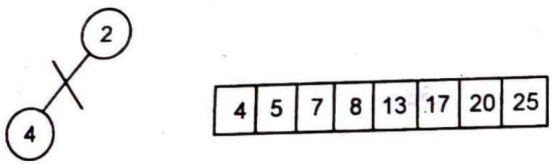
When i = 3, do exchange A [1] ↔ A [3] and heap size [A] ← 2



Again call MAX-HEAPIFY (A, 1)



When i = 2, do exchange A [1] ↔ A [2] and heap size [A] ← 1



Thus, sorted list is

2	4	5	7	8	13	17	20	25

Q 118. Using binary search algorithm, find the number of comparisons required to find key value a the given list.

–15,–6,0,7,9,23,54,82,101,112,125,131,142,151 **(PTU, May 2014)**

Ans.

–15	–6	0	7	9	23	54	82	101	112	125	131	142	151
0	1	2	3	4	5	6	7	8	9	10	11	12	13

We want to search 9, then in first pass

Middle = (0+13)/2 = 6

–15	–6	0	7	9	23	54	82	101	112	125	131	142	151
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Middle

9 is less than the middle element, then process is repleated in lst half.

Middle = (0+5)/2 = 2

–15	–6	0	7	9	23	54	82	101	112	125	131	142	151
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Middle

Now middle = $\frac{3+5}{2} = \frac{8}{2} = 4$

–15	–6	0	7	9	23	54	82	101	112	125	131	142	151
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Middle

Now the position is found that is 4 and value is found i.e. 9.

Q 129. Explain job sequencing with deadlines with a suitable example.

Ans. Job sequencing with deadlines : We are given a set of 'n' jobs. Associated with each job there is a integer deadline di > 0 and a profit pi > 0. For any job i the profit pi is earned if and only if the job is completed by its deadline. To complete a job one has to process the job on a machine for one unit of time. Only one machine is available for processing the jobs. A feasible solution for the problem will be a subset 'j' of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution 'j' is the sine of the profits of the jobs in 'j'. An optimal solution is a feasible solution with maximum value.

The problem involves identification of a subset of jobs which can be completed by its deadline. Therefore the problem suites the subset methodology and can be solved by the greedy method.

Example : Obtain the optimal sequence for the following jobs.

$$(P1, P2, P3, P4) = (\overset{J1}{100}, \overset{J2}{10}, \overset{J2}{15}, \overset{J4}{27})$$

(d1, d2, d3, d4) = 2, 1, 2, 1

n=4

Feasible solution	Processing Sequence	Value
j1, j2		
(1, 2)	(2, 1)	100 + 10 = 110
(1, 3)	(1, 3) or (3, 1)	100 + 15 = 115
(1, 4)	(4, 1)	100 + 27 = 127
(2, 3)	(2, 3)	10 + 15 = 25
(3, 4)	(4, 3)	15 + 27 = 42
(1)	(1)	100
(2)	(2)	10
(3)	(3)	15
(4)	(4)	27

In this example solution '3' is the optimal. In this solution only jobs 1 and 4 are processed and the value is 127. These jobs must be processed in the order j4 followed by j1. The process of job 4 begins at time 0 and ends at time 1. And the processing of job 1 begins at time 1 and ends at time 2. Therefore both the jobs are completed with in their deadlines. The optimization measure for determining the next job to be selected into the solution is according to the profit. The next job to include is that which increases Σp_i the most, subject to the constraint that the resulting "J" is the feasible solution.

Therefore the greedy strategy is to consider the jobs in decreasing order of profits.

Q 130. Suppose we use Dijkstra's greedy, single source shortest path algorithm on an undirected graph. What contraint must we have for the algorithm to work and Why ? **(PTU, Dec. 2019)**

Ans. Dijkstra algorithm will work fine under these constraints.

Eg. If N = 5 and the vertices are 1 (the source), 2, 3, 4 and 5 the list ([2, 3, 4], [2, 3, 4, 5], [2, 3. 4, 5], [3, 4, 5]) means that for step 2 only vertices 2, 3 and 4 can be visited and so forth.

Starting from vertex 1 we can get to 2. (Let's suppose distance d = 2), 3 (d = 7) and y (d = 11) – Current value of distance is [0, 2, 7, 11, N/A] Next, pick the vertex with the shortest distance (vertex 2). We can get from it to 2 again (shouldn't be counted), 3 (d = 3), 4 (d = 4) or 5 (d = 9). We see, that we can get to the vertex 3 with distance 2 + 3 = 5 < 7, which is shorter than 7, so update the value. The same is for the vertex 4 (2 + 4 = 6 < 11) – Current values are [0, 2, 5, 6, 9].

Mark all the vertices we visited and follow the algorithm until all the vertices are selected.

Q 131. Suppose you were to drive from Delhi to Mumbai. Your gas tank, when full, holds enough gas to travel m miles and you have a map that gives distances between gas stations along the route. Let d1 < d2< ..... < dn be the locations of all the gas stations along the route where di is the distance from Delhi to gas station. You can assume that the distance between neighbouring gas stations is at most m miles. Your goal is to make as few gas stops as possible along the way. Give the most efficient algorithm you can find a to determine at which gas stations you should stop and prove that your strategy yields an optimal solution. Be sure to give the time complexity of your algorithm as a function of n. **(PTU, Dec. 2019)**

Ans. The greedy algorithm we use is to go as far as possible before stopping for gas. Let c_i be the city with distance d_i from St. Louis. Here is the pseudo-code.

S = ;
last = 0
for i = 1 to n
if (d_i, last) > m
s = s [f_{ci}g
last = t_i]

Clearly the above is an O(n) algorithm. We now prove it is correct.

Greedy Choice Property : Let S be an optimal solution. Suppose that its sequence of stops is s_1; s_2; : : : ; s_k where s_i is to stop corresponding to distance t_i. Suppose that g is the last stop made by the above greedy algorithm. We now show that there is an optimal solution with last stop at g. If s_1 = g then S is such a solution. Now suppose that $s_1$6 = g. Since the greedy algorithm stops at the latest possible city then it follows that s_1 is before g. We now

argue that S = hg; s2; s3; : : : ;sk i is an optimal solution. First note that |S| = |S|. Second, we argue that S is legal (i.e. you never run out of gas). By definition of the greedy choice you can reach g. Finally, since S is optimal and the distance between g and s_2 is no more than the distance between s_1 and s_2, there is enough gas to get from g to s_2. The rest of S is like S and thus legal.

Optimal Substructure Property : Let P be the original problem with an optimal solution S. Then after stopping at the station g at distance d_i the subproblem P that remains is given by d_{i+1}; : : : : ; d_n (i.e. you start at the current city instead of St. Louis).

Let S be an optimal solution to P. Since, cost (S) = cost (S) + 1, clearly an optimal solution to P include within it an optical solution to P.

Q 132. Give the solution for knapsack with Branch and Bound. The Capacity of knapsack is m = 12. There are 5 Objects with profit (p1, p2, p3, p4, p5) = (10, 15, 6, 8, 4) and weights (w_1, w_2, w_3, w_4, w_5) = (4, 6, 3, 4, 2). **(PTU, Dec. 2019)**

Ans. Step 1 : (To find profit/weight ratio)

p1/w1 = 10/2 = 5
p2/w2 = 5/3 = 1.67
p3/w3 = 15/5 = 3
p4/w4 = 7/7 = 1
p5/w5 = 6/1 = 6
p6/w6 = 18/4 = 4.5
p7/w7 = 3/1 = 3

Setp 2 : (Arrange this profit/weight ratio in non-increasing order as n values) Since the highest profit/weight ratio is 6. This is p5/w5, so 1st value is 5. Second highest profit/weight ratio is 5. That is p1/w1, so 2nd value is 1. Similarly, calculate such n values and arrange them in non-increasing order.

Order = (5, 1, 6, 3, 7, 2, 4)

Step 3 : (To find optimal solution using m = 15 & n = 7)

Consider x5 = 1, profit = 6
Then consider x 1 = 1, profit = 10
So weight uptil now = 1 + 2 = 3
Now x6 = 1, profit = 18
So total profit = 16 + 18 = 34
And weight uptil now = 3 + 4 = 7
Now x3 = 1, profit = 15
So total profit = 34 + 15 = 49
And weight uptil now = 7 + 5 = 12
Now x7 = 1, profit = 3
So total profit = 49 + 3 = 52
And weight uptil now = 12 + 1 = 13
Since m = 15 so we require only 2 units more. Therefore x2 = 2/3
So total profit = 52 + 5 x 2/3 = 52 + 3.33 = 55.3
And weight uptil now = 13 + 3 x 2/3 = 15
Thus, the optimal solution that gives maximum profit is,
(1, 2/3, 1, 0, 1, 1, 1)

Q 133. Write a program for recursive binary search to find the given element within array. For what data binary search is not applicable. **(PTU, Dec. 2019)**

Ans.

```
# include <stdio.h>
int binary search (int arr[ ], int , int r, int x)
{
If (r > = 1)
{
int m = 1 + (r – 1) /2;
If (arr [mid] = = x) return mid;
if (arr [mid] > x) return binary search (arr, 1, mid –1 x);
return binary search (arr, mid + 1, r, x);
}
return – 1;
}
int + main (void)
{
int arr [ ] = {2, 3, 4, 10, 40}
int n = size of (arr)/size of (arr [0]);
int x = 10;
int result = binary search (arr, 0, n–1, x);
(result = = –1) ? printf ("element is not present in array").
}
return 0; printf ("element is present at index % d", result);
```

Binary search is not possible in linked list data structure if the list is not sorted and any andom element in it can not be accessed in constant time.

❑❑❑

Chapter 3

Graph and Tree Algorithms

Contents

Traversal algorithms : Depth First Search (DFS) and Breadth First Search (BFS); Shortest path algorithms, Transitive closure, Minimum Spanning Tree, Topological sorting, Network Flow Algorithm.

POINTS TO REMEMBER

1. When the search necessarily involves the examination of every vertex in the object being searched it is called a traversal.
2. There are two techniques for traversals in graph. These are :
   (i) Breadth first search
   (ii) Depth first search
3. Topological sort is an ordering of the vertices in a directed acyclic graph (DAG), such that : if there is a path from u to v, then u appears after u in the ordering.
4. Dijkstra's algorithm finds the length of an optimal path between two vertices in a graph.
5. A subgraph T of a undirected graph G = (V,E) is a spanning tree of G if it is a tree and contain every vertex of G.
6. A minimum spanning tree is a subgraph of an undirected weighted graph G, such that:
   (i) It is a tree (i.e. It is acyclic)
   (ii) It cover all the vertices V. It contains |V|-1 edges.

QUESTION-ANSWERS

Q 1. Define Traversal.

Ans. When the search necessarily involves the examination of every vertex in the object being searched it is called a traversal.

Q 2. List out the techniques for traversals in graph. **(PTU, Dec. 2018)**

Ans. 1. Breadth first search
2. Depth first search

Q 3. Give a suitable example and explain the breadth first search (BFS). **(PTU, Dec. 2016, 2015 ; May 2019, 2015, 2014)**

Ans. Breadth First Search (BFS) : Breadth-first-search (BFS) is a general technique for traversing a graph. A breadth first search traversal of a graph G.

❑ Visit all the vertices and edges of G.

- ❑ Determines whether G is connected.
- ❑ Compute the connected component of G.
- ❑ Compute a spanning forest of G.

Breadth first search on a graph with n vertices and m edges takes O(n + m) time.

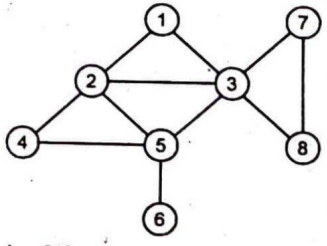
BFS can be further extended to solve other graph problems.

- ❑ Find and report a path with the minimum number of edges between 2 given vertices.
- ❑ Find a simple cycle, if there is one.

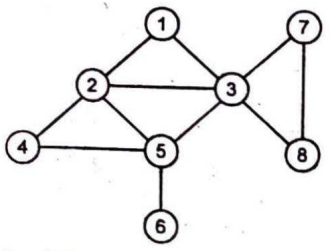
Breadth first search is obtained from **Basic Search** by processing edges using a data structure called a queue. It processes the vertices in the graph in the order of their shortest distance from the vertex s (the start vertex).

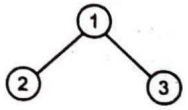
BFS Algorithm : Given (undirected or directed) graph G = (V,E) and nodes s ∈ V.

BFS(s)
Mark all vertices as unvisited
Initialize search tree T to be empty
Mark vertex s as visited
Set Q to be the empty queue
enq(s)
While Q is non-empty do
u = deq(Q)
for each vertex v ∈ Adj(u)
if v is not visited then
add edge(u, v) to T
Mark v as visited and enq(v)

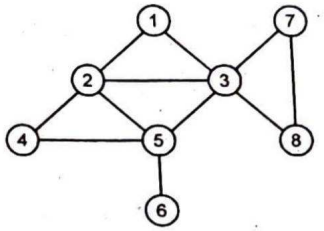


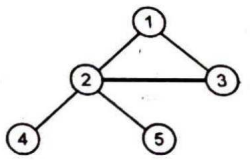
1. [1]



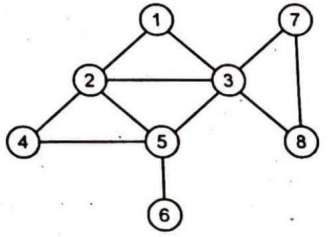


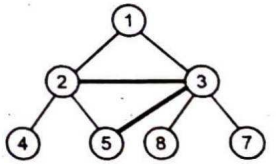
1. [1]
2. [2, 3]



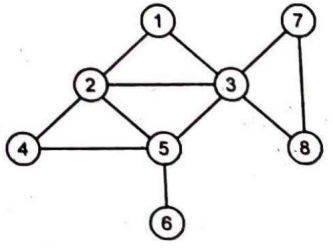


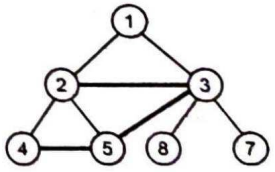
1. [1]
2. [2, 3]
3. [3, 4, 5]



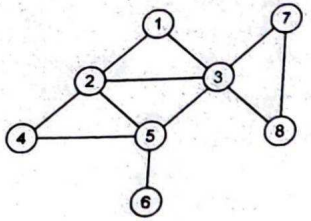
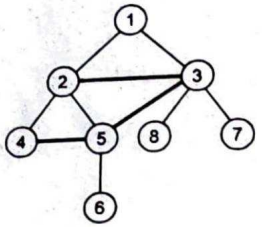


1. [1]
2. [2, 3]
3. [3, 4, 5]
4. [4, 5, 7, 8]

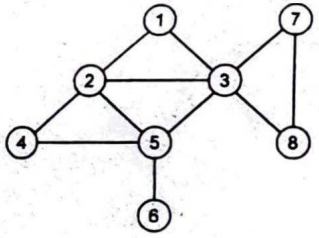
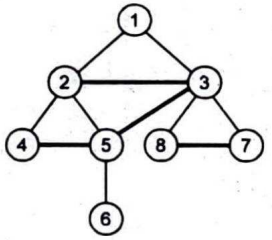




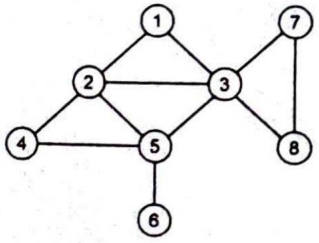
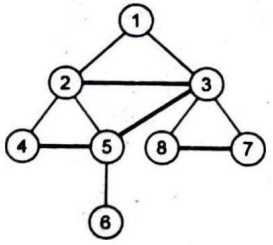
1. [1]
2. [2, 3]
3. [3, 4, 5]
4. [4, 5, 7, 8]
5. [5, 7, 8]

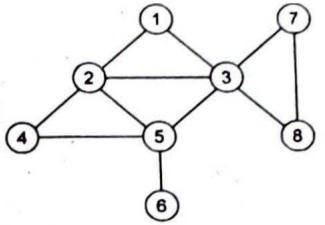
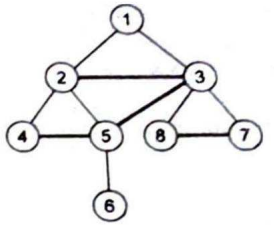
1. [1]
2. [2, 3]
3. [3, 4, 5]
4. [4, 5, 7, 8]
5. [5, 7, 8]
6. [7, 8, 6]

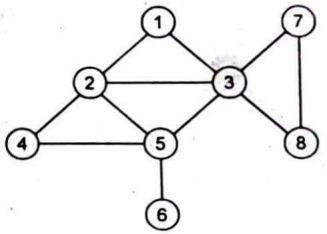
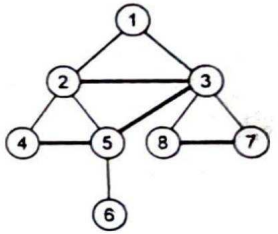
1. [1]
2. [2, 3]
3. [3, 4, 5]
4. [4, 5, 7, 8]
5. [5, 7, 8]
6. [7, 8, 6]
7. [8, 6]

1. [1]
2. [2, 3]
3. [3, 4, 5]
4. [4, 5, 7, 8]
5. [5, 7, 8]
6. [7, 8, 6]
7. [8, 6]
8. [6]

1. [1]
2. [2, 3]
3. [3, 4, 5]
4. [4, 5, 7, 8]
5. [5, 7, 8]
6. [7, 8, 6]
7. [8, 6]
8. [6]
9. [ ]

BFS tree is the set of black edges

Q 4. Write down the applications of breadth first search and depth first search.
(PTU, Dec. 2017 ; May 2019, 2018, 2016)

Ans. BFS : Using the template method pattern, we can specialize the BFS traversal of a graph G to solve the following problems in O(n + m) time.

- Compute the connected components of G.
- Compute a spanning forest of G.
- Find a simple cycle in G, or report that G is a forest.
- Given two vertices of G find a path in G between them with the minimum number of edges or report that no such path exists.

DFS :

- To find a path from a vertex S to a vertex v.
- To find the length of such a path.
- To construct a DFS tree/forest from a graph.
- Topological sort : Using depth-first search to perform topological sort of a directed acyclic graph.
- Strongly connected components : Decomposing a directed graph into a strongly connected components using two depth-first searches.

Q 5. Give a suitable example and explain the depth first search (DFS).
(PTU, May 2017 ; Dec. 2016)

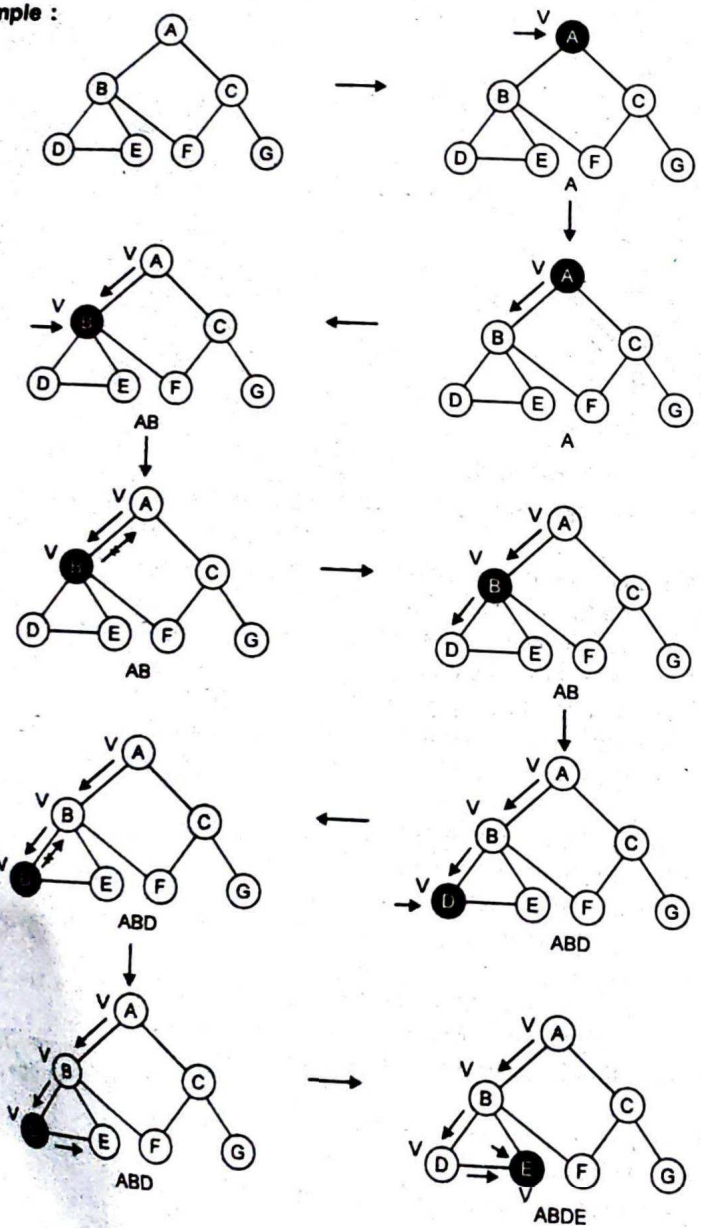
Ans. DFS : Similar to depth-first traversal of a binary tree.

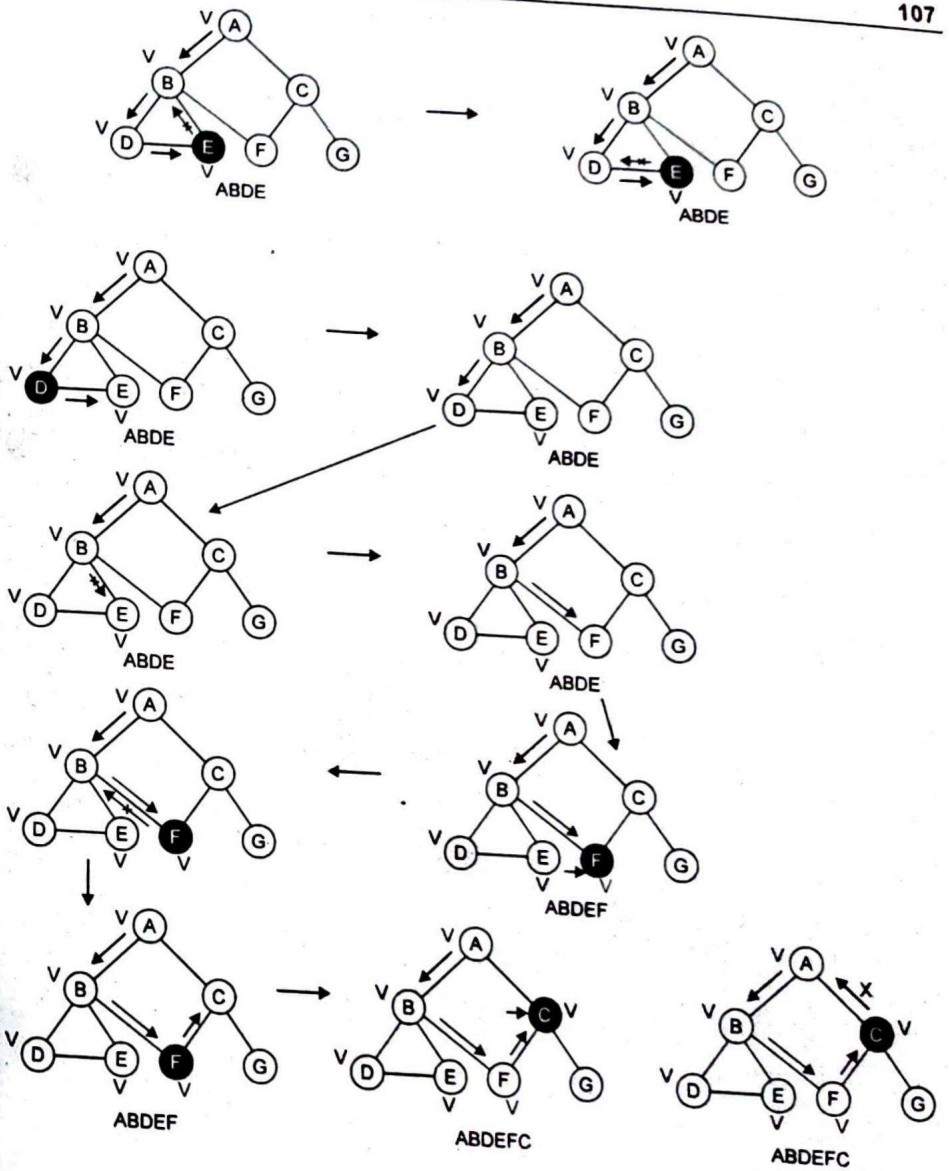
- Choose a starting vertex.
- Do a depth-first search on each adjacent vertex.

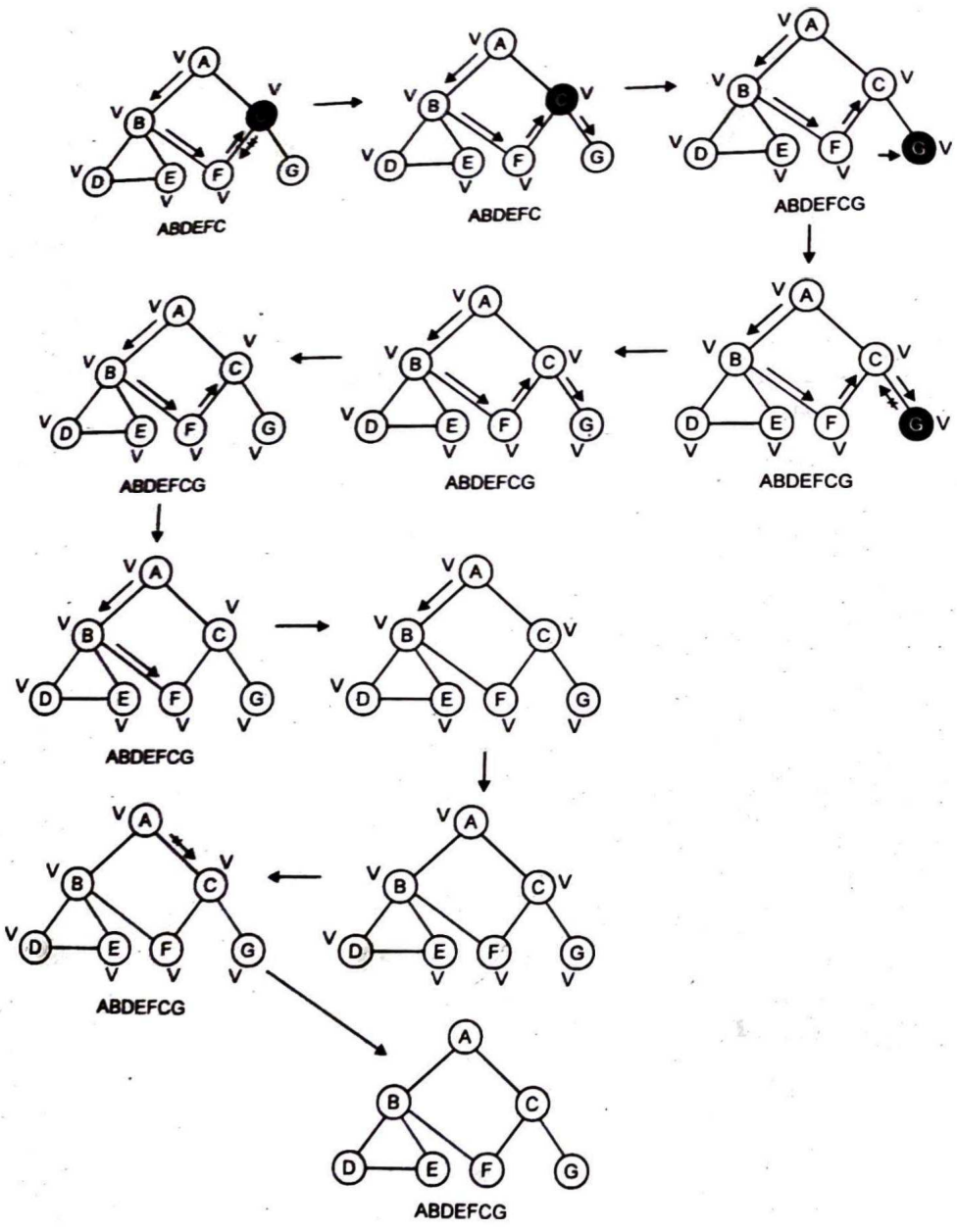
Pseudo-code for depth-first search

DFS : Mark vertex as visited
for each adjacent vertex
if unvisited
do a depth-first search on adjacent vertex.

Example :







Q 6. Explain topological sort with the help of suitable example.
(PTU, May 2019 ; Dec. 2014)

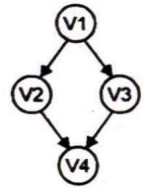
Ans. Topological Sort : An ordering of the vertices in a directed acyclic graph (DAG) such that :

If there is a path from u to v, then v appears after u in the ordering.

Types of graphs :

1. The graphs should be directed, otherwise for any edge (u, v) there would be a path from u to v and also from v to u and hence they cannot be ordered.
2. The graph should be acyclic, otherwise for any two vertices u and v on a cycle u would preceede v and v would preceede u.

The ordering may not be unique



V1, V2, V3, V4 and V1, V3, V2, V4 are legal orderings.

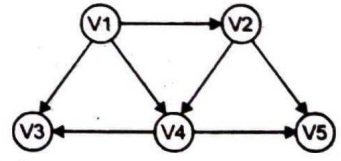
Here, degree of a vertex U is the number of edges (U, V) i.e. outgoing edges.

Indegree of a vertex U is the number of edges (V, U) i.e. incoming edges.

The algorithm for topological sort uses "indegrees" of vertices :

1. Compute the indegrees of all vertices.
2. Find a vertex U with indegree 0 and store it in the ordering. If there is no such vertex then there is a cycle and the vertices cannot be ordered. Stop.
3. Remove U and all its edges (U, V) from the graph.
4. Update the indegrees of the remaining vertices.
5. Repeat step 2 through 4 while there are vertices to be processed.

Example :



1. Firstly, compute the indegrees
   V1 : 0
   V2 : 1
   V3 : 2
   V4 : 2
   V5 : 2
2. Find a vertex with indegree 0 : V1
3. Now remove V1 and update the indegrees :
   Sorted : V1

Remove edges : (V1, V2), (V1, V3) and (V1, V4)

Updated indegrees

V2 : 0

V3 : 1

V4 : 1

V5 : 2

	Indegree					
Sorted →		V1	V1, V2	V1, V2, V4	V1, V2, V4, V3	V1,V2,V4,V3,V5
V1	0					
V2	1	0				
V3	2	1	1	0		
V4	2	1	0			
V5	2	2	1	0	0	

Complexity of this algorithm : $O(|V|^2)$, |V| i.e. the number of vertices.

To find a vertex of indegrees 0 we scan all the vertices i.e., |V| operations.

Thus, we do this for all vertices i.e, $|V|^2$

After the initial scanning to find a vertex of degree 0, we need to scan only those vertices whose updated indegrees have become equal to 0.

1. Store all vertices with indegree 0 in a queue.
2. Get a vertex U and place it in the sorted sequence i.e., array or another queue.
3. For all edges (U, V) update the indegree of V, and put U in the queue if the updated indegree is 0.
4. Perform step 2 and 3 while the queue is not empty.

Complexity : The number of operations is $O(|E| + |V|)$, where |V| is the number of vertices and |E| is the number of edges.

Q 7. Explain Dijkstra's algorithm.

Ans. Dijkstra's Algorithm : Dijkstra's algorithm solves the single-source shortest-path problem of finding shortest paths from a given vertex (the source) to all the other vertices of a weighted graph or digraph. It works as prim's algorithm but compares path lengths rather than edge lengths. Dijkstra's algorithm always yields a correct solution for a graph with non-negative weights.

Q 8. Explain Dijkstra's algorithm with the help of example. What is its time complexity. **(PTU, May 2017 ; Dec. 2017, 2013)**

Ans. Dijkstra's Algorithm : Dijkstra's algorithm finds the length of an optimal path between two vertices in a graph. Optimal can mean shortest or cheapest or fastest or optimal in some other sense : it depends on how you choose to label the edges of the graph. One can find the shortest path from a given source to all points in a graph in the same time, hence this prob is sometimes called the **single source shortest paths** problem.

Dijkstra's Algorithm – Relax

Relax(vertex u, vertex v, weight w)

if d[v] > d[u] + w(u, v) then

d[v] ← d[u] + w(u, v)

p[v] ← u

Dijkstra's algorithm – Idea

- Initialize wt[v] = ∞ and wt[s] = 0
- Insert all vertices v on to PQ with priorities wt[v].
- Repeatedly delete node v from PQ that has min wt[v].

add v to S

for each v – w , relax v – w

Dijkstra's Algorithm – SSSP – Dijkstra

SSSP – Dijkstra(graph (G, w), vertex S)

InitializeSingleSource(G, S)

S ← ϕ

Q ← V[G]

While Q ≠ 0 do

u ← ExtractMin(Q)

S ← S∪{u}

for V ∈ Adj[u] do

Relax(u, v, w)

InitializeSingleSource(graph G, vertex s)

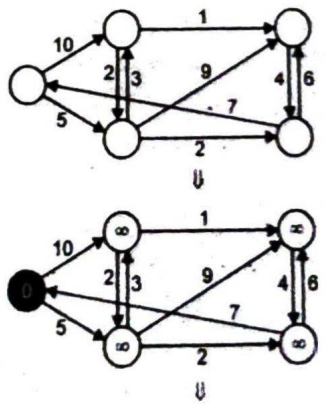
for v ∈ V[G] do

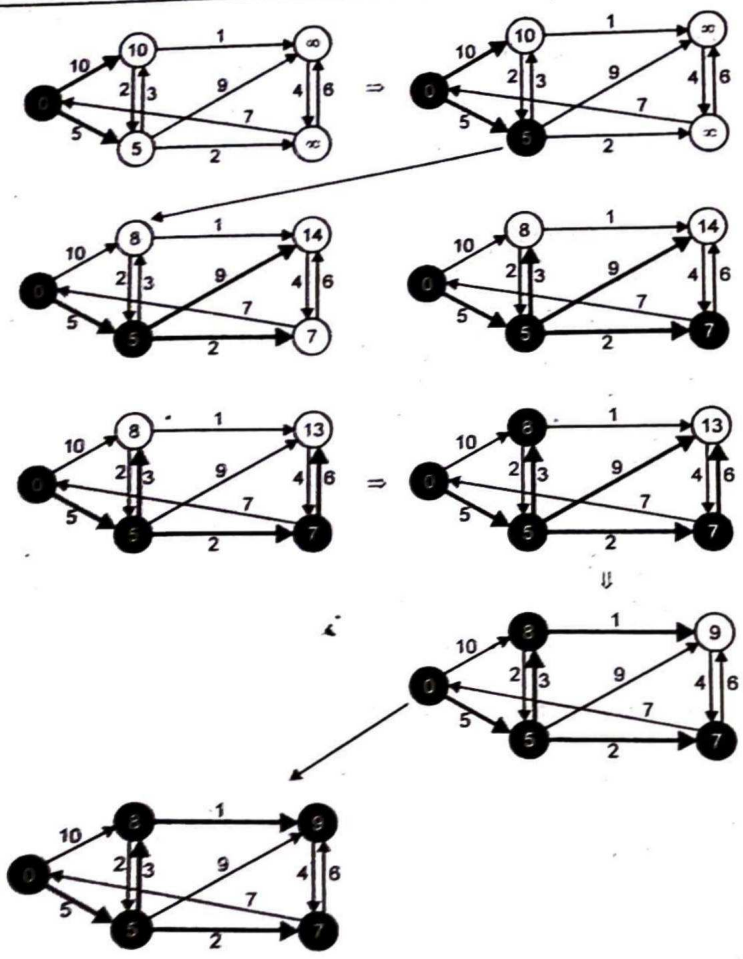
d[v] ← ∞

p[v] ← 0

d[s] ← 0

Example :





Time complexity : Time complexity of dijkaska's algorithm is O [E Log V]

Q 9. Explain Bellman-Ford algorithm with the help of suitable example.
(PTU, May 2015)

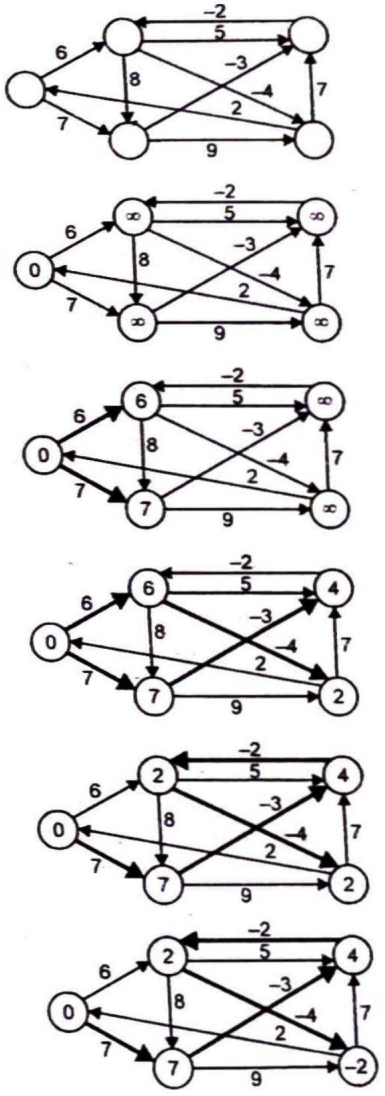
Ans. Idea in Bellman-Ford algorithm :

- Repeat the following |V| – 1 times :
  relax each edge in E.
- Test if there is any negative weight cycle by checking if d[v] > d[u] + w(u, v) for each edge (u, v).

Bellman-Ford Algorithm – SSSP-BellmanFord

```
SSSP–BellmanFord(graph (G, w),Vertex s)
InitializeSingleSource(G, s)
    for i ← 1 to |V [G] – 1| do
        for (u, v) ∈ E(G) do
        Relax(u, v, w)
    for (u, v) ∈ E(G) do
        if d[v] > d[u] + w(u, v) then
            return false
    return true.
```

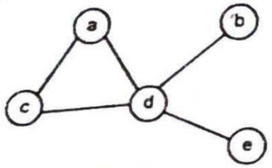

Example of Bellman-Ford Algorithm



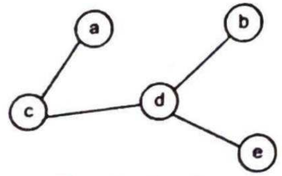
Q 10. What is spanning trees? **(PTU, Dec. 2015 ; May 2015, 2014)**

Ans. Spanning Trees : A subgraph T of a undirected graph G = (V, E) is a spanning tree of G if it is a tree and contain every vertex of G.

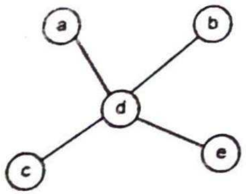
Example :



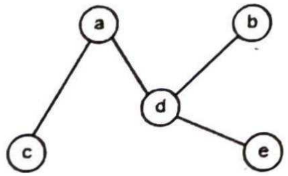
Graph



Spanning tree 1



Spanning tree 2



Spanning tree 3

Q 11. What is minimum spanning tree (MST) and its applications? **(PTU, Dec. 2018)**

Ans. Minimum spanning tree : A minimum spanning tree is a subgraph of an undirected weighted graph G, such that :

- It is a tree (i.e., it is acyclic)
- It covers all the vertices V.
  It contains |V| – 1 edges.
- The total cost associated with tree edges is the minimum among all possible spanning trees.
- Not necessarily unique.

Applications of MST :

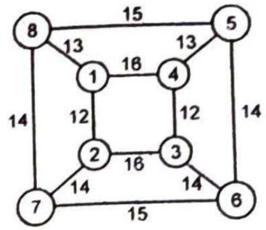
1. Any time you want to visit all vertices in a graph at minimum cost (e.g., wire routing on printed circuit boards, sewer pipe layout, road planning....)
2. Internet content distribution
   $$$, also a hot research topic. Publisher produces web pages, content distribution network replicates web pages to many locations so consumers can access at higher speed. Minimum spanning tree may not be good enough! i.e., content distribution on minimum cost tree may take a long time!
3. Provides a heuristic for traveling salesman problems. The optimum traveling salesman tour is at most twice the length of the minimum spanning tree.

Q 12. Explain Kruskal's algorithm with the help of example.

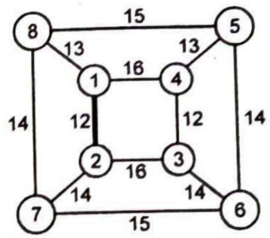
Ans. Kruskal's algorithm :

1. Arrange all edges in a list (L) in non-decreasing order.
2. Select edges from L, and include that in set T, avoid cycle.
3. Repeat 3 until T becomes a tree that cover all vertices.

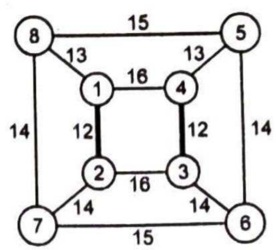
Kruskal's algorithm



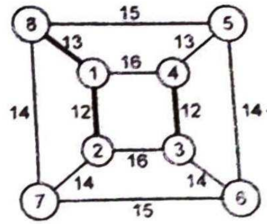
{1, 2}	1 2
{3, 4}	1 2
{1, 8}	1 3
{4, 5}	1 3
{2, 7}	1 4
{3, 6}	14
{7, 8}	14
{5, 6}	14
{5, 8}	15
{6, 7}	15
{1, 4}	16
{2, 3}	16



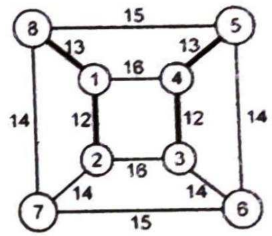
{1, 2}	**12**
{3, 4}	12
{1, 8}	13
{4, 5}	13
{2, 7}	14
{3, 6}	14
{7, 8}	14
{5, 6}	14
{5; 8}	15
{6, 7}	15
{1, 4}	16
{2, 3}	16



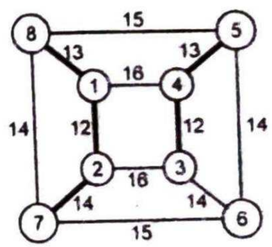
{1, 2}	**12**
{3, 4}	**12**
{1, 8}	13
{4, 5}	13
{2, 7}	14
{3, 6}	14
{7, 8}	14
{5, 6}	14
{5, 8}	15
{6, 7}	15
{1, 4}	16
{2, 3}	16



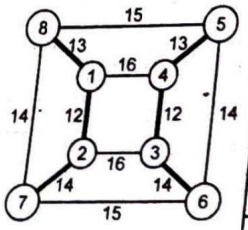
{1, 2}	**12**
{3, 4}	**12**
{1, 8}	**13**
{4, 5}	13
{2, 7}	14
{3, 6}	14
{7, 8}	14
{5, 6}	14
{5, 8}	15
{6, 7}	15
{1, 4}	16
{2, 3}	16



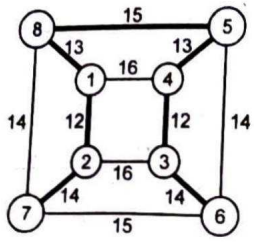
{1, 2}	**12**
{3, 4}	**12**
{1, 8}	**13**
{4, 5}	**13**
{2, 7}	14
{3, 6}	14
{7, 8}	14
{5, 6}	14
{5, 8}	15
{6, 7}	15
{1, 4}	16
{2, 3} .	16



{1, 2}	**12**
{3, 4}	**12**
{1, 8}	**13**
{4, 5}	**13**
{2, 7}	**14**
{3, 6}	14
{7, 8}	14
{5, 6}	14
{5, 8}	15
{6, 7}	15
{1, 4}	16
{2, 3}	16

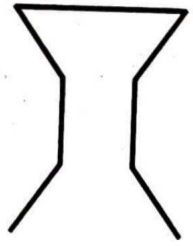


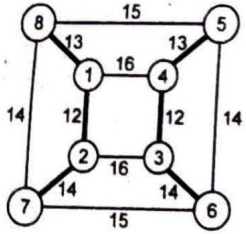
{1, 2}	12
{3, 4}	12
{1, 8}	13
{4, 5}	13
{2, 7}	14
{3, 6}	14
{7, 8}	14
{5, 6}	14
{5, 8}	15
{6, 7}	15
{1, 4}	16
{2, 3}	16



{1, 2}	12
{3, 4}	12
{1, 8}	13
{4, 5}	13
{2, 7}	14
{3, 6}	14
{7, 8}	14
{5, 6}	14
{5, 8}	**15**
{6, 7}	15
{1, 4}	16
{2, 3}	16

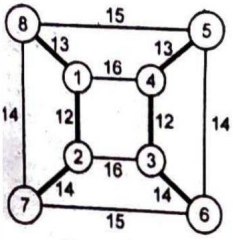
Minimum Spanning Tree





Skip {7,8} to avoid cycle

{1, 2}	12	
{3, 4}	12	
{1, 8}	13	
{4, 5}	13	
{2, 7}	14	
{3, 6}	14	
{7, 8}	14	Skip
{5, 6}	14	
{5, 8}	15	
{6, 7}	15	
{1, 4}	16	
{2, 3}	16	



Skip {5,6} to avoid cycle

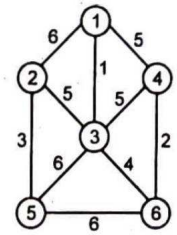
{1, 2}	12	
{3, 4}	12	
{1, 8}	13	
{4, 5}	13	
{2, 7}	14	
{3, 6}	14	
{7, 8}	14	Skip
{5, 6}	14	Skip
{5, 8}	15	
{6, 7}	15	
{1, 4}	16	
{2, 3}	16	

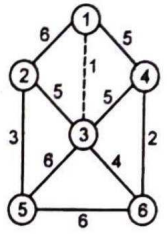
Q 13. Explain Prim's algorithm with the help of example. **(PTU, May 2014)**

Ans. Prim's algorithm :

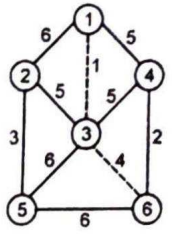
1. Start from any arbitrary vertex.
2. Find the edge that has minimum weight from all known vertices.
3. Stop when the tree covers all vertices.

Prim's

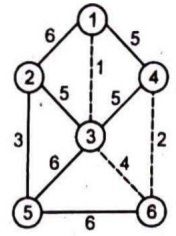




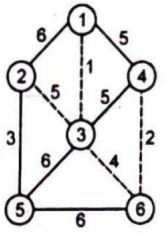
Iteration 1. U = {1}



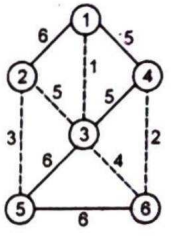
Iteration 2. U = {1,3}



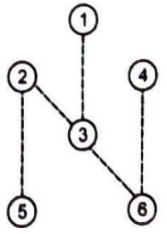
Iteration 3. U = {1,3,6}



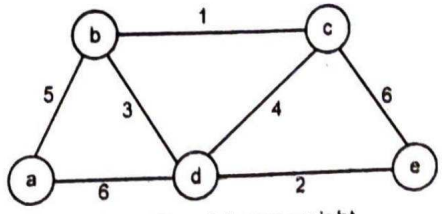
Iteration 4. U = {1,3,6,4}



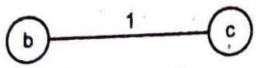
Iteration 1. U = {1,3,6,4,2}



Q 14. Apply Kruskal's algorithm to find a minimum spanning tree of a given graph. **(PTU, Dec. 2011)**

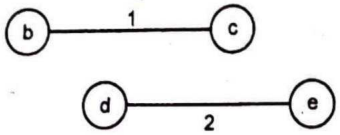


Ans. 1. We first select an edeg with minimum weight



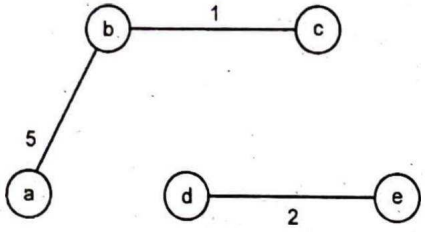
Total Cost = 1

2. Then we select the next minimum weighted edge. It is not necessary that selected edge is adjacent.



Total Cost = 3

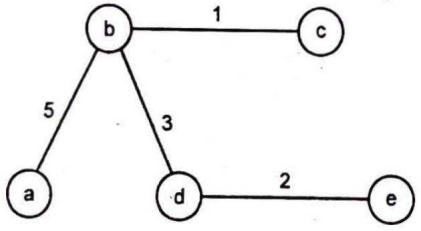
3. Then we select next minimum weight for an unvisited vertice



Total Cost = 8

4. All the vertices are visited but since the spanning tree should be connected one. Hence we select an edge with minimum weight.

Thus we get a minimum spanning tree with Kruskal's algorithm



Total Cost = 12

Q 15. Let n = 5 (P_1, P_2, ... P_5) = (18, 14, 12, 5, 1) and (d_1, d_2, ... d_5) = (2, 2, 1, 3, 3). Find optimal solution. **(PTU, Dec. 2006)**

Ans.

Tasks	T_2	T_1	T_4	
Time →	0	1	2	3

Profit = $(P_1 + P_2 + P_4)$ = 18 + 14 + 5 = 37

Then optimal schedule (1, 2, 4) wiith profit 37.

Q 16. What is minimum cost spanning tree algorithm? **(PTU, May 2006)**

Ans. Let G = (V, E) be an undirected connected graph. A sub-graph t = (V, E^1) of G is a spanning tree of G if and only if t is a tree.

Q 17. What is depth-first search algorithm? **(PTU, May 2006)**

Ans. Depth-first search (DFS) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.

Q 18. What is Eulerian cycle in a graph? **(PTU, Dec. 2006)**

Ans. In graph theory, an **Eulerian trail** is a trail in a graph which visits every edge exactly once. Similarly, an **Eulerian circuit** or **Eulerian cycle** is an Eulerian trail which starts and ends on the same vertex. They were first discussed by Leonhard Euler while solving the famous Seven Bridges of Konigsberg problem in 1736. Mathematically the problem can be stated like this :

Given the graph on the right, is it possible to construct a path (or a cycle, i.e. a path starting and ending of the same vertex) which visits each edge exactly once.

Q 19. What are row major and column major ordering? **(PTU, May 2009)**

Ans. n computing, **row-major order** and **column-major order** describe methods for storing multidimensional arrays in linear memory. Following standard matrix notation, rows are identified by the first index of a two-dimensional array and columns by the second index. Array layout is critical for correctly passing arrays between programs written in different languages. It is also important for performance when traversing an array because accessing array elements that are contiguous in memory is usually faster than accessing elements which are not, due to caching. Row-major order is used in C ; column-major order is used in Fortran and MATLAB.

Q 20. Describe a path in an undirected path. **(PTU, Dec. 2010, 2007)**

Ans. Sequence of vertices, such that there is an edge from each vertex to the next in sequence, is called path. First vertex is the path is called the start vertex ; the last vertex in the path is called the end vertex. If start and end vertices are the same, path is called cycle. Path is called simple, if it includes every vertex only once. Cycle is called simple, if it includes every vertex, except start one, only once.

Q 21. List the uses of graph coloring. **(PTU, Dec. 2011)**

Ans. Graph coloring is an arbitrary assignment of labels (colors) to objects within graph. Such objects can be vertices, edges, faces or a mixture of these. A graph coloring is distinct from a graph labeling in that in the former, the same label may be used more than once.

The applications of graph coloring found in following areas :

1. Scheduling
2. Register allocation in compilers
3. Frequency assignment in mobile radios
4. Pattern matching
5. Sudoku.

Q 22. Define Kruskal's algorithm. **(PTU, May 2008)**

Ans. Kruskal's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm.

Description :

- create a forest F (a set of trees), where each vertex in the graph is a separate tree
- create a set S containing all the edges in the graph
- while S is nonempty all F is not yet spanning
  - remove an edge with minimum weight from S
  - if that edge connects two different trees, then add it to the forest, combining two trees into a single tree otherwise discard that edge.

At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.

Q 23. What are Prim's and Kruskal's algorithms for minimum cost spanning tree? **(PTU, May 2011)**

Ans. Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

Description : The only spanning tree of the empty graph is again the empty graph. The following description assumes that this special case is handled separately.

The algorithm continuously increase the size of a tree, one edge at a time, starting with a tree consisting of a single vertex, until it spans all vertices.

- **Input :** A non empty connected weighted graph with vertics V and edges E.
- **Initialize :** $V_{new} = \{x\}$, where x is an arbitrary node (starting point) from V, $E_{new} = \{\}$
- Repeat until $V_{new} = V$:
  - > Choose an edge (u, v) with minimal weight such that us is in V_{new} and v is not.
  - > Add V to V_{new} and (u, v) to E_{new}
- **Output :** V_{new} and E_{new} describe a minimal spanning tree.

Kruskal's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted graph. This mean it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If graph is not connected, then it finds a minimum spanning forest.

Description :

- Create a forest F (a set of trees), where each vertex in the graph is a separate tree
- Create a set S containing all the edges in the graph
- While S in non empty and F is not yet spanning
  - > remove an edge with minimum weight from S
  - > if that edge connects two different trees, then add it to the forest, combining two trees into a single tree
  - > other wise discard that edge.

At the termination of algorithm, the forest has only one component and forms a minimum spanning tree of the graph.

Q 24. Define a minimum spanning tree. Write Prim's algorithm to find minimum spanning tree. **(PTU, Dec. 2008)**

Ans. A minimum spanning tree of a weighted connected graph is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges.

Prime's algorithm to find minimum spanning tree :

Procedure prime (G, W, S)

The above procedure subalgorithm finds the minimum spanning tree of a given 'g'. The procedure takes the advantages of priority queue data structure. It uses three array 'color', 'pre' and 'key'.

Step 1. Initialization

For a ← V_1, V_2, ... V_3 a ∈ V
{
Set key [a] ← + ∞
Set color [a] ← white
} End of loop

Step 2. Start at root vertex

Set key [S] ← 0
Set pre [S] ← Null
Set Q ← call to prio queues (V)
put vertices in Q

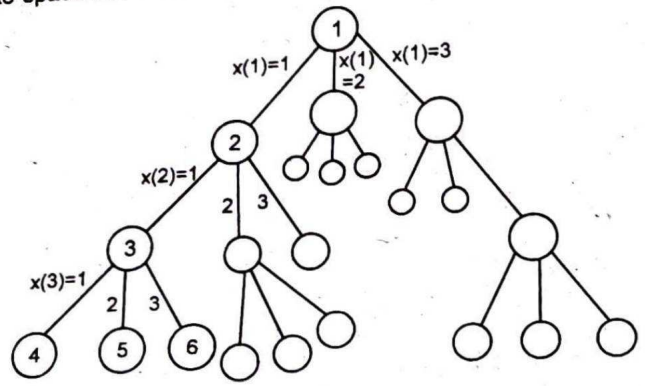
Step 3. Loop, searched until all vertices in MST

While (non-empty) (Q))
{
Set a ← Extract_Minimum (Q) vertex with light edge
start loop 2
For V ← V_1, V_2, ... Adj [a] V ∈ Adj [a]
{
If (color [V] = white) & & (W (a, V) < key [V]) then
{
Set key [V] ← W (a, V)
new lighter edge out of V
Call to decrease key (Q, V, key [V])
Set pre [V] ← a
End of loop 2
Set color [a] ← black
End of loop 1

Step 4. return at the point of call
return.

Q 25. Draw the state space tree for m coloring when n = 3 and m = 3. **(PTU, Dec. 2011)**

Ans. State space tree for m colouring with n = 3 and m = 3



Q 26. Define a minimum spanning tree. Write Kruskal's algorithm to find minimum panning tree. **(PTU, May 2010, 2009)**

Ans. A minimum spanning tree : A connected, undirected graph, a spanning tree of at graph is a subgraph that is a tree and connects all the vertices together. A single graph in have many different spanning trees. We can also assign a weight to each edge, which is number representing how unfavourable it is, and use this to assign a weight to a spanning e by computing the sum of the weights of the edges in that spanning tree. A minimum anning tree (MST) or minimum weight spanning tree is then a spanning tree with weight s than or equal to the weight of every other spanning tree. More generally, any undirected aph (not necessarily connected) has a minimum spanning forest, which is a union of minimum anning trees for its connected components.

Kruskal's Algorithm : One idea given by J.B. Kruskal is the following. A minimum anning tree T is built edge by edge. The edges are sorted in the non-decreasing order of ir costs. At each stage, we choose the smallest unused edge that does not form a cycle n the already chosen edges. The algorithm Kruskal is described below.

Procedure Kruskal (G : graph) :

{Finds a set T of edges of G = (V, E) which forms a minimum spanning tree of G.}
var T : set, u, : vertex ;
begin
initialize T to be empty ;
while |T| < |V| – 1 and |E| > 0 do
begin
choose edge (u, v) of the lowest weight in E ;
remove (u, v) from E ;
if edge (u, v) does not form a cycle with edges in T then

insert (u, v) into T
end ;
if |T| < |V| – 1 then write ('no spanning tree') {G is not connected}
end ;

Kruskal's algorithm is a greedy algorithm, since it makes the locally optimal (i.e., greedy) decision at each stage. In general, this strategy does not guarantee that the result is globally optimal, although it is locally optimal.

Q 27. Explain quick-union and quick-find set algorithms. Give suitable examples. **(PTU, May 2011)**

Ans. Disjoint Sets : Disjoint sets is a data structure which partitions a set of items. A partition is a set of sets such that each item is in one and only one set. It has operations :

makeset (x) : makes a set from a single item
find (x) : finds the set that x belongs to
union (x, y) : makes the union of the sets containing x and y

We can assume that the items are represented by integers, which can be the index into an array.

There are two popular implementation for disjoint sets, using linked lists or using trees.

Quick Find : Uses linked lists to represent the sets, and an array, called representative array, which is indexed by the item number and the value give the set name (smallest integer number in the set). The operation makeset is obvious, update the representative array and make the single element link list. The cost is Ⓗ(1).

The operation find is also obvious, just access the representative array. The cost is Ⓗ (1).

The operation union is more expensive. Join the two link list (easy enough) but the representative arrays must be update. The cost is linear in the set size.

For sequence of n random unions the cost is Ⓗ (n^2). We can do better if the set name of the representative array is the larger set, then algorithm only needs to update the representative array for the smaller array. This is called union by size. Then the cost is O (n log n) because the set size doubles after each union. The cost of n – 1 unions and m finds is O (n log n + m).

Quick Union : The implementation uses trees of the items to represent the sets. The integer in the root of the tree is the set name. The links of the tree point from the children to the parent note this is not a binary tree and the links point in the opposite direction of the most trees.

The operation makeset is obvious, just make a single node tree. The cost is Ⓗ(1). The operation union links the root of one tree to the root of the other tree. The cost is Ⓗ(1).

The operation find requires traversing up the tree and cost Ⓗ(h), where h is the height of the tree. The height could be on the order of the set size. To control the cost, the union

should make the smaller tree in the union operation the sub tree of the larger tree. This is union by size (by set size) or union by rank (by tree height). Naturally, this requires storing the tree size or height in the root. Using union by size or rank the height of the tree is logarithm with the number of unions (in other words the tree/set size). The cost for n – 1 unions and m finds is O (n + m log n).

We can do even better by using path compression. Path compression makes every node encounter during a find linked with the root directly. Then a sequence of n – 1 unions and m finds is only slightly more than linear in n and m.

Q 28. Consider the undirected weighted graph in fig 1. Find a minimum spanning tree for the graph using Kruskal's algorithm. **(PTU, Dec. 2013)**

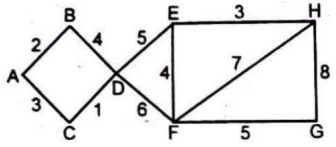
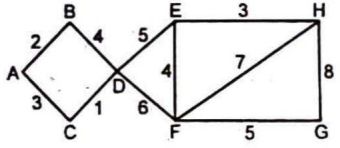


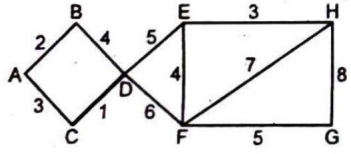
Fig. 1

Ans.

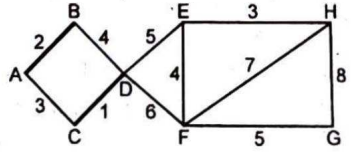


C – D → 1
A – B → 2
A – C → 3
E – H → 3
B – D → 4
E – F → 4
D – E → 5
F – G → 5
D – F → 6
F – H → 7
G – H → 8

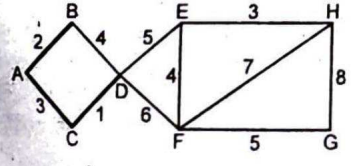
Step 1. C – D → 1



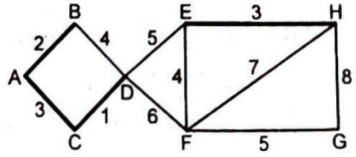
Step 2. A – B → 2



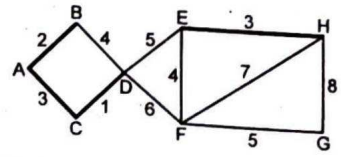
Step 3. A – C → 3



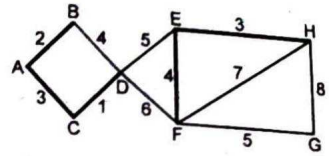
Step 4. E – H → 3



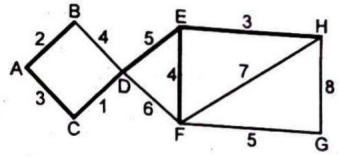
Step 5. Skip B – D → 4 to avoid cycle



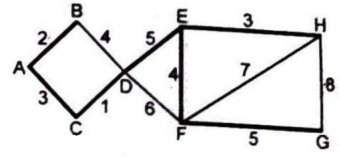
Step 6. E – F → 4

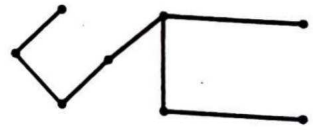


Step 7. D – E → 5



Step 8. F – G → 5





Q 29. What is difference between Dijkstra and Bellman Ford algorithms for solving single source shortest path problem? **(PTU, May 2014)**

Ans. Both Dijkstra and Bellman Ford solves the single source shortest path problem but the primary difference in the function of the two algorithm is that Dijkstra's algorithm can not handle negative edge weights. Bellman-Ford's algorithm can handle some edges with negative weights. It must be remembered, however, that if there is a negative cycle there is no shortest path.

Q 30. What is difference between Prim's and Kruskal's algorithm for finding minimum cost spanning tree? **(PTU, Dec. 2019, 2017 ; May 2014)**

Ans. Prim's algorithm is for obtaining minimum spanning tree by selecting the adjacent vertices of already selected vertices.

Kruskal's algorithm is for obtaining minimum spanning tree but it is not necessary to choose adjacent vertices of already selected vertices.

The main difference between Prim's and kruskal's is that kruskal does not require the edge e to be connected to the evolving tree T. That means that T isn't necessarily connected at intermediate steps in kruskal's algorithm. So strictly speaking the T in Kruskal's algorithm is a forest and not a tree. Kruskal's algorithm can also be implemented easily in O(m log n) time.

- Both have the same output i.e. minimum spanning tree.
- Kruskal's begins with forest and merge into a tree.
- Prim's always stays as a tree.
- Unlike Kruskal's Prims doesn't need to see all of the graph at once. It can deal with

it one piece at a time. It also doesn't need to worry if adding an edge will create a cycle since this algorithm deals primarily with the nodes, and not the edges.

Q 31. What is the difference between a Live Node and Dead Node ? **(PTU, Dec. 2014)**

Ans. Live Node : A node which has been generated and all of whose children have not yet been generated is called as a live node.

Dead node : Dead node is defined as a generated node, which is to be expanded further all of whose children have been generated.

Q 32. State shortest path problem for a graph. **(PTU, May 2015)**

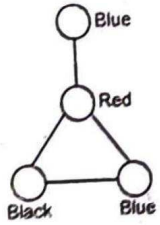
Ans. In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weigths of its constituent edges is minimized. The shortest path problem can be defined for graphs whether undirected, directed or mixed.

Q 33. Explain concept of graph coloring with suitable example.

Ans. There are three main concepts are related to graph coloring.

1. Vertex coloring (the default)
2. Edge coloring
3. Face coloring (planar)

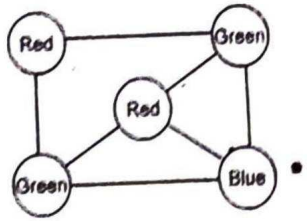
1. Vertex coloring : Vertex coloring is the most common graph coloring problem. The problem is, given n colors, find a way of coloring the vertices of a graph such that no two adjacent vertices are colored using same color.



2. Edge coloring : An edge coloring assign a color to each edge so that no two incident edges share the same color.

3. Face coloring : A face coloring of a planar graph assigns a color to each face or region so that no two faces that share a boundary share the same color.

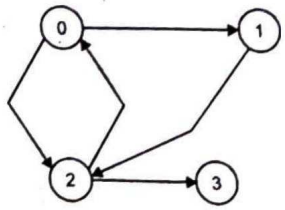
Chromatic number : The smallest number of colors needed to color a graph G is called its chromatic number. For example, the following can be colored minimum 3 colors.



Q 34. Explain transitive closure of a graph.

Ans. Given a directed graph, find out if a vertex j is reachable from another vertex 1 for all vertex parts (i, j) in the given graph. Here reachable mean that there is a path from vertex i to j. The reachability matrix is called transitive closure of a graph.

e.g. : Consider below graph



Transitive closure of above graph is

1	1	1	1
1	1	1	1
1	1	1	1
0	0	0	1

The graph is given in the form of adjacency matrix say graph [v] [v] where graph [i] [j] is 1. If there is an edge. from vertex i to vertex j or i is equal to j, otherwise graph [i] [j] is 0. Floyd Warshall Algorithms can be used, we can calculate the distance matrix dist [v] [v] using Floyd Warshall, If dist [i] [j] is infinite, then j is not reachable from i, otherwise j is reachable and value of dist [i] [j] will be less than V. Instead of directly using Floyd Warshall, we can optimize it in terms of space and time, for this particular problem. Following are the optimizations :

1. Instead of integer resultant matrix (dist [v] [v] in floyd warshall), we can create a boolean reachability matrix reach [v] [v]. The value reach [i] [j] will be i if j is reachable from i, otherwise 0.
2. Instead of using arithmatic operations, we can use logical operations. For arthematic operation '+', logical and '&&' is used, and for min, logical or '||' is used.

Q 35. What is advantage of binary search over linear search ? Also state limitations of binary search. **(PTU, May 2017)**

Ans. Binary search method is a method to search a specific data from a large volume of data. In this method, data at the middle is checked, if it is found search is completed otherwise that half is selected in which data can be found and this process continues till the data is found e.g. as we do to find a word in a dictionary.

Advantages of binary search on linear search : A binary search runs in 0 (log n) time, compared to linear search's 0(n) time. What this means is that the more elements are present in the search array, the faster a binary search will be compared to a linear search. As an example, given 100 elements, a binary search will discover the item using no more than 7

iterations, while a linear search will require upto 100 iterations, going to upto 1000 elements requires only up to 10 iterations, compared to linear search's 1000 maximum iterations. The downside to binary search, however is, it only operates on a sorted array, which means the data must be pre-sorted using some means.

Disadvantages of using Binary search : The binary search algorithm works as access the middle element of list. This means that the list must be stored in some type of array, the problem occur when inserting an array require move down the elements of the list and in case of deleting from an array moved up the elements of the list.

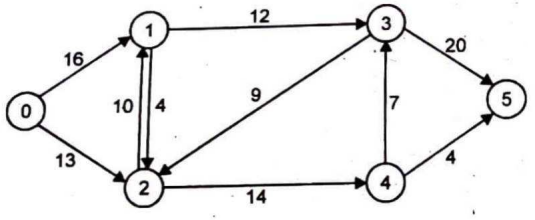
Q 36. Explain Ford Fulkerson Algorithm for Maximum flow problem.

Ans. Given a graph which represents a flow network where every edge has a capacity. Also given two vertices source 's' and sink '+' in the graph, find the maximum possible flow from s to t with following constraints :

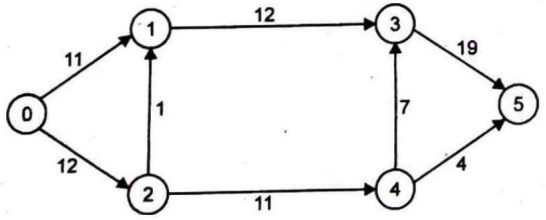
(a) Flow on an edge doesn't exceed the given capacity of the edge.

(b) Incoming, flow is equal to outgoing flow, for every vertex except s and t.

e.g. Consider the following graph from CLRS book.



The maximum possible flow in the above graph is 23.



Prerequisite : Max flow problem

Introduction

Ford-Fulkerson Algorithm : The following is simple idea of Ford-Fulkerson algorithm

1. Start with initial flow as 0.
2. While there is a augmenting path from source to sink

Add this path-flow to flow

3. Return Flow

Time Complexity : Time Complexity of the above algorithm is 0 (max – flow * e). We

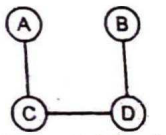
run a loop while there is an augmenting path. In worst case, we may add 1 unit flow in every iteration. Therefore the time complexity becomes 0 (max – flow * e).

How to implement the above simple algorithms : Let us first define the concept of Residual Graph. Which is needed for understanding the implementation. Residual Graph is a flow network is a graph which indicates additional possible flow. If there is a path from source to sink in residual graph, then it is possible to add flow. Every edge of a residual graph has a value called residual capacity which is equal to original capacity of the edge minus current flow. Residual capacity is basically the current capacity of the edge. Residual capacity is 0 If there is no edge between two vertices of residual graph. We can initialize the residual graph as original graph as there is no initial flow and initially residual capacity is equal to original capacity. To find an augmenting path, we can either do a BFS or DFS of the residual graph. We have used BFS in below implementation. Using BFS, we can find out of there is a path from source to sink. BFS also builds parent [ ] array. Using the parent [ ] array, we traverse through the found path and find possible flow through this path by finding minimum residual capacity along the path. We later add the found path flow to overall flow.

The important thing is we need to update residual capacities in the residual graph. We subtract path flow from all edges along the path and we add path flow along the reverse edges. We need to add path flow along reverse edges because we may later need to send flow in reverse direction.

Q 37. Define connected components. **(PTU, Dec. 2015)**

Ans. A graph is said to be connected if at least one path exists between every pair of vertices in the graph. Alternatively, two vertices are defined to be in the same connected component if there exists a path between them. Other words we can also say that If G is a connected undirected graph, then we can visit all the vertices of the graph in the first call to BFS. The subgraph which we obtain after traversing the graph using BFS represents the connected component of the graph.

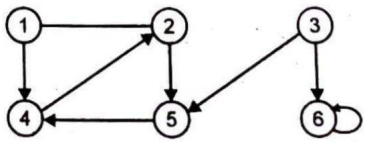


Connected Graph

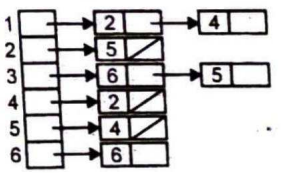
Q 38. From a given adjacency list representation of a directed graph how do you find the in degree and out degree of the vertices ? Analyse the algorithm.

(PTU, Dec. 2015

Ans.



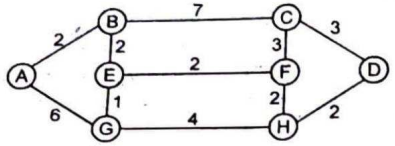
Directed graph



Adjacency-List representation of a Directed graph

The in-degree of a vertex is the number of edges entering it. The out–degree of a vertex is the number of edges leaving it.

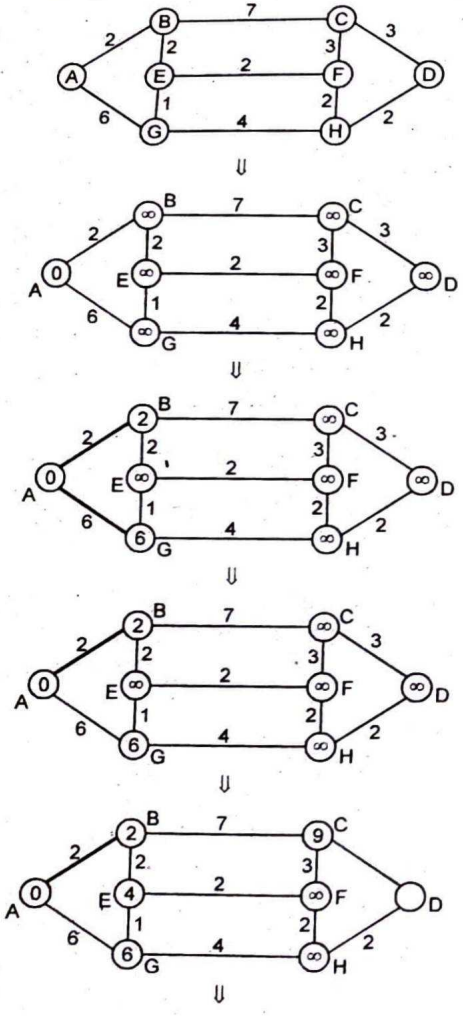
Q 39. Using Dijkstra's algorithm find the shortest path from A to D for the following graph.

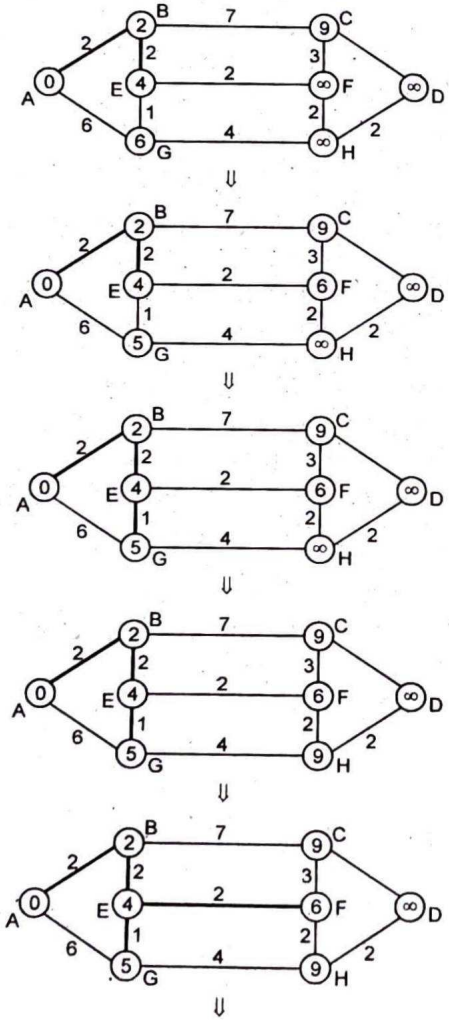


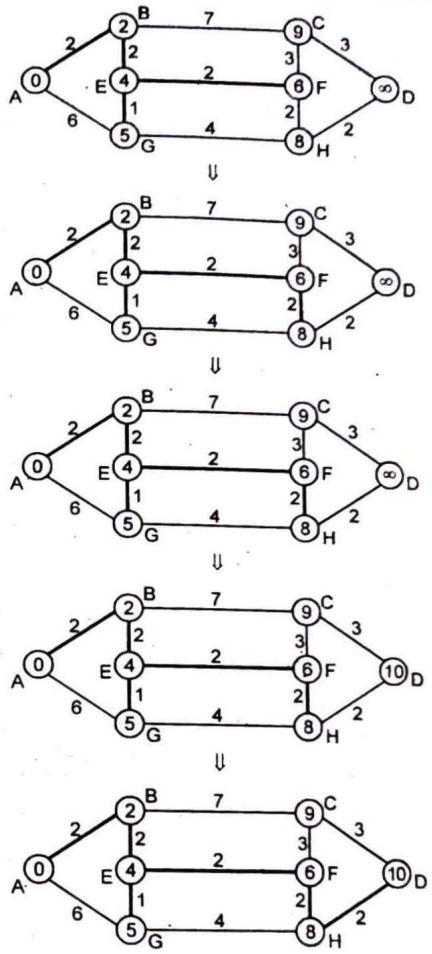
(PTU, May 2019)

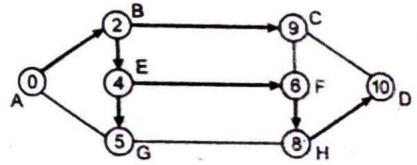
Ans. *Refer to Q.No. 8*

Example :









Q 40 Differentiate between graph and a tree. **(PTU, May 2016)**

Ans. The graph and the tree both are the collection of nodes and the edges but the main difference between the two is that in tree there is an unique node called as root from which the subtrees arise whereas in the graph no such root node is there.

Q 41. Extend the Dijkastra's algorithm to find All-pairs-shortest-path (ASSP) problem. **(PTU, May 2018)**

Ans. Given a directed graph G = (V, E) where each edge (V, W) has a non negative cost C [V, W] for all pairs of vertices (V, W) find the cost of the lowest cost path from V to W.

A generalization of the single source shortest path problem.

Use Dijkstra's algorithm varying the source node among all the modes in the graph.

We will consider a slight extension to this problem find the lowest cost path between each pair of vertices.

We must recover the path itself and not just the cost of the path.

Q 42. What are the applications of BST ? **(PTU, Dec. 2019)**

Ans. 1. A self balancing BST is used to maintain sorted stream of data.

2. A self balancing BST is used to implement double ended priority queue.

Q 43. Suppose that M [1.....n] is an array containing a Min-Heap. Give pseudeode for an algorithm Extract Min [H,n] that remain the smalling element from the heap M of size n and returns its value. Analyse the time complexity of your algorithm. Explain your algorithm. **(PTU, Dec. 2019)**

Ans. Procedure : Min-Heapify (v, i)

Input : v : an array of elements ; i : an index array.

Output : v : modified such that element i roots a min-heap

l ← left (i);

r ← Right (i);

If l ≤ |v| and v [l] < v [i] then

min ← l;

else

min ← i;

If r ≤ |v| and v [r] < v [min] then

min ← r;

If min ≠ i then

Exchange v[i] and v[min];

min = Heapify (v, min);

Procedure MHSA (V, P)

Input : v : Computer array, sorted by capacity in decreasing order, P; set (queue) of process.

Output : A scheduling of set P over set v.

Build = Min = Heap(v);

While P is not empty do

Dequeue process Pi;

Assign process pi to Computer Vroot located at min-heaproot,

update the load of Computer Vroot; Min-Heapify (V, root);

Q 44. A max heap is given with n elements and its height is log(n). Write an efficient algorithm to find minimum element in heap. Also calculate the time and space complexity. **(PTU, Dec. 2019)**

Ans. Algorithm :

In each step you need traverse both left and right subtrees in order to search for the minimum element.

Min element from Max Heap :

1. Search at last level = 0 (n/2) = 0(n)
2. Replace searched element with last element and decrease heap size by 1 = 0(1).
3. Apply more heapify on replaced element = 0(logn)

The time complexity of above approach is 0(n).

❑❑❑

Chapter 4

Tractable and Intractable Problems

Contents

Computability of Algorithms, Computability classes - P, NP, NP-complete and NP-hard. Cook's theorem, Standard NP-complete problems and Reduction techniques.

POINTS TO REMEMBER

1. A computational problem is the encoded format of a problem, which is independent of specific input .
2. A computational problem which only answers in the form of Yes-or-No is called decision problem.
3. A decision problem is called decidable or effectively solvable if it is a recursive set.
4. P is also known as DTIME/PTIME, which is one of the most primary complexity classes.
5. A P-problem is always lies in NP.
6. Linear programming is known to be NP and not to be P. It was proposed by L. Khachian in 1979.
7. Class P is a class of decision problems that can be solved in polynomial time by (deterministic) algorithms.
8. Class NP is the class of decision problems that can be solved by nondeterministic polynomial algorithms.
9. All NP-complete problems are NP-hard problems but some NP-hard problems are not NP-complete problems.
10. There are two types of polynomial reductions. These are :
    (i) Karp Reductions
    (ii) Cook Reductions.
11. A Hamiltonian circuit, also known as a Hamiltonial cycle.
12. There are three main concepts are related to graph coloring. These are :
    (i) Vertex coloring
    (ii) Edge coloring
    (iii) Face coloring

1. Karp Reductions : Problem X is polynomial-time (Karp) reducible to problem Y if any instance of problem X can be solved using

(i) Polynomially many standard computation steps, plus

(ii) One call on some instance to an algorithm that solves problem Y.

2. Cook Reductions : Problem X is polynomial-time (cook) reducible to problem Y if any instance of problem X can be solved using

(i) Polynomially many standard computation steps, plus.

(ii) Polynomially many call on some instance to an algorithm that solves problem Y.

Q 7. Explain Cooks theorem.

Ans. Cook's theorem : The Cook's theorem shows that the satisfiability problem is NP-complete. Without loss of generality we assume that languages in NP are over the alphabet {0, 1}. Lemma 1, useful for the proof, states that we can restrict the form of a computation of a NTM that accepts languages in NP.

Lemma 1 : If $L \in NP$, then L is accepted by a 1-tape NTM N with alphabet {0, 1} such that for some polynomial p (n), the following properties hold.

1. N's computation is composed of two phases. These are :

(a) The guessing phase

(b) The checking phase

2. In the guessing phase, N non-deterministically writes a string ␣y directly after the input string, and in the checking phase, N behaves deterministically.

3. N uses at most p(n) tape cells, never moves its head to the left of w, and takes exactly p(n) steps in the checking phase.

A boolean formula f over variable set V is in conjunctive normal form (CNF) if

$$f = \Lambda_{i=1}^{m} \; V_{j=1}^{K_i} \; l_{ij}$$

for some value of m and Ki, $1 \le i \le m$, where literal $l_{i,j}$ is either x or $\bar{x}$ for some $x \in V$. For each i, the term $V_{j=1}^{K_i} l_{i,j}$ is called a clause of the formula. f is satisfiable if there exist a truth assignment to the variables in V that sets f to true. CNFSAT is the set of satisfiable Boolean formulas in CNF.

Q 8. Prove that CNFSAT is NP-complete.

Ans. It is not hard to show that CNFSAT$\in$NP. To prove that CNFSAT is NP-complete, we show that for any language $L \in NP$, $L \le_m^p$ CNFSAT.

Let $L \in NP$ and let N be a NTM accepting L that satisfies the properties of Lemma 1. Let the transition function of N be δ. Let the states of N be $q_0, \ldots, q_r$. Let S_0, S_1, S_2 denote 0, 1, ␣, respectively. Assume that the tape cells are numbered consecutively from the left end of the input, starting at 0. On input w of length n, we show how to construct a formula in CNF form fw, which is satisfiable if and only if w is accepted by N. The variables of fw are as follows :

Variables	Q[i, K]	H[i, j]	S[i, j, l]
Range	$0 \le i \le p(n)$ $0 \le k \le r$	$0 \le i \le p(n)$ $0 \le j \le p(n)$	$0 \le i \le p(n)$ $0 \le j \le p(n)$ $0 \le l \le 2$
Meaning	At step i of the checking phase, the state of N is qk.	At step i of the checking phase, the head of N is on tape square j.	At step i of the checking phase, the symbol in square j is sl.

A computation of N naturally corresponds to an assignment of truth values to the variables. Other assignment to the variables may be meaningless. For example, an assignment with Q [i, K] = Q [i, K'] = true, K ≠ K', would imply that N is in two different states at step i, which is impossible. Our goal is to construct fw so that it is satisfied only by assignments to the variables that correspond to accepting computations of N on w . The clauses of fw are constructed to ensure that the following conditions are satisfied :

1. At each step i of the checking phase, N is in exactly 1 state.
2. At each step i, the head is on exactly one tape square.
3. At each step i, there is exactly 1 symbol in each tape square.
4. At step 0 of the checking phase, the state is the initial state of N in its checking phase, and the tape contents are w␣y for some y.
5. A step p(n) of the checking phase, N is in as accepting state.
6. The configuration of N at the (i + 1)st step follows from that at the ith step, by applying the transition function of N.

Consider condition 1. For each i, we have the following clause :

$$Q[i, 0] \vee Q[i, 1] \vee \ldots\ldots \vee Q[i, r].$$

This clause ensures that the machine is in at least 1 state at step i. We also need clauses to ensure that N is not both in state qj and qj' :

$$\overline{Q[i, j]} \vee \overline{Q[i, j']} \text{ for each } j \ne j', \; 0 \le j, j' \le r.$$

Conditions 2 and 3 are handled similarly. Conditions 4 and 5 are quite easy. Finally, consider condition 6. For each (i, j, k, l) we add clauses that ensure the following : If at step i, the tape head of N is pointing to the jth tape cell, N is in state qk, sl is the symbol under the tape head, and (qk, sl, qk', sl', X) $\in \delta$, where $X \in \{L, R\}$ then at step i + 1, the tape head is pointing to the (j + y)th tape cell where y = 1 if X = R and y = –1 if X = L, N is in state qk' and the symbol in cell j is sl'. The following clauses ensure this :

$$\overline{Q[i, K]} \vee \overline{H[i, j]} \vee \overline{S[i, j, l]} \vee Q[i+1, K']$$

$$\overline{Q[i, K]} \vee \overline{H[i, j]} \vee \overline{S[i, j, l]} \vee H[i+1, j+y]$$

$$\overline{Q[i, K]} \vee \overline{H[i, j]} \vee \overline{S[i, j, l]} \vee S[i+1, j, l']$$

All of the clauses for condition 1 to 6 can be computed in polynomial time.

Q 9. Prove that CLIQUE is NP-Complete.

Ans. It is easy to verify that a graph has a clique of size K if we guess the vertices forming the clique. We merely examine the edges. This can be done in polynomial time.

We shall now reduce 3-SAT to CLIQUE. We are given a set of K clauses and must build a graph which has a clique if and only if the clauses are satisfiable. The literals from the clauses become the graph's vertices. And collections of three literals shall make up the clique in the graph we build. Then a truth assignment which makes at least one literals true per clause with force a CLIQUE of size K to appear in the graph. And, if no truth assignment satisfies all of the clauses, there will not be a clique of size K in the graph.

To do this, let every literal in every clause be a vertex of the graph we are building. We wish to be able to connect true literals, but not two from the same clause. And two which are complements cannot both be true at once. So, connect all of the literals which are not in the same clause and are not complement of each other. We are building the graph G = (V, E) where :

$$V = \{<X, i> \mid X \text{ is in the ith clause}\}$$

$$E = \{(<X, i>, <Y, j>) \mid X \neq \overline{Y} \text{ and } i \neq j\}$$

Now we shall claim that if there were K clauses and there is some truth assignment to the variables which satisfies them, then there is a clique of size K in our graph. If the clauses are satisfiable then one literal from each clause is true. That is the clique because a collection of literals (one from each clause) which are all true cannot contain a literal and its complement. And they are all connected by edges because we connected literals not in the same clause (except for complements).

On the other hand, suppose that there is a clique of size K in the graph. These K vertices must have come from different clauses since no two literals from the same clause are connected. And, no literal and its complement are in the clique, so setting the truth assignment to make the literal in the clique true provides satisfaction. A small inspection reveals that the above transformation can indeed be carried out in polynomial time. Thus the CLIQUE problem has been shown to be NP-hard just as we wished.

Q 10. Write Cooks statement. **(PTU, Dec. 2005)**

Ans. In the celebrated Cook-Levin theorem, Cook proved that the Boolean satisfiability problem is NP-complete.

Statement : Satisfiability is in P if and only if P = NP.

Q 11. What is Hamiltonian Cycle? Give suitable example.

Ans. Hamiltonian Cycle : Let G be the given graph which is connected with n vertices.

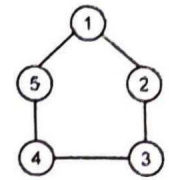
So, G = (V, E)

where V is the set of vertices, E is the set of edges and n is the number of vertices.

A Hamiltonian cycle is a cycle or a round trip which is start from one point or vertex and visits each vertex exactly once and comes back to its starting point.

Or we can also say that a hamiltonian circuit, also known as a Hamiltonian cycle is a path in an undirected graph which touches each vertex exactly once and also return to the starting vertex.

For example :



The graph G contain 5 vertices. There are two Hamiltonian cycle in this

Cycle 1 : $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 1$

The cycle 1 starts from vertex 1 then vertex 2, 3, 4, 5 and comes back to vertex 1. In this cycle each vertex visits exactly once and its starting and ending point is same that is 1.

Cycle 2 : $1 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$

The cycle 2 also starts from vertex 1 then visit vertex 5, 4, 3, 2 and comes back to vertex 1.

Q 12. What are the steps involved in proving a problem NP-complete ? Specify the problems already proved to be NP-complete. **(PTU, Dec. 2014)**

Ans. First, try to prove it to be NP-hard, by

1. finding a related problem which is already found to be NP-hard (choosing such a suitable "source" problem close to your "target" problem, for the purpose of developing poly-trans, is the most difficult step), and then

2. developing a truth-preserving polynomial problem-transformation from that source problem to your target problem (You will have to show the transformation - algorithm's)

(i) Correctness and

(ii) Poly-time complexity.

Significance : If anyone finds poly algorithm for your "target" problem, then by using your poly-trans algorithm one would be able to solve that "source" NP-hard problem in poly-time, or in other words P would be = NP.

Second try to prove that the given problem is in NP class : by developing a polynomial algorithm for checking any "Certificate" of any problem-instance.

- 3-SAT is NP-Complete
- SAT in CNF is NP-Complete
- The CLIQUE PROBLEM is NP-Complete
- The INDEPENDENT SET PROBELM is NP-Complete.

Q 13. Differentiate between deterministic and non-deterministic algorithms. **(PTU, May 2010 ; Dec. 2009)**

Ans. Algorithm is deterministic if for a given input the output generated is same for a function. A mathematical function is deterministic. Hence, the state is known at every step of the algorithm.

Algorithm is non deterministic if there are more than one path the algorithm can take. Due to this, one cannot determine the next state of the machine running the algorithm. Example would be a random function.

Non deterministic machines that can't solve problems in polynomial time are NP. Hence, finding a solution to an NP problem is hard but verifying it can be done in polynomial time.

Q 14. What is Np-complete?

(PTU, Dec. 2017, 2016, 2015, 2006 ; May 2018, 2017, 2016, 2014, 2009, 2006)

Ans. In computational complexity theory, the complexity class **NP-complete** (abbreviated ***NP-C*** or ***NPC)*** is a class of decision problems. A decision problem L is NP-complete if it is in the set of NP problems so that any given solution to the decision problem can be verified in polynomial time, and also in the set of NP-hard problems so that any NP problem can be converted into L by a transformation of the inputs in polynomial time.

A decision problem d is said to be NP-complete if :

1. It belongs to class NP.
2. Every problem in NP is polynomially reducible to D.

Q 15. What is a NP-Hard problem? (PTU, Dec. 2015 ; May 2017, 2014, 2013, 2008)

Ans. NP-hard : In spite of its name, to say that problem is NP-hard does not mean that it is hard to solve. Rather it means that if we could solve this problem in polynomial time, then we could solve all NP problems in polynomial time. Note that for a problem to be NP hard, it does not have to be in the class NP. Since, it is widely believed that all NP problems are not solvable in polynomial time, it is widely believed that no NP-hard problem is solvable in polynomial time.

A notion of an NP-hard problem can be defined more formally by extending the notion of polynomial reducability to problems that are not necessary in class NP including optimization problems.

Q 16. What are P and NP problems? (PTU, May 2012)

Ans. Computer scientists use the Big O notation to describe concisely the running time of an algorithm. If we say that the running time of an algorithm is quadratic. Or O (n^2), it means that the running time of an algorithm on an input of size n is limited by a quadratic function of n.

Q 17. What are NP-complete algorithms? (PTU, May 2012)

Ans. The polynomial time algorithms are used to solve the NP-complete problem.

It is believe that NP-complete problems do not have polynomial time algorithms and therefore are intractable. Secondly, if any single NP-complete problem can be solved in polynomial time, then every NP-complete problem has a polynomial time algorithm.

Q 18. What are NP, NP Hard and NP complete problems? Explain by giving an example of each. (PTU, Dec. 2007)

Ans. Briefly, NP stands for "Non-deterministic polynomial time" and it marks a class of problems that can be solved in polynomial time on a non-deterministic turing machine. The key idea to understand is the "non-deterministic" which means that it is the case when more paths of solution could be chosen and at least one path solves it in a polynomial time.

The problem is how to find the right solution path and whether we can find the deterministic way that could also solve it in polynomial time -- marked as P.

In other words, there is a bunch of problems that could be solved by non-deterministic machine in polynomial time. Apparently, all problems that could be solved in polynomial time on deterministic machines also belongs to NP problems (they are subset of them, because non-deterministic machine is somehow more capable).

For some problems, for which no deterministic solution was found that would find the solution in polynomial time. These problems are marked NP-complete. It is believed (but not proved) that it is not possible to find them, hence the intersection between P and NP-complete is empty.

See also P = NP problem

NP-hard are "at least as hard as the hardest problems in NP". The difference is that the non-deterministic Turing machine uses some special thing called "oracle" that helps to make some decisions in constant time (i.e. clearly very artificial thing that is used only to study some decision problem).

Q 19. Differentiate between NP-hard and NP-complete problems with example. (PTU, Dec. 2019, 2011, 2009, 2008 ; May 2015, 2007)

Ans. Formally, **NP-complete** is a notion for so called recognition (or decision) problems, i.e., problems defined by a question for which the only two possible answers are a YES or a NO. It is defined with respect to polynomial reductions. From Nemhauser and Wolsey "X NP is said to be NP-complete if all problems in NP can be polynomially reduced to X."

NP-hard problems are usually optimization problems whose recognition version is NP-complete. For example the TSP-optimization is NP-hard because its TSP-recognition version is NP-complete (TSP-rec is as follows : given k is there a tour of length $\leq k$?). Nemhauser and Wolsey say "A problem is NP-hard if there is an NP-complete problem that can be polynomially reduced to it." Thus if a problem is NP-hard it is at least as difficult as any NP-complete problem.

The notion of an NP-hard problem can be defined more formally by extending the notion of polynomial reducability to problems that are not necessary in class NP including optimization problems.

Q 20. Compare NP hard and NP complete problems by taking examples of both. (PTU, May 2010)

Ans. NP-complete : A problem that is NP-complete can be solved in polynomial time iff all other NP-complete problems can also be solved in polynomial time.

All NP-complete problems are NP-hard but some NP-hard problems are known not to be NP-complete.

NP-complete $\subset$ NP-hard

Q 21. Give an example of NP-complete problem. (PTU, May 2011 ; Dec. 2007)

Ans. An interesting example is the graph isomorphism problem, the graph theory problem of determining whether a graph isomorphism exists between two graphs. Two graphs are isomorphism if one can be transformed into the other simply by renaming vertices. Consider these two problems :

- **Graph Isomorphism :** Is graph G_1 isomorphic to graph G_2?
- **Subgraph Isomorphism :** Is graph G_1 isomorphic to a subgraph of graph G_2?

The Subgraph Isomorphism problem is NP-complete. The graph isomorphism problem is suspected to be neither in P nor NP-complete, though it is in NP. This is an example of a problem that is thought to be hard, but isn't thought to be NP-complete. The easiest way to prove that some new problem in NP-complete is first to prove that it is in NP, and then to reduce some known NP-complete problem to it. Therefore, it is useful to know a variety of NP-complete problems. The list below containts some well-known problems that are NP-complete when expressed as decision problems.

- Boolean satisfiability problem (Sat.)
- N-puzzle
- Knapsack problem
- Hamiltonian path problem
- Travelling salesman problem
- Subgraph isomorphism problem
- Subset sum problem
- Clique problem
- Vertex cover problem
- Independent set problem
- Dominating set problem
- Graph coloring problem

To the right is a diagram of some of the problems and the reductions typically used to prove their NP-completeness. In this diagram, an arrow from one problem to another indicates the direction of the reduction. Note that this diagram is misleading as a description of the mathematical relationship between these problems, as there exists a polynomial-time reduction between any two NP-complete problems ; but it indicates where demonstrating this polynomial-time reduction has been easiest.

Q 22. Explain in detail basic concepts of P, NP, NP-hard and NP-complete problems. **(PTU, May 2013, 2012 ; Dec. 2010, 2009)**

Ans. A decision problem is in P if there is a known polynomial-time algorithm to get that answer. A decision problem is in NP if there is a known polynomial-time algorithm for a non-deterministic machine to get the answer.

Problems known to be in P are trivially in NP – the non-determistic machine just never troubles itself to fork another process, and acts just like a deterministic one. There are problems that are known to be neither in P nor PN, a simple example is to enumerate all the bit vectors of length n. No matter what, that takes 2^n steps. (Strictly, a decision problem is in NP if a non-deterministic machine can arrive at an answer in poly-time and a deterministic machine can verify that the solution is correct in poly time.)

But there are some problems which are known to be in NP for which no poly-time determistic algorithm is known ; in other words, we know they're in NP, but don't know if they'

re in P. The traditional example is the decision-problem version of the Traveling Salesman Problem (decision-TSP).

NP-hard : If an NP-hard problem can be solved in polynomial time then all NP-complete problems can also be solved in polynomial time.

All NP-complete problems are NP-hard but some NP-hard problems are known not to be NP-complete.

NP-complete : In computational complexity theory, the complexity class **NP-complete** (abbreviated **NP-C** or **NPC)** is a class of decision problems. A decision problem L is NP-complete if it is in the set of NP problems so that any given solution to the decision problem can be verified in polynomial time, and also in the set of NP-hard problems so that any NP problem can be converted into L by a transformation of the inputs in polynomial time.

Q 23. Describe how polynomial-time reductions are used to prove that a problem is NP-complete. **(PTU, May 2016)**

Ans. Polynomial time reductions are used to prove that a problem in NP complete for a given problem U, the steps involved in proving that it is NP complete are mentioned below :

1. Show that U is NP.
2. Select a known NP-complete problem V
3. Construct a reduction from V to U.
4. Show that the reduction requires Polynomial time.

These steps show that polynomial time reduction is important to prove a problem is NP complete. If we can find a polynomial time algorithm for satisfiability, then all other problems in NP can be solved in polynomial time. To prove that all other problems reduce to the given problem, that is a candidate problem is to be tested for NP completeness, is an involved process. An alternative is to show that some other know NP complete problem reduce to the new problem to be characterized. The other NP-Complete prob that are quite hardy to prove NP completeness of many other problems include 3-SAT, 3-dimensional match up, vertex cover, clique etc.When constructing a polynomial time reduction from 3-SAT to a prob, we look for structures in the problem that can simulate the variable and clauses in Boolean formula.

Q 24. What is closest pair problem ? **(PTU, Dec. 2019)**

Ans. The closest pair problem is a problem of computational geometry given n points in metric space, find a pair of points with the smallest distance between them.

Q 25. What is a 3SAT problem ? **(PTU, Dec. 2019)**

Ans. 3SAT or the boolean satisfiability problem, is a problem that asks what is the fastest algorithm to tell for a given formula in boolean algebra whether it is satisfiable that is whether there is some combination of values of the variables that will give.

❑❑❑

Chapter 5

Advanced Topics

Contents

Approximation algorithms, Randomized algorithms, Heuristics and their characteristics.

POINTS TO REMEMBER

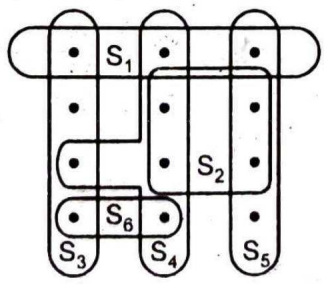
1. **Approximating algorithms :** An approximate algorithm is a way of dealing with NP-completeness for optimization problem.
2. **Heuristies :** Heuristic are a flexible technique for quick decisions, particularly when working with complex data.
3. **Randoized algorithms :** Randomized algorithms uses random number to decide, what to do next anywhere in its logic.

QUESTION-ANSWERS

Q 1. Define Set cover approximation problem.

Ans. The Set-cover problem : A finite set X and a collection of its subsets F such that $\bigcup_{S \in F} S = X$.

The set cover optimization problem is to find a minimum set $C \subseteq F$ that covers X.




```
GREEDY-SET-COVER(X, F) ;
    U : = X
    C : = ∅
    while U ≠ ∅
    do Choose S∈F with |S ∩ U| → max
       U : = U – S
       C : = C∪{S}
       return C
```

Since the while-loop is executed at most min {1×1, |F|} times and each its iteration requires 0 (1×1. |F|) computations, the running time of GREEDY-SET-Cover is 0 (1×1. |F|.min {1 × 1, |F|}).

Q 2. Explain vertex cover approximation algorithm in detail. (PTU, May 2014)

Ans. The vertex cover approximation algorithm : Let G be an undirected graph. A vertex cover of G is a subset COVER of V such that for every (u,v) ∈ E, at least one of u or V∈COVER.

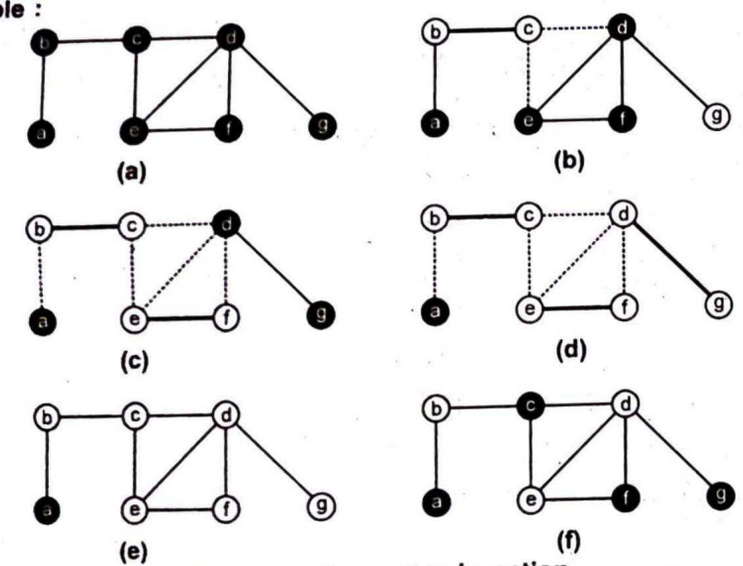
The vertex cover optimization problem is to find a vertex cover of minimum size = optimal vertex cover.

While finding the optimal vertex cover is an NPC problem, there is a p-time approximation algorithm that returns a near-optimal vertex cover.

APPROX-VERTEX COVER (G)

1. C = ∅
2. E' = E(G)
3. While E' ≠ ∅
4. Let (u, v) be an arbitrary edge of E'
5. C = C∪{u,v}
6. Remove from E' every edge incident on either u or v
7. Return C

Example :



Approx-vertex-cover in action

Approx-vertex-cover is a p-time 2-approximation algorithm :

Proof : The algorithm runs in O(V + E time)

The algorithm clearly returns a vertex cover, since it loops until all edges have been noved. Each edge removed in line 6 was covered by some vertex of an edge removed in 4.

Let A denote the set of edges removed in line 4 :

- The optimal cover C* must contain at least one end point of each of the edges in A.
- No two edges in A share an end point because all edges that share an endpoint with an edge in A are removed in line 6.

Since no two edges in A are covered by the same vertex in C*, we have that

$$|C^*| \geq |A| \quad \text{....(i)}$$

is a lower bound on the size of C*.

Each execution of line 4 picks an edge, both of whose endpoints are added to the approximate cover C. This gives us an upper bound on the size of C.

$$|C| = 2\,|A| \quad \text{....(ii)}$$

Combining equations (1) and (2), we get

$$|C| \leq 2\,|C^*|$$

Q 3. Prove that

The GREEDY-SET-COVER is a polynomial time p(n) – approximation algorithm,

where p(n) = H(Max {|S| |S∈F}) and H (d) = $\sum_{i=1}^{d}\left(\frac{1}{i}\right)$.

Ans. Proof : Let C be the set cover constructed by the GREEDY-SET-COVER algorithm and let C* be a minimum cover.

Let Si be the set chosen at the i-th execution of the while-loop. Furthermore, let x ∈ X be covered for the first time by Si. We set the weight Cx of x as follows :

$$Cx = \frac{1}{\left|S_i - (S_1 \cup \ldots\ldots \cup S_i - 1)\right|}$$

One has :

$$|C| = \sum_{x \in X} Cx \leq \sum_{S \in C^*} \sum_{x \in S} Cx \quad \text{....(1)}$$

We will show later that for any S∈F

$$\sum_{x \in S} Cx \leq H\,(|S|) \quad \text{....(2)}$$

from (1) and (2) one gets :

$$|C| \leq \sum_{S \in C^*} H\left(|S|\right) \leq |C^*|.\,H\left(\max\left\{|S| \mid S \in F\right\}\right),$$

Which completes the proof of the theorem.

To show (2) we define for a fixed S ⊆ F and i ≤ |C|

$$u_i = |S - (S_1 U \ldots\ldots USi)|,$$

that is, number of elements of S which are not covered by S1, ..., Si.

Let $u_0 = |S|$ and k be the minimum index such that $u_K = 0$. Then $u_{i-1} \geq u_i$ and $u_{i-1} - u_i$, elements of S are covered for the first time by Si for i = 1, ..., K.

One has :

$$\sum_{x \in S} Cx = \sum_{i=1}^{K} (u_{i-1} - u_i) \cdot \frac{1}{|S_i - (S_1 \cup ..... \cup S_i - 1)|}$$

Since for any $S \in F \setminus \{S1, ..., Si-1\}$

$|S_i - (S_1 \cup ..... \cup Si - 1)| \geq |S - (S_1 \cup .... \cup Si - 1)| = u_{i-1}$

due to the greedy choice of Si, we get :

$$\sum_{x \in S} C_x \leq \sum_{i=1}^{K} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}}$$

Since for any integers a, b with a < b it holds :

$$H(b) - H(a) = \sum_{i=a+1}^{b} \left(\frac{1}{i}\right) \geq (b - a) \cdot \left(\frac{1}{b}\right),$$

We get a telescopic sum :

$$\sum_{x \in S} C_x \leq \sum_{i=1}^{K} (H(u_i - 1) - H(u_i))$$

$$= H(u_0) - H(u_K) = H(u_0) - H(0)$$
$$= H(u_0) = H(|S|)$$

which completes the proof of (2).

Since, $H(d) \leq \ln d + 1$, the GREEDY-SET-COVER algorithm has the approximation rate $(\ln |x| + 1)$.

Q 4. Prove that the APPROX-TSP is a polynomial-time 2-approx. algorithm for the TSP problem with the triangle inequality.

Ans. Let H* be an optimal Hamilton cycle. We construct a cycle H with $C(H) \leq 2.C(H^*)$.

Since T is a minimal spanning tree, one has :

$$C(T) \leq C(H^*)$$

We construct a list L of vertices taken in the same order as in the MST-PRIM algorithm and get a walk W around T. Since W goes through every edge twice, we get :

$$C(W) = 2.C(T),$$

Which implies

$$C(W) \leq 2.C(H^*).$$

The walk W is, however, not Hamiltonian.

We go through the list L and delete from W the vertices which have already been visited.

This way we obtain a Hamilton cycle H. The triangle inequality provides

$$C(H) \leq C(W)$$

Therefore, $C(H) \leq 2.C(H^*)$.

Q 5. Explain TSP approximation problem.

Ans. The TSP Problem : A complete graph G = (V, E) and a weight function

$$C : E \to R \geq 0.$$

The TSP approximation problem is to find a Hamiltonian cycle in G of minimum weight.

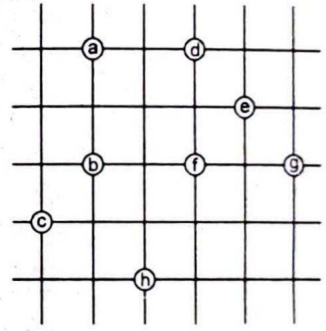
For $A \subseteq E$ define

$$C(A) = \sum_{(u,v) \in A} C(u, v)$$

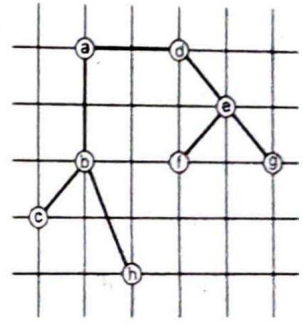
We assume the weights satisfy the triangle inequality for all $u,v;w \in V$.

APPROX-TSP (G, C) :

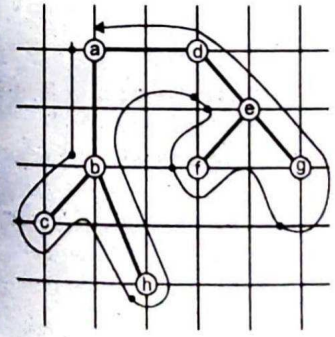
1. Choose a vertex $v \in V$.
2. Construct a minimum spanning tree T for G rooted in V (use, e.g. MST – PRIM algorithm).
3. Construct the pre-order traversal W of T.
4. Construct a Hamilton cycle that visits the vertices in order W.



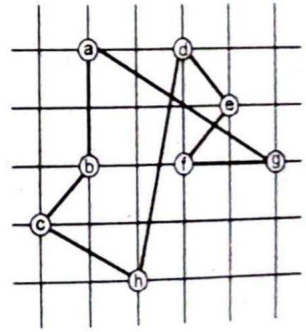
(a)



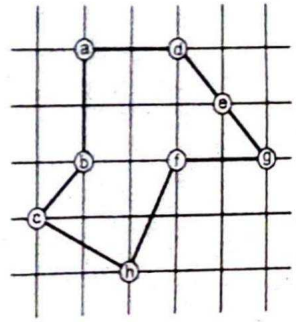
(b)



(c)



(d)



(e)

Q 6. Explain independent-set problem.

Ans. The independent-set problem : An undirected graph G = (V, E). The independent set problem is to find a maximum independent set.

For $v \in V$ and $n = |V|$ define $\delta = \frac{1}{n}\sum_{v \in V} \deg(v)$ and

$$N(V) = \{u \in V \mid dist(u,v) = 1\}.$$

INDEPENDENT-SET(G) ;

$S := \varnothing$

While $V(G) \neq \varnothing$ do

Find $v \in V$ with $deg(v) = \min u \in V$ deg(u)

$S := S \cup \{V\}$

$G := G - (v \cup N(v))$

return S.

The independent-set algorithm computes an independent set S of size $q \geq n/(\delta + 1)$.

Let v_i be the vertex chosen at step i and let $d_i = deg(v_i)$.

One has :

$\sum_{i=1}^{q} (d_i + 1) = n$. Since at step i we delete di + 1 vertices of degree at least di each, for the sum of degrees Si of the deleted vertices one has $Si \geq di(di + 1)$. Therefore,

$$\delta n = \sum_{v \in V} deg(v) \geq \sum_{i=1}^{q} S_i \geq \sum_{i=1}^{q} d_i (d_i + 1) .$$

This implies

$$\delta n + n \geq \sum_{i=1}^{q} (d_i(d_i + 1) + (d_i + 1)) = \sum_{i=1}^{q} (d_i + 1)^2 \geq \frac{n^2}{q}$$

$$\Rightarrow \qquad q \geq \frac{n}{\delta + 1}.$$

Q 7. Briefly explain maximum set-cover problem.

Ans. The maximum-set-cover-problem : A finite set X, a weight function $w : X \to R$, a collection F of subsets of X and $K \in IN$.

The maximum set cover problem is to find a collection $C \subseteq F$ of subsets with $|C| = K$ such that $\sum_{x \in C} w(x)$ is maximum

MAXIMUM-COVER(X, F, W) :

$U := X$

$C := \varnothing$

for i : = 1 to K do

Choose $S \in F$ with $W(S \cap \cup) \to max$

$U := U - S$

$C := C \cup S$

return C.

Q 8. Write note on the Approximating algorithms.

(PTU, May 2019, 2017 ; Dec. 2018, 2017, 2016, 2013, 2005)

Ans. Approximating Algorithms : An approximate algorithm is a way of dealing with NP-completeness for optimization problem. This technique does not guarantee the best solution. The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at most polynomial time.

Suppose we have some optimization problem instance i, which has a large number of feasible solutions. Also let c (i) be the cost of solution produced by approximate algorithm and c^* (i) be the cost of optimal solution. For minimization problem, we are interested in finding a solution of a given instance i in the set of feasible solutions, such that c (i) / c^* (i) be as small as possible. On the other hand, for maximization problem, we are interested in finding a solution in the feasible solution set such that $c^*(i)/c(i)$ be as small as possible.

We say that an approximation algorithm for the given problem instance i, has a ratio bound of p (n) if for any input of sign n, the cost c of the solution produced by the approximation algorithm is within a factor of p (n) of the cost c^* of an optimal solution.

That is

$$\max (c(i) / c^*(i) / c(i)) \leq p(n)$$

This definition applies for both minimization and maximization problems.

Note that p (n) is always greater than or equal to 1. If solution produced by approximation algorithm is true optimal solution then clearly we have p (n) = 1.

For a minimization problem, $0 < c^*(i) \leq c(i)$, and hte ratio c (i) / c^* (i) gives the factor by which the cost of the approximate solution is larger than the cost of an optimal solution. Similarly, for a maximization problem, $0 < c(i) \leq c^*(i)$, and the ratio c^* (i) / c (i) gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution.

Relative Error : We define the relative error of the approximate algorithm for any input size as

$$\text{mod } [c(i) - c^*(i) / c^*(i)]$$

We say that an approximate algorithm has a relative bound of ε (n) if

$$\text{mod } [c(i)\ c^*(i) / c^*(i)] \leq \varepsilon(n)$$

Q 9. What are approximation algorithms? Define absolute approximation and E-approximation with example. **(PTU, May 2008)**

OR

What is the importance of approximation algorithms? Also explain the various types of approximation algorithms. **(PTU, May 2011)**

Ans. Approximation algorithms are algorithms used to find approximate solutions to

optimization problems. Approximation algorithms are often associated with NP-hard problems ; since it is unlikely that there can never be efficient polynomial time exact algorithms solving NP-hard problems, one settles for polynomial time sub-optimal solutions. Unlike heuristics, which usually only find reasonably good solutions reasonably fast, one wants provable solution quality and provable run time bounds. Ideally, the approximation is optimal up to a small constant factor (for instance within 5% of the optimal solution). Approximation algorithms are increasingly being used for problems where exact polynomial-time algorithms are known but are too expensive due to the input size.

A typical example for an approximation algorithm is the one for vertex cover in graphs: find an uncovered edge and add both end points to the vertex cover, until none remain. It is clear that the resulting cover is at most twice as large as the optimal one. This is a constant factor approximation algorithm with a factor of 2.

Definition 1. A is an absolute approximation algorithm if and only if for every instance I of problem P, $|C^*(I) - C(I)| \le k$ for some constant k.

Approximation ratio p (n)

- Given input to size n
- C (I) is within a factor p (N) of C^* (I) if

$$\max\left(\frac{\hat{C}(I)}{C^*(I)}, \frac{C^*(I)}{\hat{C}(I)}\right) \le p(n)$$

- Another measure of approximation is given in literature as

$$\frac{\left|C^*(I) - \hat{C}(I)\right|}{C^*(I)} \le p(n)$$

Definition 2. An ϵ-aproximation algorithm is a p (n) approximation algorithm for which $p(n) \le \epsilon$ for some constant ϵ

- 1-approximation implies $\hat{C}(I) = C^*(I)$, resulting in an optional solution
- An approximation algorithm with a large p (n) may return a solution that is far worse than optimal

Approximation scheme

- Tradeoff between computation time and quality of approximation
- An algorithm may achieve increasingly smaller p (n) using more and more computations time
- Approximation algorithm takes a value $\epsilon > 0$ as an additional input such that for any fixed ϵ, the scheme is a $(1 + \epsilon)$ approximation algorithm.

Q 10. What are approximation algorithms? Explain approximation vertex cover. (PTU, May 2009)

Ans. Approximation algorithms are algorithms used to find approximate solutions to optimization problems. Approximation algorithms are often associated with NP-hard problems;

since it is unlikely that there can never be efficient polynomial time exact algorithms solving NP-hard problems, one settles for polynomial time sub-optimal solutions. Unlike heuristics, which usually only find reasonably good solutions reasonably fast, one wants provable solution quality and provable run time bounds. Ideally, the approximation is optimal up to a small constant factor (for instance within 5% of the optimal solution). Approximation algorithms are increasingly being used for problems where exact polynomial-time algorithms are known but are too expensive due to the input size.

A typical example for an approximation algorithm is the one for vertex cover in graphs: find an uncovered edge and add both end points to the vertex cover, until none remain. It is clear that the resulting cover is at most twice as large as the optimal one. This is a constant factor approximation algorithm with a factor of 2.

NP-hard problems vary greatly in their approximability ; some, such as the bin packing problem, can be approximated within any factor greater than 1 (such a family of approximation algorithms is often called a polynomial time approximation scheme or PTAS). Others are impossible to approximate within any constant, or even polynomial factor unless P = NP, such as the maximum clique problem.

NP-hard problems can often be expressed as integer programs (IP) and solved exactly in exponential time. Many approximation algorithms emerge from the linear programming relaxation of the integer program.

Vertex Cover : The minimum vertex cover problem on a graph asks for as small a set of vertices as possible that between them contain at least one end point of every edge in the graph. It is known that vertex cover is NP-hard, so we can't really hope to find a polynomial-time algorithm for solving the problem exactly. Instead, here is a simple 2-approximation algorithm :

```
Approximate Vertex Cover :
    while there are unmarked edge
    choose an unmarked edge
    mark both its endpoints
```

To show that this gives a 2-approximation, consider the set E' of all edges the algorithm chooses. None of these edges share a vertex, so any vertex cover must include at least |E'| vertices. The algorithm marks 2 |E'| vertices.

Q 11. What are heuristics ?

Ans. Heuristics are a problem solving method that uses shortents to produce good enough solutions given a limited time frame or deadline. Hecuristics are a flexible technique for quick decisions, particularly when working with complex data. Decisions made using an heuristic approach may not necessarily be optimal. Heuristic is derived from the Greek word meaning 'to discover'.

Heuristics facilitate timely decisions. Analysts in every industry use rules of thumb such as intelligent guesswork, trial and error, process of elimination, past formulas and the analyis of historical data to solve a problem. Heuristics methods make decision making simpler and faster through short cuts and good enough calculations.

Q 12. What is randomized algorithm ?

Ans. An algorithm that uses random number to decide what to do next anywhere in its logic is called Randomized Algorithm. For eg, in Randomized quick sort, We use random number to pick the next pivot.

Q 13. How you can analyse Randomized algorithms ?

Ans. Some randomized algorithms have deterministic time complexity. For example, this implementation of Karges algorithm has time complexity as O(e). Such algorithms are called Monte Carlo Algorithms and are easier to analyse for worst case. On the other hand, time complexity of other randomized algorithms is dependent on value of random variable. Such Randomized algorithm are called Las Vegas Algorithm. These algorithm are typically analysed for expected worst case. To compute expected time taken in worst case, all possible values of the used random variable needs to be considered in worst case and time taken by every possible values needs to be evaluated. Average of all evaluated times is the expected worst case time complexity.

Q 14. What are the advantages of randomized algorithm ?

Ans. 1. The algorithm is usually simple and easy to implement

2. The algorithm is fast with very high probability and/or it produces optimum output with very high probability.

Q 15. Explain Bin packing problem with the help of an example.

Ans. Given n items of different weights and bins each of capacity c, assign each item to a bin such that number to total used bins is minimized. It may be assumed that all items have weights smaller than bin capacity.

Input : Weight [ ] = {4, 8, 1, 4, 2, 1}
Bin capacity C = 10

Output : 2

We need minimum 2 bins to accommodate all items First bin Contains {4, 4, 2} and second bin {8, 2}

Input : Weight [ ] = {9, 8, 2, 2, 5, 4}
Bin capacity C = 10

Output : 4

We need minimum 4 bins to accomodate all items.

Lower bound : We can always find a lower bound on minimum number of bins required. The lower bound can be given as :

Min no. of bins > = ceil ((Total Weight)/(Bin Capacity))

In the above examples, lower bound for first example is "ceil (4 + 8 + 1 + 4 + 2 + 1)/10" = 2 and lower bound in second example is" ceil (9 + 8 + 2 + 2 + 5 + 4)/10" = 3 The problem is a NP hard problem and finding an exact minimum number of bins takes exponential time.

These algorithm are for Bin packing problems where items arrive one at a time, each must be put in a bin, before considering the next item.

1. **Next fit :** When processing next item, check if it fits in the same bin as the last item.

Use a new bin only if it does not. Next fit is a simple algorithm. It requires only O(n) time and O(1) extra space to process n items. Next fit is 2 approximate i.e. the number of bins used by this algorithm is bounded by twice of optional. Consider any two adjacent bins. The sum of items in these two bins must be >c; otherwise, Next fit would have put all the items of second bin into the first. The same holds for all other bins. Thus, at most half the space is wasted and so next fit uses at most 2M bins if M is optimal.

2. **First fit :** When processing the next items, scan the previous bins in order and place the item in the first bit that fits. Start a new bin only if it does not fit in any of the existing bins.

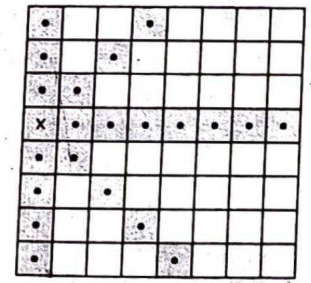
3. **Best fit :** The idea is to place the next item in the tightest spot. That is, put it in the bin so that smallest empty space is left. Best fit can also be implemented in O(n logn) time using self-balancing Binary search trees.

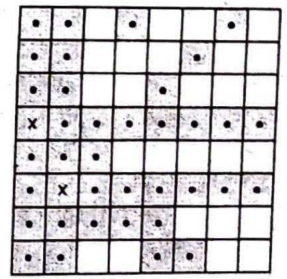
4. **First Fit Decreasing :** A trouble with online algorithm is that packing large items is difficult, especially if they occur late in the sequence. We can circumvent this by sorting the input sequence and placing the large items first. With sorting, we get first fit decreasing and Best Fit Decreasing as offline analogs of online first fit and best fit. First fit decreasing produces the best result for the sample input because items are sorted first. First fit decreasing can also be implemented in 0 (n log n) time using self Balancing Binary Search Trees.

Q 16. Explain Back tracking using No. Queens Puzzle.

Ans. Branch and Bound solution, after building a partial solution, we figure out that there is no point going any deeper as we are going to hit a dead end. In back tracking, solution we back track when we hit a dead end.

Back tracking : "The idea is to place queen one by one in different columns, starting from the left-most column. When we place a queen in a column, we check for clashes with already placed queen. In the current column, If we find a row for which then is no cash, we mark this row and column as part of the solution, If we do not find such a row due to clashes, then we backtrack and return false."





1. For the Ist Queen, there are total 8 possibilities we can place Ist Queen in any row of first column. Let's place Queen 1 on row 3.
2. After placing Ist Queen, there are 7 possibilities left for the 2nd Queen. But wait, we don't really have 7 possibilities. We cannot place Queen 7 on rows 2, 3 or 4 as those cells are under attack from Queen 1. So Queen 2 has only 8 – 3 = 5 valid positions left.

3. After picking a position for Queen 2, Queen 3 has even fewer options as most of the cells in its column are under attack from first 2 Queens. We need to figure out an efficient way of keeping track of which cells are under attack. Basically, we have to ensure 4 things :
   (i) No two queens share a column.
   (ii) No two queens share a row.
   (iii) No two queens share a top right to left bottom diagonal.
   (iv) No two queens share a top left to bottom right diagonal.

Number 1 is automatic because of the way we store the solution. For number 2, 3 and 4 we can perform, updates in 0(1) time. The idea is to keep three Boolean arrays that tell us which row and which diagonals are occupied.

Now for preprocessing we will create two N×N matrix one for/diagonal and other one for\diagonal. Let's call them slash code and back slash code respectively. The trick is to fill them in such a way that two queens sharing a same/diagonal will have the same value in matrix slash code, and it they share\diagonal, they will have the same value in backslash code matrix.

From N×N matrix, fill slash code and backslash code matrix using below formula :

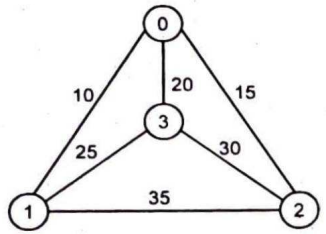
slash code [row] [col] = row + col

backslash code [row] [col] = row – col + (N – 1)

Now before we place queen i or row j, we first check whether row j is used. Then we check whether slash code (j + i) or backslash code (j – i + j) are used. If yes then we have to try a different location for queen. If not, then we mark the row and the two diagonals are used and recursion queen i + 1. After the recursive call return and before we try another position for queen i, we need to reset the row, slash code and back slash code as unused again, like in the code.

Q 17. Explain travelling saleshman problem using Branch and Bound.

Ans. Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point. For example, Consider the graph shown in figure on right side. ATSP tour in the graph is 0–1–3–2–0. The cost of the tour is 10+25+30+15 which is 80.



Branch and Bound solution : In branch and bound method, for current node in tree, we compute a bound on best possible solution that we can get it we down this node. If the bound on best possible solution itself is worse than current best, then we ignore the subtree rooted with the node. The cost through a node includes two costs.

1. Cost of reaching the node from the root.
2. Cost of reaching an answer from current node to a leaf.

❑ In cases of a maximization problem, an upper bounds tells us the maximum possible solution if we follow the given node.

❑ In cases of a minimization problem, a lower bound tells us the minimum possible solution if we follow the given mode.

In branch and bound, the challenging part is figuring out a way to compute a bound on best possible solution. Below is an idea based to compute bounds for Travelling salesman problem.

Cost of any tour can be written as below :

Cost of tour T = (1/2) * & sum; (Sum of cost of two edges adjacent to 4 and in the tour T)

Where u ∈ v

For every vertex 4, If we consider two edges through if n , and sum their costs, The overall sum for all vertices would be twice of cost of tour T

(Sum of two tour edges adjacent to u) = (Sum of minimum weight two edge adjacent to u) >

(Cost of any tour > = 1/2 * & sum; (sum of cost of two minimum weight edges adjacent to u)

Where u ∈ v.

Below are minimum cost two edges adjacent to every node for above shows graph.

Node	Least Cost edges	Total cost
0	(0,1), (0, 2)	25
1	(0, 1), (1,3)	35
2	(0,2), (2,3)	45
3	(0,3), (1,3)	45

Thus a lower bound on the cost of any tour =

1/2 (25 + 35 + 45 + 45) = 75

Lets start enumerating all possible nodes :

1. The Root Node : Without loss of generality, we assume we start at vertex "0" for which the lower bound has been calculated above.

Dealing with Level 2 : The next level enumerates all possible vertices, we can go to which are 1, 2, 3, N consider we are calculating the vertex 1, since we moved from 0 to 1, our tour has now included the edge 0–1. This allows us to make necessary changes in the lower bound of the root.

To include edge 0–1, we add the edge cost of 0–1, and subtract an edge weight such that the lowers bound remains as tight as possible which would be the sum of minimum edges of 0 and 1 divided by 2. Clealry, the edge subtracted can't be smaller than this.

Dealing with other levels : As we move on to the next level, we again enumerate all possible vertices, for the above case going further after 1, we check out for 2, 3, 4, ... n. Consider lower bound for 2 as we moved from 1 to 1, we include the edge 1–2 to the tour and alter the new lower bound for this node.

Lower bound (2) = Old lower bound – ((Second minimum edge cost of 1 + minimum edge cost of 2)/2) + edge cost (1–2)

Q 18. Why do we need approximation algorithms ? How do we characterize approximation algorithms ? **(PTU, Dec. 2015)**

Ans. Sometimes, approximate solutions close to the exact solution may be what we are interested. For example : the TSP travels 19,000 km or 19,200 km does not make much difference in practice. For NP - hard problems, an approximate solution may be all that one can expect to find with in a reasonable amount of computing time. We can characterize approximation algorithms based on the deviation of the result found by the approximation algorithm from the optimal solution. The deviation could be absolute or relative. Let A be an algorithm that generates a feasible solution to every instance I of a problem P. Let F0(I) be the value of an optimal solution to I and let FA(I) be the value of the feasible solution generated by A.

We can characterize a number of approximation algorithms as described below :

1. Absolute approximation : A is an absolute approximation algorithm for problem P if and only if for every instance I or P,

$|F0(I) - FA(I)| \leq k$ for some constant k.

2. f(n) - approximation : A is an f(n) - approximation algorithm if and only if for every instance I of size n,

$|F0(I) - FA(I)| / F0(I) \leq f(n)$

assuming that $F0(I) > 0$.

3. ε-approximation : A is an ε-approximation algorithm if and only if for every instance I of size n,

$|F0(I) - FA(I)| / F0(I) \leq \varepsilon$,

for some constant ε, F0(I) is assumed to be greater than zero.

❑❑❑