Few Pending

DAA MODULE 5

Click the link below to watch overview on Module 5

P, NP, NP Hard and NP Complete Problem | Reduction | NP Hard and NP C...

PART - A

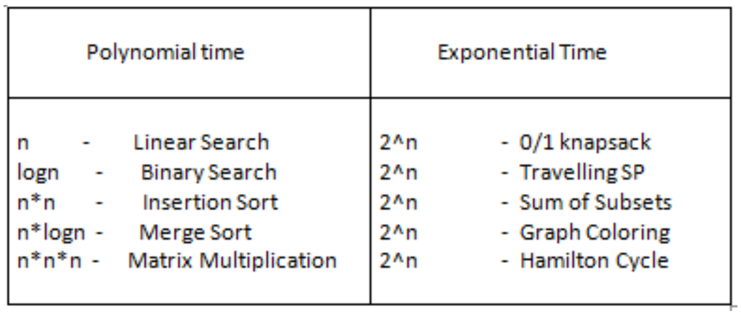
1. Show that satisfiability is at most three literals reduced to chromatic number.

Refer - 3-coloring is NP Complete - GeeksforGeeks

4. Explain two problems that have polynomial time algorithms. Justify your answer.

A polynomial-time algorithm is an algorithm whose execution time is either given by a polynomial on the size of the input, or can be bounded by such a polynomial. Problems that can be solved by a polynomial-time algorithm are called tractable problems.

For example, most algorithms on arrays can use the array size, n, as the input size. To find the largest element in an array requires a single pass through the array, so the algorithm for doing this is O(n), or linear time.



Polynomial time	Exponential Time
n - Linear Search logn - Binary Search n*n - Insertion Sort n*logn - Merge Sort n*n*n - Matrix Multiplication	2^n - 0/1 knapsack 2^n - Travelling SP 2^n - Sum of Subsets 2^n - Graph Coloring 2^n - Hamilton Cycle

write any polynomial time problem

5. Explain 3CNF satisfiability problems.

Concept: - In 3CNF SAT, you have at least 3 clauses, and in clauses, you will have almost 3 literals or constants

Such as (X+Y+Z) (X+Y+Z) (X+Y+Z)

You can define as (XvYvZ) ^ (XvYvZ) ^ (XvYvZ)

V=OR operator

^ =AND operator

These all the following points need to be considered in 3CNF SAT.

To prove: -

1. Concept of 3CNF SAT
2. SAT≤ρ 3CNF SAT
3. 3CNF≤ρ SAT
4. 3CNF ϵ NPC

1. CONCEPT: - In 3CNF SAT, you have at least 3 clauses, and in clauses, you will have almost 3 literals or constants.
2. SAT ≤ρ 3CNF SAT:- In which firstly you need to convert a Boolean function created in SAT into 3CNF either in POS or SOP form within the polynomial time

```
    F=X+YZ
         = (X+Y) (X+Z)
         = (X+Y+ZZ') (X+YY'+Z)
         = (X+Y+Z) (X+Y+Z') (X+Y+Z) (X+Y'+Z)
         = (X+Y+Z) (X+Y+Z') (X+Y'+Z)
```

3. 3CNF ≤p SAT: - From the Boolean Function having three literals we can reduce the whole function into a shorter one.

```
F= (X+Y+Z) (X+Y+Z') (X+Y'+Z)
     = (X+Y+Z) (X+Y+Z') (X+Y+Z) (X+Y'+Z)
     = (X+Y+ZZ') (X+YY'+Z)
     = (X+Y) (X+Z)
     = X+YZ
```

4. 3CNF ϵ NPC: - As you know very well, you can get the 3CNF through SAT and SAT through CIRCUIT SAT that comes from NP.

   Proof of NPC:-

1. It shows us that we can easily convert a Boolean function of SAT into 3CNF SAT and satisfy the concept of 3CNF SAT also within polynomial time through Reduction concept.
2. If we want to verify the output in 3CNF SAT then perform the Reduction and convert into SAT and CIRCUIT also to check the output

As we have achieved these two points that means 3CNF SAT also in NPC

6. Explain P type problems with examples.

Say that an algorithm A runs in polynomial time if there is a positive integer k so that,

A runs in time $O(n^k)$, where n is the length of x.

For example, an algorithm that takes no more than n^3 steps runs in polynomial time, or is a polynomial time algorithm.

If we accept, for the moment, that a polynomial time algorithm is "efficient", then it makes sense to give a name to decision problems that have efficient algorithms.

➔ P is the class of all decision problems (languages) that can be solved by polynomial time algorithms.

Notice that the definition has nothing to do with whether you or I are clever enough to find a polynomial time algorithm for the problem. A decision problem is in P if there exists a polynomial time algorithm for it.

Examples:

1. Recognizing palindromes:

The problem of recognizing palindromes is solvable in linear time, which is certainly polynomial time. A palindrome is a string that is equal to its own reversal. For example, aabcbaa is a palindrome. Let's define

```
PALINDROME =   {x | x ∈ {a, b, c}* and x is a palindrome}
```

It is easy to see that PALINDROME is in P. To decide if x is a palindrome, just reverse x and check whether the reversal of x is equal to x.

2. String matching

The string matching problem is as follows.

Input. Two strings, p and t.

Question. Does p occur as a contiguous substring of t?

For example if p = rocket and t = The rocketry expert is here., then the answer is yes.

One algorithm just tries each position in t to see if p occurs at that position. It keeps going until it finds a match or exhausts the positions that it can try.

The rocketry expert is here.

```
rocket

 rocket

  rocket

   rocket

    rocket
```

Suppose that p has length u and t has length v. There are v – u + 1 positions to try. It takes no more than u steps to check each position. So the total time is proportional to u(v – u + 1).

Since $n \approx u + v$, the cost is highest when v is about 2u, which means $u \approx n/3$. Then the time is about (n/3)(n/3), which is $O(n^2)$.

7. Explain in detail about approximation algorithms for NP hard problems.

Firstly let's discuss different approaches to handle difficult problems of combinatorial optimization, such as the travelling salesman problem and the knapsack problem.
The decision versions of these problems are NP-complete. Their optimization versions fall in the class of NP-hard problems. Theoretically there are no known polynomial-time algorithms for these problems.

There is a radically different way of dealing with difficult optimization problems: solve them approximately by a fast algorithm. This approach is particularly appealing for applications where a good but not necessarily optimal solution will suffice. Besides, in real-life applications, we often have to operate with inaccurate data to begin with. Under such circumstances, going for an approximate solution can be a particularly sensible choice.

Although approximation algorithms run a gamut in level of sophistication, most of them are based on some problem-specific heuristic. A heuristic is a common-sense rule drawn from experience rather than from a mathematically proved assertion.

We use an algorithm whose output is just an approximation of the actual optimal solution, we would like to know how accurate this approximation is. We can quantify the accuracy of an approximate solution s_a to a problem of minimising some function f by the size of the relative error of this approximation,

However, we cannot compute the accuracy ratio, because we typically do not know f (s∗), the true optimal value of the objective function. Therefore, our hope should lie in obtaining a good upper bound on the values of $r(s_a)$.
A polynomial-time approximation algorithm is said to be a c-approximation algorithm, where $c \geq 1$, if the accuracy ratio of the approximation it produces does not exceed c for any instance of the problem in question:

$$r(s_a) \leq c. \qquad \textbf{(12.3)}$$

The best (i.e., the smallest) value of c for which inequality (12.3) holds for all instances of the problem is called the performance ratio of the algorithm and denoted RA.

The performance ratio serves as the principal metric indicating the quality of the approximation algorithm. We would like to have approximation algorithms with RA as close to 1 as possible.

Note: Unfortunately, some approximation algorithms have infinitely large performance ratios (RA = ∞). This does not necessarily rule out using such algorithms, but it does call for a cautious treatment of their outputs.

8. Show that satisfiability of Boolean formula in 3 conjunctive normal form is NP - complete.

Concept: - In 3CNF SAT, you have at least 3 clauses, and in clauses, you will have almost 3 literals or constants

Such as (X+Y+Z) (X+Y+Z) (X+Y+Z)

You can define as (XvYvZ) ^ (XvYvZ) ^ (XvYvZ)

V=OR operator

^ =AND operator

These all the following points need to be considered in 3CNF SAT.

To prove: -

5. Concept of 3CNF SAT
6. SAT≤ρ 3CNF SAT
7. 3CNF≤ρ SAT
8. 3CNF ϵ NPC
9. CONCEPT: - In 3CNF SAT, you have at least 3 clauses, and in clauses, you will have almost 3 literals or constants.
10. SAT ≤ρ 3CNF SAT:- In which firstly you need to convert a Boolean function created in SAT into 3CNF either in POS or SOP form within the polynomial time

```
F=X+YZ
      = (X+Y) (X+Z)
      = (X+Y+ZZ') (X+YY'+Z)
      = (X+Y+Z) (X+Y+Z') (X+Y+Z) (X+Y'+Z)
      = (X+Y+Z) (X+Y+Z') (X+Y'+Z)
```

11. 3CNF ≤p SAT: - From the Boolean Function having three literals we can reduce the whole function into a shorter one.

```
F= (X+Y+Z) (X+Y+Z') (X+Y'+Z)
    = (X+Y+Z) (X+Y+Z') (X+Y+Z) (X+Y'+Z)
    = (X+Y+ZZ') (X+YY'+Z)
    = (X+Y) (X+Z)
    = X+YZ
```

12. 3CNF ϵ NPC: - As you know very well, you can get the 3CNF through SAT and SAT through CIRCUIT SAT that comes from NP.

    Proof of NPC:-

3. It shows us that we can easily convert a Boolean function of SAT into 3CNF SAT and satisfy the concept of 3CNF SAT also within polynomial time through Reduction concept.
4. If we want to verify the output in 3CNF SAT then perform the Reduction and convert into SAT and CIRCUIT also to check the output

As we have achieved these two points that means 3CNF SAT also in NPC

9. Show that the Clique Decision problem is NP - Complete.

To Prove: - Clique is an NPC or not?

For this we have to satisfy the following below-mentioned points: -

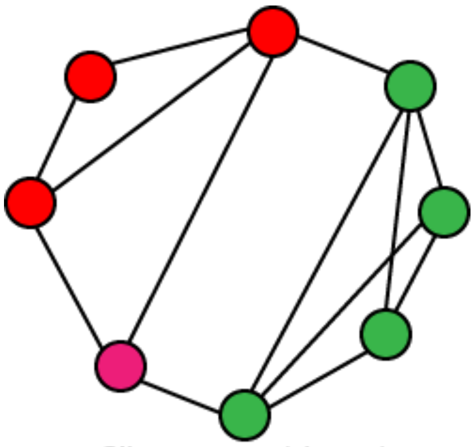
1. Clique
2. 3CNF ≤ρ Clique
3. Clique ≤ρ 3CNF≤SAT
4. Clique ϵ NP

1) Clique

Definition: - In Clique, every vertex is directly connected to another vertex, and the number of vertices in the Clique represents the Size of Clique.

CLIQUE COVER: - Given a graph G and an integer k, can we find k subsets of verticesV_1, V_2...V_K, such that UiVi = V, and that each Vi is a clique of G.

The following figure shows a graph that has a clique cover of size 3.



Clique cover (size=3)

2)3CNF ≤ρ Clique

Proof:-For the successful conversion from 3CNF to Clique, we have to follow the two steps:-

Draw the clause in the form of vertices, and each vertex represents the literals of the clauses.

1. They do not complement each other
2. They don't belong to the same clause
   In the conversion, the size of the Clique and size of 3CNF must be the same, and you successfully converted 3CNF into Clique within the polynomial time

3) Clique ≤ρ 3CNF≤SAT

Proof: - As we know that a function of K clause, there must exist a Clique of size k. It means that P variables which are from the different clauses can assign the same value (say it is 1). By using these values of all the variables of the CLIQUES, you can make the value of each clause in the function is equal to 1

Example: - You have a Boolean function in 3CNF:-

```
(X+Y+Z) (X+Y+Z') (X+Y'+Z)
```

After Reduction/Conversion from 3CNF to CLIQUE, you will get P variables such as: - x +y=1, x +z=1 and x=1

Put the value of P variables in equation (i)

(1+1+0)(1+0+0)(1+0+1)

(1)(1)(1)=1 output verified

4) Clique ϵ NP:-

Proof: - As you know very well, you can get the Clique through 3CNF and to convert the decision-based NP problem into 3CNF you have to first convert into SAT and SAT comes from NP.

So, I concluded that CLIQUE belongs to NP.

Proof of NPC:-

1. Reduction achieved within the polynomial time from 3CNF to Clique
2. And verified the output after Reduction from Clique To 3CNF above
   So, concluded that, if both Reduction and verification can be done within the polynomial time that means Clique also in NPC.

10. Show that the Chromatic number Decision problem is NP - Complete.

PART - B

Bhavani, Rishi and Ujjwal

1. Explain and prove Cook's theorem.

In computational complexity theory, the Cook–Levin theorem, also known as Cook’s theorem, states that the Boolean satisfiability problem is NP-complete. That is, it is in NP, and any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the Boolean satisfiability problem.

Stephen Arthur Cook and L.A. Levin in 1973 independently proved that the satisfiability problem(SAT) is NP-complete. He proved Circuit-SAT and 3CNF-SAT problems are as hard as SAT. Similarly, Leonid Levin independently worked on this problem in the then Soviet Union. The proof that SAT is NP-complete was obtained due to the efforts of these two scientists. Later, Karp reduced 21 optimization problems, such as Hamiltonian tour, vertex cover, and clique, to the SAT and proved that those problems are NP-complete.

Seems like some complex theorem. If interested, refer link below:
Design and Analysis Cook's Theorem

2. Compare deterministic and non deterministic algorithms.

Sr. No.	Key	Deterministic Algorithm	Non-deterministic Algorithm
1	Definition	The algorithms in which the result of every algorithm is uniquely defined are known as the	On other hand, the algorithms in which the result of every algorithm is not

Sr. No.	Key	Deterministic Algorithm	Non-deterministic Algorithm
		Deterministic Algorithm. In other words, we can say that the deterministic algorithm is the algorithm that performs a fixed number of steps and always gets finished with an accept or reject state with the same result.	uniquely defined and the result could be random are known as the Non-Deterministic Algorithm.
2	Execution	In Deterministic Algorithms execution, the target machine executes the same instruction and results the same outcome which is not dependent on the way or process in which the instruction gets executed.	On other hand in case of Non-Deterministic Algorithms, the machine executing each operation is allowed to choose any one of these outcomes subject to a determination condition to be defined later.
3	Type	On the basis of execution and outcome in case of Deterministic algorithm, they are also classified as reliable algorithms as for	On other hand Non deterministic algorithms are classified as non-reliable

Sr. No.	Key	Deterministic Algorithm	Non-deterministic Algorithm
		a particular input instruction the machine will always give the same output.	algorithms for a particular input the machine will give different output on different executions.
4	Execution Time	As outcome is known and is consistent on different executions so the Deterministic algorithm takes polynomial time for their execution.	On other hand as outcome is not known and is non-consistent on different executions so Non-Deterministic algorithm could not get executed in polynomial time.
5	Execution path	In deterministic algorithms the path of execution for an algorithm is the same in every execution.	On the other hand in case of a Non-Deterministic algorithm the path of execution is not the same for algorithm in every execution and could take any random path for its execution.

3. Explain non deterministic algorithms for sorting and searching.

In computer programming, a nondeterministic algorithm is an algorithm that, even for the same input, can exhibit different behaviours on different runs, as opposed to a deterministic algorithm. There are several ways an algorithm may behave differently from run to run. A concurrent algorithm can perform differently on different runs due to a race condition.
The design of ND Algorithms is based on three major functions:

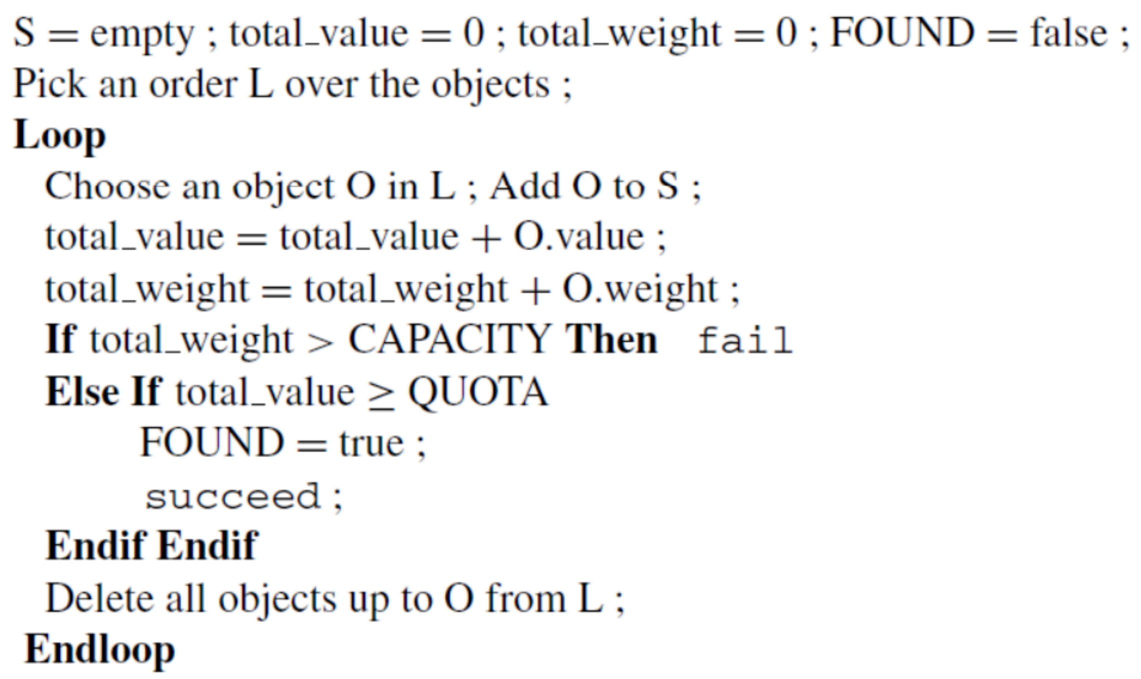
1. Select ()
2. Success ()
3. Failure ()

To declare the success or failure, a verification process should be designed.

```
Algorithm nd_search(a,n,x)
{
// "a" = array of size "n" and "x" is element to be searched
for i = 1 to n do
{
j = select(a,n) //select a location "j" from given array
if (a[j] = x)
success();
}
failure();
}
//Try above algorithm using repeat until and comment upon the time
complexity//
Algorithm nd_sort(a,b,n)
{
//"a" is array to be sorted and "b" is array for auxiliary storage
For i = 1 to n do
{
```

```
j = select(a,n)
b[i] = a[j] //create the array "b" by selecting "n" elements
}
for j = 1 to n do
{
if(b[i] > b[i+1])
failure();
}
success();
}
```

4. Explain non deterministic algorithm for sorting non deterministic knapsack algorithm.



```
S = empty ; total_value = 0 ; total_weight = 0 ; FOUND = false ;
Pick an order L over the objects ;
Loop
  Choose an object O in L ; Add O to S ;
  total_value = total_value + O.value ;
  total_weight = total_weight + O.weight ;
  If total_weight > CAPACITY Then  fail
  Else If total_value ≥ QUOTA
        FOUND = true ;
        succeed ;
  Endif Endif
  Delete all objects up to O from L ;
Endloop
```

5. explain how P and NP problems are related

P-Class

- The class P consists of those problems that are solvable in polynomial time, i.e. these problems can be solved in time O(n^k) in worst-case, where k is constant.
- These problems are called tractable, while others are called intractable or superpolynomial.
- Formally, an algorithm is polynomial time algorithm, if there exists a polynomial p(n) such that the algorithm can solve any instance of size n in a time O(p(n)).
- Problem requiring $\Omega(n^{50})$ time to solve are essentially intractable for large n. Most known polynomial time algorithm run in time $O(n^k)$ for fairly low value of k.
- The advantages in considering the class of polynomial-time algorithms is that all reasonable deterministic single processor model of computation can be simulated on each other with at most a polynomial slow-d.

NP-Class

- The class NP consists of those problems that are verifiable in polynomial time. NP is the class of decision problems for which it is easy to check the correctness of a claimed answer, with the aid of a little extra information.
- Hence, we aren't asking for a way to find a solution, but only to verify that an alleged solution really is correct.
- Every problem in this class can be solved in exponential time using exhaustive search.

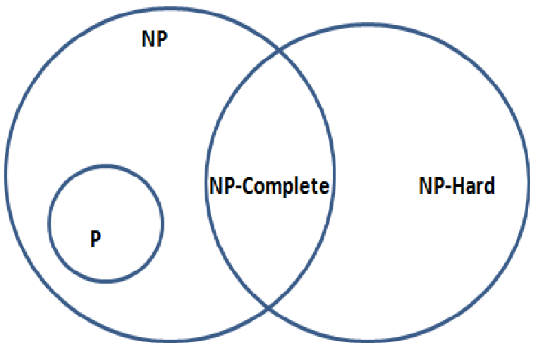
P versus NP

Every decision problem that is solvable by a deterministic polynomial time algorithm is also solvable by a polynomial time non-deterministic algorithm.

All problems in P can be solved with polynomial time algorithms, whereas all problems in NP - P are intractable.

It is not known whether P = NP. However, many problems are known in NP with the property that if they belong to P, then it can be proved that P = NP.

If P ≠ NP, there are problems in NP that are neither in P nor in NP-Complete. The problem belongs to class P if it's easy to find a solution for the problem. The problem belongs to NP, if it's easy to check a solution that may have been very tedious to find.



6. compare NP -hard and NP -complete problems.

NP Problem:

The NP problems set of problems whose solutions are hard to find but easy to verify and are solved by Non-Deterministic Machine in polynomial time.

NP-Hard Problem:

A Problem X is NP-Hard if there is an NP-Complete problem Y, such that Y is reducible to X in polynomial time. NP-Hard problems are as hard as NP-Complete problems. NP-Hard Problem need not be in NP class.

NP-Complete Problem:

A problem X is NP-Complete if there is an NP problem Y, such that Y is reducible to X in polynomial time. NP-Complete problems are as hard as NP problems. A problem is NP-Complete if it is a part of both NP and NP-Hard Problem. A non-deterministic Turing machine can solve NP-Complete problem in polynomial time.

NP-hard	**NP-Complete**
NP-Hard problems(say X) can be solved if and only if there is a NP-Complete problem(say Y) that can be reducible into X in polynomial time.	NP-Complete problems can be solved by a non-deterministic Algorithm/Turing Machine in polynomial time.
To solve this problem, it does not have to be in NP .	To solve this problem, it must be both NP and NP-hard problems.
Do not have to be a Decision problem.	It is exclusively a Decision problem.
Example: Halting problem, Vertex cover problem, etc.	Example: Determine whether a graph has a Hamiltonian cycle, Determine whether a Boolean formula is satisfiable or not, Circuit-satisfiability problem, etc.

7. Explain clique decision problems with an example.

In the field of computer science, the clique decision problem is a kind of computation problem for finding the cliques or the subsets of the vertices which when all of them are adjacent to each other are also called complete subgraphs.
A clique is a subgraph of a graph such that all the vertices in this subgraph are connected with each other, that is the subgraph is a complete graph. The Maximal Clique Problem is to find the maximum sized clique of a given graph G, that is a complete graph which is a subgraph of G and contains the maximum number of vertices. This is an optimization problem. Correspondingly, the Clique Decision Problem is to find if a clique of size k exists in the given graph

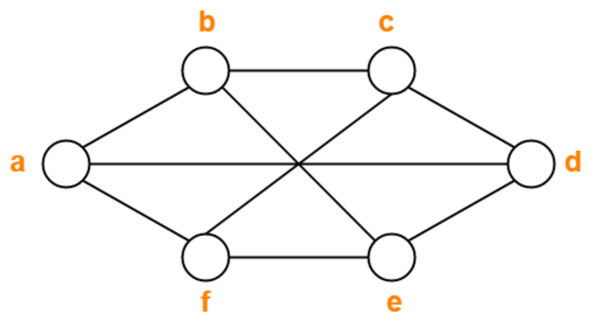
or not. The clique decision problem has many formulations based on which the cliques and about the cliques the information should be found. There are some common formulations based on which the cliques are based such as finding the maximum clique, finding the maximum weight of the clique in a weighted graph, then listing all the maximum or maximal cliques, and finally solving the problem based on the decision of testing whether the graph has the larger cliques than that of the given size.
Maximum clique: a particular clique that has the largest possible number of vertices.
Maximal cliques: the cliques which further cannot be enlarged.

```
When S = NULL.
for i = 1 till k then start do-while loop.
t : = ch(1 to n)
if t belongs to S then
return fail.
When S:= S union t
Then for all pairs of I and j such that when i belongs to S and j belongs to S  and if i not equal to j then start do-while loop. And check if I and j is not an edge of that given graph then
Return fail.
Else return a true value.
```

Conclusion :
Thus, a clique is actually a complete total subgraph of a given particular graph and the max clique association problem is basically a computational problem for finding the maximum clique of the graph.

8. Explain chromatic number decision problem and clique decision problem.

Chromatic Number is the minimum number of colors required to properly color any graph.
Consider the below example.



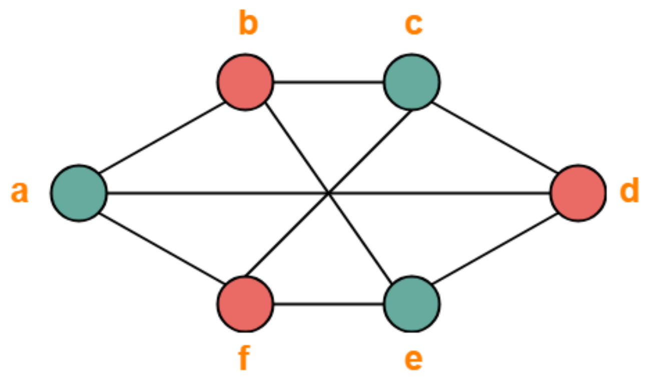
Applying Greedy Algorithm, we have-

Vertex	a	b	c	d	e	f
Color	C1	C2	C1	C2	C1	C2

From here,

- Minimum number of colors used to color the given graph are 2.
- Therefore, Chromatic Number of the given graph = 2.

The given graph may be properly colored using 2 colors as shown below-



Refer 7th Question

9. Explain the strategy to prove that a problem is NP - hard.

Reductions and SAT

To prove that any problem other than circuit satisfiability is NP-hard, we use a reduction argument. Reducing problem A to another problem B means

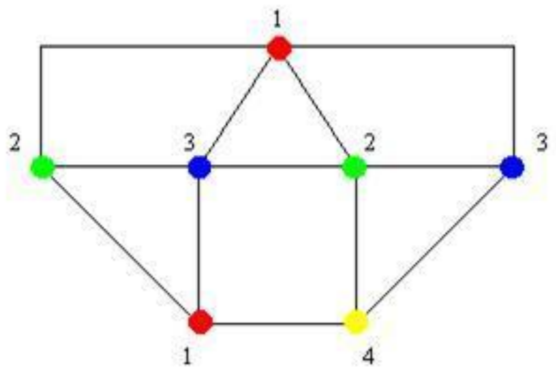
describing an algorithm to solve problem A under the assumption that an algorithm for problem B already exists. You've already been doing reduction for years, even before starting this book, only you probably called them something else, like subroutines or utility functions or modular programming or using a calculator. To prove something is NP-hard, we describe a similar transformation between problems, but not in the direction that most people expect.
To prove that your problem is hard, you need to describe an efficient algorithm to solve a different problem, which you already know is hard, using an hypothetical efficient algorithm for your problem as a black-box subroutine. The essential logic is a proof by contradiction. The reduction implies that if your problem were easy, then the other problem would be easy, which it ain't. Equivalently, since you know the other problem is hard, the reduction implies that your problem must also be hard;

10. Explain intractable problems with examples.

- From a computational complexity stance, intractable problems are problems for which there exist no efficient algorithms to solve them.
- Most intractable problems have an algorithm – the same algorithm – that provides a solution, and that algorithm is the brute-force search.
- This algorithm, however, does not provide an efficient solution and is, therefore, not feasible for computation with anything more than the smallest input.
- The reason there are no efficient algorithms for these problems is that these problems are all in a category which I like to refer to as "slightly less than random." They are so close to random, in fact, that they do not as yet allow for any meaningful algorithm other than that of brute-force.
- If any problem were truly random, there would not be even the possibility of any such algorithm.

As the size of the integers (i.e. the size of n) grows linearly, the size of the computations required to check all subsets and their respective sums grows exponentially. This is because, once again, we are forced to use the brute-force method to test the subsets of each division and their sums.

EXAMPLE : Graph colouring: How many colors do you need to color a graph such that no two adjacent vertices are of the same color?



Given any graph with a large number of vertices, we see that we are again faced with resorting to a systematic tracing of all paths, comparison of neighbouring colors, backtracking, etc., resulting in exponential time complexity once again.

11 - 16 Questions are as it is repeated from 1 - 6

17. What is the efficiency of Warshall's algorithm?

Warshall's Algorithm

Warshall's algorithm is used to determine the transitive closure of a directed graph or all paths in a directed graph by using the adjacency matrix. For this, it generates a sequence of n matrices. Where, n is used to describe the number of vertices.

$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots, R^{(n)}$

A sequence of vertices is used to define a path in a simple graph. In the kth matrix ($R^{(k)}$), ($r_{ij}^{(k)}$), the element's definition at the ith row and jth column will be one if it contains a path from v_i to v_j. For all intermediate vertices, w_q is among the first k vertices that mean $1 \le q \le k$.

```
Warshall(A[1...n, 1...n])
// A is the adjacency matrix
R(0) ← A
for k ← 1 to n do
for i ← 1 to n do
for j ← 1 to n do
R(k)[i, j] ← R(k-1)[i, j] or (R(k-1)[i, k] and R(k-1)[k, j])
return R(n)
```

- Time efficiency of this algorithm is (n^3)
- In the Space efficiency of this algorithm, the matrices can be written over their predecessors.
- $\Theta(n^3)$ is the worst-case cost. We should know that the brute force algorithm is better than Warshall's algorithm. In fact, the brute force algorithm is also faster for a space graph.

18. Define the principle of backtracking.

Backtracking is an algorithmic technique whose goal is to use brute force to find all solutions to a problem. It entails gradually compiling a set of all possible solutions. Because a problem will have constraints, solutions that do not meet them will be removed.

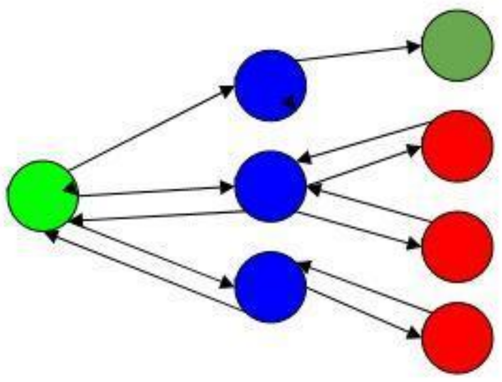
Backtracking is a technique based on algorithm to solve problem. It uses recursive calling to find the solution by building a solution step by step increasing values with time. It removes the solutions that don't give rise to the solution of the problem based on the constraints given to solve the problem.

Backtracking algorithm is applied to some specific types of problems,

- Decision problem used to find a feasible solution of the problem.
- Optimisation problem used to find the best solution that can be applied.

- Enumeration problem used to find the set of all feasible solutions of the problem.

In a backtracking problem, the algorithm tries to find a sequence path to the solution which has some small checkpoints from where the problem can backtrack if no feasible solution is found for the problem.



Here,
Green is the start point, blue is the intermediate point, red are points with no feasible solution, dark green is the end solution.
Here, when the algorithm propagates to an end to check if it is a solution or not, if it is then returns the solution otherwise backtracks to the point one step behind it to find the next point to find solution.

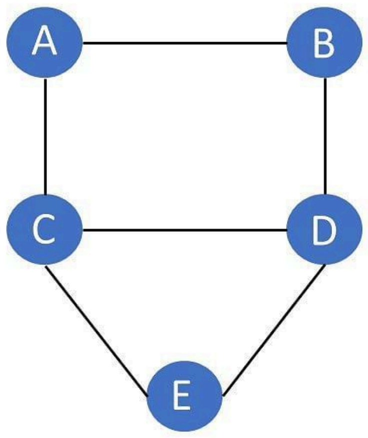
18. Define control abstraction for backtracking.

Control abstraction for backtracking defines the flow of how it solves the given problem in an abstract way.
In backtracking, solution is defined as n-tuple X = (x_1, x_2, ..., x_n), where x_i = 0 or 1. x_i is chosen from set of finite components S_i. If component x_i is selected then set x_i = 1 else set it to 0. Backtracking approach tries to find the vector X such that it maximizes or minimizes certain criterion function P(x_1, x_2, ..., x_n).

19. List the applications of backtracking.

1. To Find All Hamiltonian Paths Present in a Graph.

A Hamiltonian path, also known as a Hamilton path, is a graph path connecting two graph vertices that visit each vertex exactly once. If a Hamiltonian way exists with adjacent endpoints, the resulting graph cycle is a Hamiltonian or Hamiltonian cycle.



Hamilton Path: ABCDE

2. To Solve the N Queen Problem.

- The problem of placing n queens on the nxn chessboard so that no two queens attack each other is known as the n-queens puzzle.
- Return all distinct solutions to the n-queens puzzle given an integer n. You are free to return the answer in any order.
- Each solution has a unique board configuration for the placement of the n-queens, where 'Q' and. '' represent a queen and a space, respectively.

3. Maze Solving Problems

There are numerous maze-solving algorithms, which are automated methods for solving mazes. The random mouse, wall follower, Pledge, and Trémaux's algorithms are used within the maze by a traveller who has no prior knowledge of the maze. In contrast, a person or computer programmer uses the dead-end filling and shortest path algorithms to see the entire maze at once.

4. The Knight's Tour Problem

The Knight's tour problem is the mathematical problem of determining a knight's tour. A common problem assigned to computer science students is to write a program to find a knight's tour. Variations of the Knight's tour problem involve chess boards of different sizes than the usual n x n irregular (non-rectangular) boards.