MODULE 5
INTRODUCTION TO COMPLEXITY THEORY

Tractable and Intractable Problems

Tractable Problem: A problem that is solvable by a polynomial-time algorithm. The upper bound is polynomial.

Examples:

– Searching an unordered list

– Searching an ordered list

– Sorting a list

– Multiplication of integers

– Finding a minimum spanning tree in a graph

Intractable Problem: A problem that cannot be solved by a polynomial-time algorithm. Intractable problems are problems for which there exist no efficient algorithms to solve them. The lower bound is exponential.

Examples:

– Towers of Hanoi: we can prove that any algorithm that solves this problem must have a worst-case running time that is at least $2^n - 1$.

– List all permutations (all possible orderings) of n numbers.

Complexity Classes

In computer science, there exist some problems whose solutions are not yet found, the problems are divided into classes known as **Complexity Classes**. In complexity theory, a Complexity Class is a set of problems with related complexity. These classes help scientists to groups problems based on how much time and space they require to solve problems and verify the solutions.

The common resources are **time** and **space**, meaning how much time the algorithm takes to solve a problem and the corresponding memory usage. The time complexity of an algorithm is used to describe the number of steps required to solve a problem, but it can also be used to describe how long it takes to verify the answer. The space complexity of an algorithm describes how much memory is required for the algorithm to operate.

Types of Complexity Classes

There are 4 complexity classes:

1. **P Class**
2. **NP Class**
3. **NP- Hard**
4. **NP- Complete**

P Class

The P in the P class stands for **Polynomial Time.** It is the collection of decision problems (problems with a "yes" or "no" answer) that can be solved by a deterministic machine in polynomial time.

Features:

1. The solution to P problems is easy to find.
2. P is often a class of computational problems that are solvable and tractable. Tractable means that the problems can be solved in theory as well as in practice. But the problems that can be solved in theory but not in practice are known as intractable.

This class contains many natural problems like:

1. **Calculating the greatest common divisor.**
2. **Finding a maximum matching.**
3. **Decision versions of linear programming.**

NP Class

The NP in NP class stands for **Non-deterministic Polynomial Time**. It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time. NP class contains P class as a subset.

Features:

1. The solutions of the NP class are hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.
2. Problems of NP can be verified by a Turing machine in polynomial time.

This class contains many problems that one would like to be able to solve effectively:

1. **Boolean Satisfiability Problem (SAT).**
2. **Hamiltonian Path Problem.**
3. **Graph coloring.**

NP- Hard class

A problem is NP-Hard if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself.

Features:

1. All NP-hard problems are not in NP.
2. It takes a long time to check them. This means if a solution for an NP- Hard problem is given then it takes a long time to check whether it is right or not.
3. A problem A is in NP- Hard if, for every problem L in NP there exists a polynomial-time reduction from L to A.

Some of the examples of problems in NP-hard are:

1. **Halting problem.**
2. **Qualified Boolean formulas.**
3. **No Hamiltonian cycle.**

NP-complete class

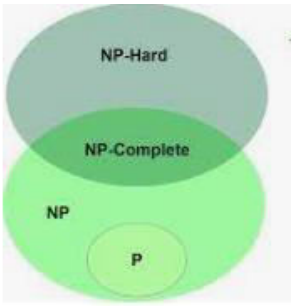
A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hard problems in NP.

Features:

1. NP-complete problems are special as any problem in NP class can be transformed or reduced into NP-complete problems in polynomial time.
2. If one could solve an NP-complete problem in polynomial time, then one could also solve any NP problem in polynomial time.

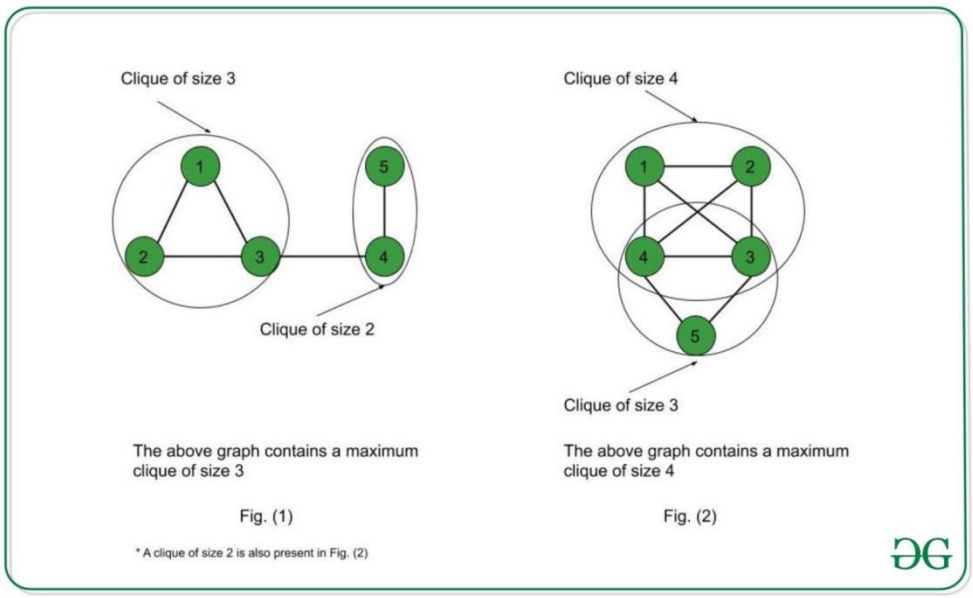
Some example problems include:

1. **0/1 Knapsack.**
2. **Hamiltonian Cycle.**
3. **Satisfiability.**
4. **Vertex cover.**



NP Completeness Proof of Clique Problem

A clique is a subgraph of a graph such that all the vertices in this subgraph are connected with each other i.e 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.



To prove that a problem is NP-Complete, we have to show that it belongs to both NP and NP-Hard Classes.

The Clique Decision Problem belongs to NP – If a problem belongs to the NP class, then it should have polynomial-time verifiability that is given a certificate(a solution), we should be able to verify in polynomial time if it is a solution to the problem.

Proof:

1. Certificate – Let the certificate be a set S consisting of nodes in the clique and S is a subgraph of G.
2. Verification – We have to check if there exists a clique of size k in the graph. Hence, verifying if number of nodes in S equals k, takes O(1) time. Verifying whether each vertex has an out-degree of (k-1) takes $O(k^2)$ time. (Since in a complete graph, each vertex is connected to every other vertex through an edge. Hence the total number of edges in a complete graph = ${}^kC_2 = k*(k-1)/2$. Therefore, to check if the graph formed by the k nodes in S is complete or not, it takes $O(k^2) = O(n^2)$ time (since k<=n, where n is number of vertices in G).

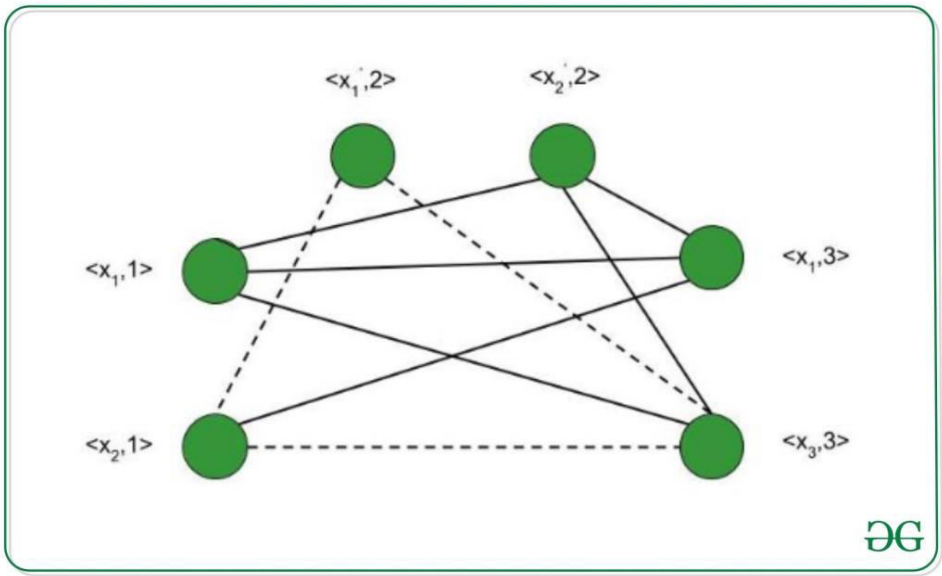
Therefore, the Clique Decision Problem has polynomial time verifiability and hence belongs to the NP Class.

The Clique Decision Problem belongs to NP-Hard – A problem L belongs to NP-Hard if every NP problem is reducible to L in polynomial time. Now, let the Clique Decision Problem is denoted by C. To prove that C is NP-Hard, we take an already known NP-Hard problem, say S, and reduce it to C for a particular instance. If this reduction can be done in polynomial time, then C is also an NP-Hard problem.

The Boolean Satisfiability Problem (S) is an NP-Complete problem as proved by the Cook's theorem. Therefore, every problem in NP can be reduced to S in polynomial time. Thus, if S is reducible to C in polynomial time, every NP problem can be reduced to C in polynomial time, thereby proving C to be NP-Hard.

Proof that the Boolean Satisfiability problem reduces to the Clique Decision Problem: Let the boolean expression be $F = (x_1 \text{ v } x_2) \text{ ^ } (x_1\text{'} \text{ v } x_2\text{'}) \text{ ^ } (x_1 \text{ v } x_3)$ where x_1, x_2, x_3 are the variables, '^' denotes logical 'and', 'v' denotes logical 'or' and x' denotes the complement of x. Let the expression within each parentheses be a clause. Hence we have three clauses – C_1, C_2 and C_3. From the above expression, if we are able to construct a graph containing a clique with size k = 3 (where k is no of clauses), then we could conclude that Boolean Satisfiability problem is reducible to clique problem. Consider the vertices as $<x_1, 1>$; $<x_2, 1>$; $<x_1', 2>$; $<x_2', 2>$; $<x_1, 3>$; $<x_3, 3>$ where the second term in each vertex denotes the clause number they belong to. We connect these vertices such that

1. No two vertices belonging to the same clause are connected.
2. No variable is connected to its complement.



Thus, the graph G (V, E) is constructed such that V = { <a, i> | a belongs to C_i } and E = { (<a, i>, <b, j>) | i is not equal to j ; b is not equal to a' } Consider the subgraph of G with the vertices <x_2, 1>; <x_1', 2>; <x_3, 3>. It forms a clique of size 3 (Depicted by dotted line in above figure). Corresponding to this, for the assignment <x_1, x_2, x_3> = <0, 1, 1> F evaluates to true. Therefore, if we have k clauses in our satisfiability expression, we get a max clique of size k and for the corresponding assignment of values, the satisfiability expression evaluates to true. i.e. F = (x_1 v x_2) ^ (x_1' v x_2') ^ (x_1 v x_3) = (0 v 1) ^ (1 v 0) ^ (0 v 1) = 1. Hence, for a particular instance, the satisfiability problem is reduced to the clique decision problem.

Therefore, the Clique Decision Problem is NP-Hard.

Conclusion

The Clique Decision Problem is NP and NP-Hard. Therefore, the Clique decision problem is NP-Complete.

NP Completeness Proof of Vertex Cover Problem

Problem – Given a graph G(V, E) and a positive integer k, the problem is to find whether there is a subset V' of vertices of size at most k, such that every edge in the graph is connected to some vertex in V'.

Explanation –

An instance of a problem is nothing but an input to the given problem. An instance of the Vertex Cover problem is a graph G (V, E) and a positive integer k, and the problem is to check whether a vertex cover of size at most k exists in G. Since an NP Complete problem, by definition, is a problem which is both in NP and NP hard, the proof for the statement that a problem is NP Complete consists of two parts:

1. **Proof that vertex cover is in NP –**

   If any problem is in NP, then, given a 'certificate' (a solution) to the problem and an instance of the problem (a graph G and a positive integer k, in this case), we will be able to verify (check whether the solution given is correct or not) the certificate in polynomial time.

   The certificate for the vertex cover problem is a subset V' of V, which contains the vertices in the vertex cover. We can check whether the set V' is a vertex cover of size k using the following strategy (for a graph G(V, E)):

   let count be an integer

```
set count to 0
for each vertex v in V'
   remove all edges adjacent to v from set E
   increment count by 1
   if count = k and E is empty
   then
      the given solution is correct
   else
      the given solution is wrong
```

It is plain to see that this can be done in polynomial time. Thus the vertex cover problem is in the class NP.

2. **Proof that vertex cover is NP Hard –**

To prove that Vertex Cover is NP Hard, we take some problem which has already been proven to be NP Hard, and show that this problem can be reduced to the Vertex Cover problem. For this, we consider the Clique problem, which is NP Complete (and hence NP Hard).

Here, we consider the problem of finding out whether there is a clique of size k in the given graph. Therefore, an instance of the clique problem is a graph G (V, E) and a non-negative integer k, and we need to check for the existence of a clique of size k in G.

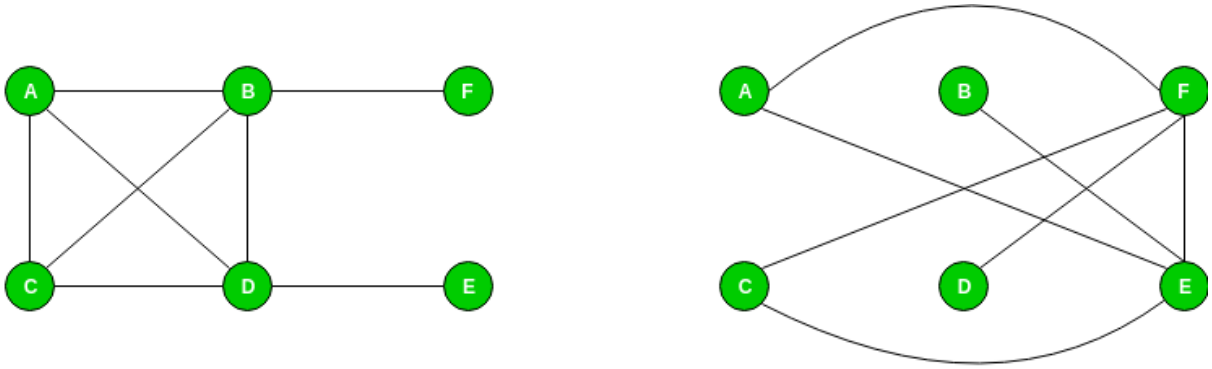
Now, we need to show that any instance (G, k) of the Clique problem can be reduced to an instance of the vertex cover problem. Consider the graph G' which consists of all edges not in G, but in the complete graph using all vertices in G. Let us call this the complement of G. Now, the problem of finding whether a clique of size k exists in the graph G is the same as the problem of finding whether there is a vertex cover of size |V| – k in G'. We need to show that this is indeed the case.

Assume that there is a clique of size k in G. Let the set of vertices in the clique be V'. This means |V'| = k. In the complement graph G', let us pick any edge (u, v). Then at least one of u or v must be in the set V – V'. This is because, if both u and v were from the set V', then the edge (u, v) would belong to V', which, in turn would mean that the edge (u, v) is in G. This is not possible since (u, v) is not in G. Thus, all edges in G' are covered by vertices in the set V – V'.

Now assume that there is a vertex cover V'' of size |V| – k in G'. This means that all edges in G' are connected to some vertex in V''. As a result, if we pick any edge (u, v) from G', both of them cannot be outside the set V''. This means, all edges (u, v) such that both u and v are outside the set V'' are in G, i.e., these edges constitute a clique of size k.

Thus, we can say that there is a clique of size k in graph G if and only if there is a vertex cover of size |V| – k in G', and hence, any instance of the clique problem can be reduced to an instance of the vertex cover problem. Thus, vertex cover is NP Hard. Since vertex cover is in both NP and NP Hard classes, it is NP Complete.

To understand the proof, consider the following example graph and its complement:



G
V' ={A, B, C, D}

G'
V'' ={E, F}

Approximation Algorithms

An approximation algorithm is an algorithm used to solve NP-complete optimization problems. This technique does not guarantee the best solution. The goal of the approximation algorithm is to come as close as possible to the optimal solution in polynomial time. Such algorithms are called approximation algorithms or heuristic algorithms.

Features of Approximation Algorithm :

- An approximation algorithm guarantees to run in polynomial time though it does not guarantee the most effective solution.
- An approximation algorithm guarantees to seek out high accuracy and top quality solution(say within 1% of optimum)

- Approximation algorithms are used to get an answer near the (optimal) solution of an optimization problem in polynomial time

Performance Ratios

Suppose we work on an optimization problem where every solution carries a cost. An Approximate Algorithm returns a legal solution, but the cost of that legal solution may not be optimal.

For Example, suppose we are considering for a **minimum size vertex-cover (VC)**. An approximate algorithm returns a VC for us, but the size (cost) may not be minimized.

Another Example is we are considering for a **maximum size Independent set (IS)**. An approximate Algorithm returns an IS for us, but the size (cost) may not be maximum.

Let C be the cost of the solution returned by an approximate algorithm, and C* is the cost of the optimal solution.

We say the approximate algorithm has an approximate ratio P (n) for an input size n, where

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq P(n)$$

Intuitively, the approximation ratio measures how bad the approximate solution is distinguished with the optimal solution. A large (small) approximation ratio measures the solution is much worse than (more or less the same as) an optimal solution.

Observe that P (n) is always ≥ 1, if the ratio does not depend on n, we may write P. Therefore, a 1-approximation algorithm gives an optimal solution. Some problems have polynomial-time approximation algorithm with small constant approximate ratios, while others have best-known polynomial time approximation algorithms whose approximate ratios grow with n.

Bin Packing Problem

Given n items of different weights and bins each of capacity c, assign each item to a bin such that number of 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, 1, 1}

Input: weight[] = {9, 8, 2, 2, 5, 4}
Bin Capacity c = 10
Output: 4
We need minimum 4 bins to accommodate all items.

Input: weight[] = {2, 5, 4, 7, 1, 3, 8};
Bin Capacity c = 10
Output: 3

Lower Bound

The lower bound on minimum number of bins required 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. This problem is a NP Hard problem and finding an exact minimum number of bins takes exponential time.

Applications

1. Loading of containers like trucks.
2. Placing data on multiple disks.
3. Job scheduling.
4. Packing advertisements in fixed length radio/TV station breaks.
5. Storing a large collection of music onto tapes/CD's, etc.

Online Algorithms

These algorithms are for Bin Packing problems where items arrive one at a time (in unknown order), 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 optimal. 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.

Example:

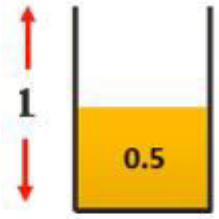
Consider bins of size 1

Assuming the sizes of the items be {0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6}.

The minimum number of bins required would be Ceil ((Total Weight) / (Bin Capacity))= Celi(3.7/1) = 4 bins.

The Next fit solution (NF (I)) for this instance I would be-

Considering 0.5 sized item first, we can place it in the first bin

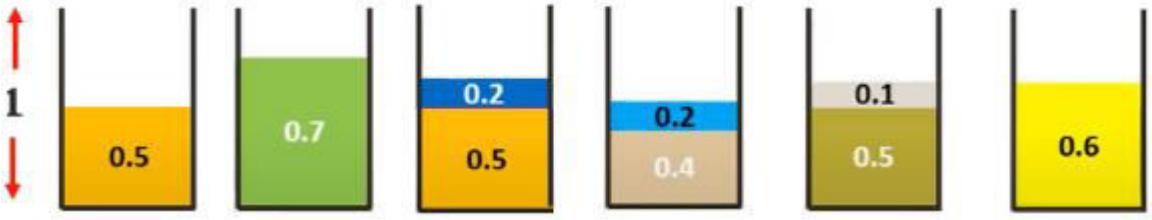


Moving on to the 0.7 sized item, we cannot place it in the first bin. Hence we place it in a new bin.

Moving on to the 0.5 sized item, we cannot place it in the current bin. Hence we place it in a new bin.

Moving on to the 0.2 sized item, we can place it in the current (third bin)

Similarly, placing all the other items following the Next-Fit algorithm we get-



Thus we need 6 bins as opposed to the 4 bins of the optimal solution. Thus we can see that this algorithm is not very efficient.

2. First Fit: When processing the next item, scan the previous bins in order and place the item in the first bin that fits. Start a new bin only if it does not fit in any of the existing bins.

The above implementation of First Fit requires $O(n^2)$ time, but First Fit can be implemented in O(n Log n) time using Self-Balancing Binary Search Trees.

If M is the optimal number of bins, then First Fit never uses more than 1.7M bins. So First-Fit is better than Next Fit in terms of upper bound on number of bins.

Example:

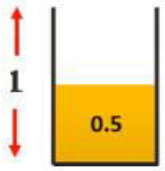
Consider bins of size 1

Assuming the sizes of the items be {0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6}.

The minimum number of bins required would be Ceil ((Total Weight) / (Bin Capacity))= Celi(3.7/1) = 4 bins.

The First fit solution (FF(I))for this instance I would be-

Considering 0.5 sized item first, we can place it in the first bin



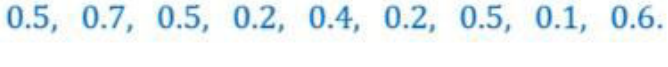
Moving on to the 0.7 sized item, we cannot place it in the first bin. Hence we place it in a new bin.

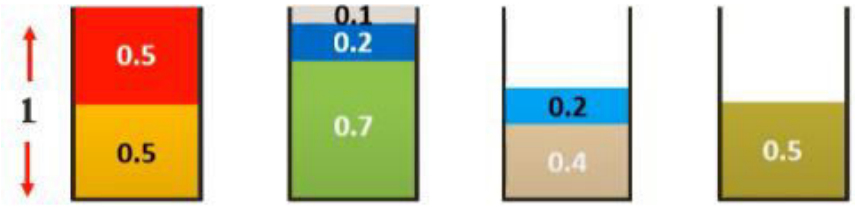
Moving on to the 0.5 sized item, we can place it in the first bin.

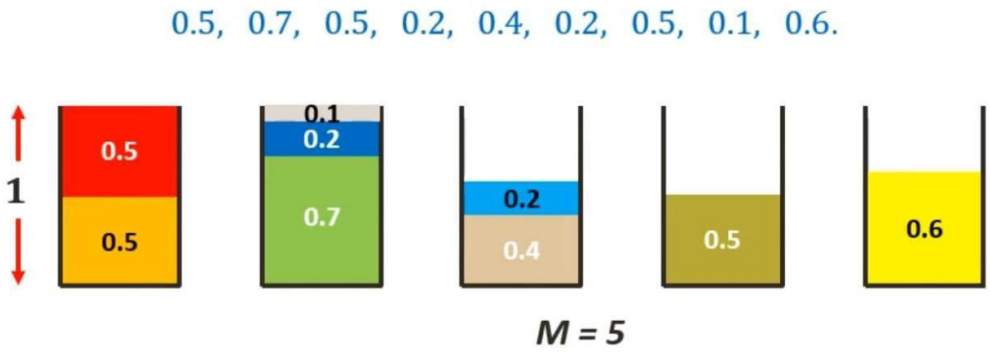
Moving on to the 0.2 sized item, we can place it in the first bin, we check with the second bin and we can place it there.

Moving on to the 0.4 sized item, we cannot place it in any existing bin. Hence we place it in a new bin.

Similarly, placing all the other items following the First-Fit algorithm we get-







Thus we need 5 bins as opposed to the 4 bins of the optimal solution but is much more efficient than Next-Fit algorithm.

3. Best Fit: The idea is to places the next item in the *tightest* spot. That is, put it in the bin so that the smallest empty space is left.

Best Fit can also be implemented in O(n Log n) time using Self-Balancing Binary Search Trees.

If M is the optimal number of bins, then Best Fit never uses more than 1.7M bins. So Best Fit is same as First Fit and better than Next Fit in terms of upper bound on number of bins.

Example:

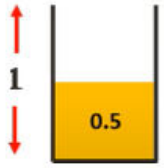
Consider bins of size 1

Assuming the sizes of the items be {0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6}.

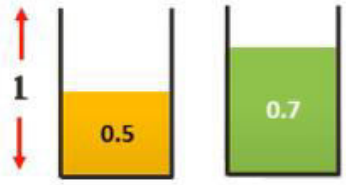
The minimum number of bins required would be Ceil ((Total Weight) / (Bin Capacity))= Celi(3.7/1) = 4 bins.

The First fit solution (FF(I))for this instance I would be-

Considering 0.5 sized item first, we can place it in the first bin



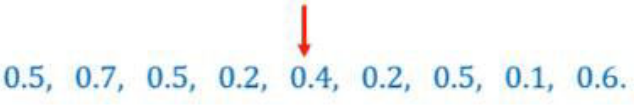
Moving on to the 0.7 sized item, we cannot place it in the first bin. Hence we place it in a new bin.



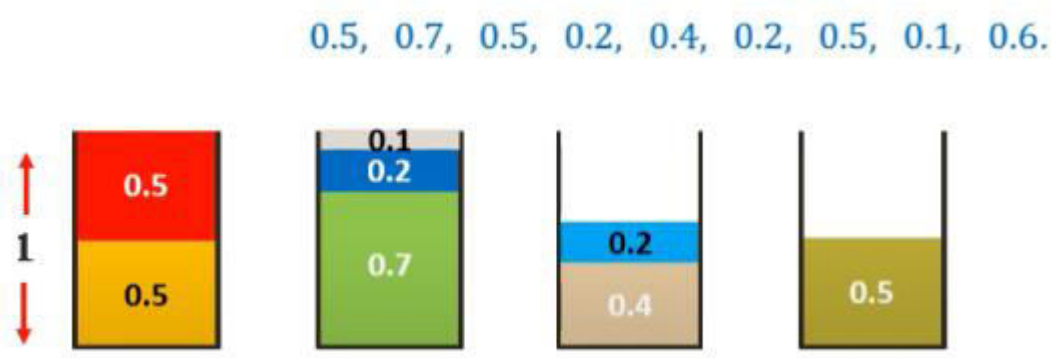
Moving on to the 0.5 sized item, we can place it in the first bin tightly.

Moving on to the 0.2 sized item, we cannot place it in the first bin but we can place it in second bin tightly.

Moving on to the 0.4 sized item, we cannot place it in any existing bin. Hence we place it in a new bin.



Similarly, placing all the other items following the Best-Fit algorithm we get-



Thus we need 5 bins as opposed to the 4 bins of the optimal solution but is much more efficient than Next-Fit algorithm.

4. Worst Fit: The idea is to places the next item in the least tight spot to even out the bins. That is, put it in the bin so that most empty space is left. Worst Fit can also be implemented in O(n Log n) time using Self-Balancing Binary Search Trees. If M is the optimal number of bins, then Best Fit never uses more than 2M-2 bins. So Worst Fit is same as Next Fit in terms of upper bound on number of bins.

Offline Algorithms

In the offline version, we have all items upfront. Unfortunately offline version is also NP Complete, but we have a better approximate algorithm for it. First Fit Decreasing uses at most (4M + 1)/3 bins if the optimal is M.

4. First Fit Decreasing:

We first sort the array of items in decreasing size by weight and apply first-fit algorithm as discussed above

Algorithm

- Read the inputs of items
- Sort the array of items in decreasing order by their sizes
- Apply First-Fit algorithm

First Fit Decreasing can also be implemented in O(n Log n) time using Self-Balancing Binary Search Trees.

Example:

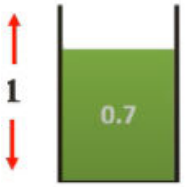
Assuming the sizes of the items be {0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6}.

Sorting them we get {0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1}

The First fit Decreasing solution would be-

We will start with 0.7 and place it in the first bin

0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1

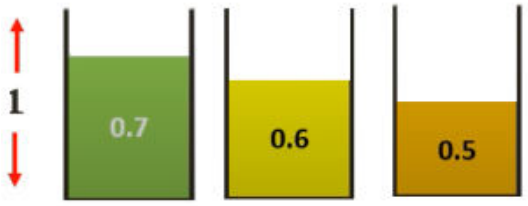


We then select 0.6 sized item. We cannot place it in bin 1. So, we place it in bin 2

0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1

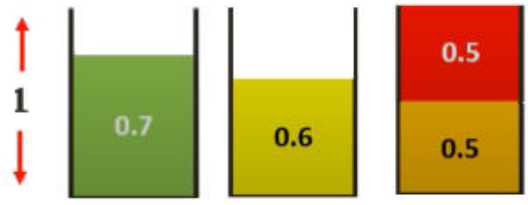
We then select 0.5 sized item. We cannot place it in any existing. So, we place it in bin 3

0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1



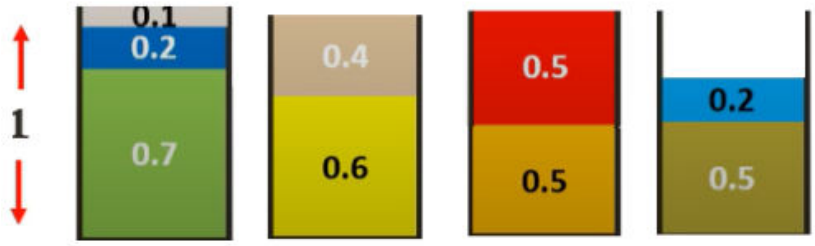
We then select 0.5 sized item. We can place it in bin 3

0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1



Doing the same for all items, we get

0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1



Thus only 4 bins are required which is the same as the optimal solution.

6. Best fit algorithm

We first sort the array of items in decreasing size by weight and apply Best-fit algorithm as discussed above

Algorithm

- Read the inputs of items
- Sort the array of items in decreasing order by their sizes
- Apply Best-Fit algorithm

Example:

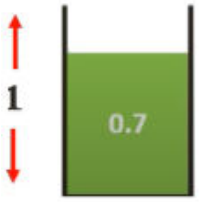
Consider bins of size 1

Assuming the sizes of the items be {0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6}. Sorting them we get {0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1}

The Best fit Decreasing solution would be-

We will start with 0.7 and place it in the first bin

0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1

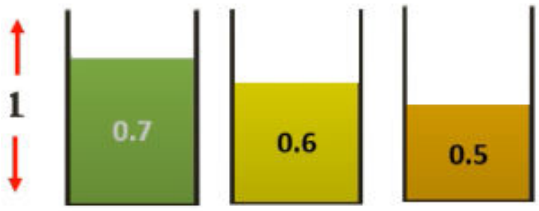


We then select 0.6 sized item. We cannot place it in bin 1. So, we place it in bin 2

0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1

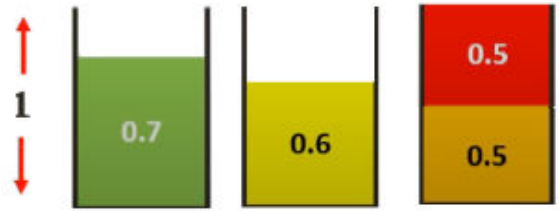
We then select 0.5 sized item. We cannot place it in any existing. So, we place it in bin 3

0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1



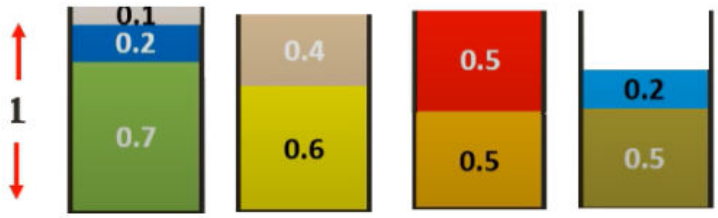
We then select 0.5 sized item. We can place it in bin 3

0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1



Doing the same for all items, we get.

0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1



Thus only 4 bins are required which is the same as the optimal solution.

Graph Coloring algorithm

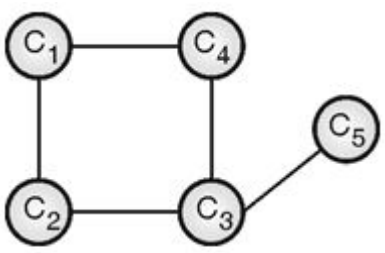
Graph coloring refers to the problem of coloring vertices of a graph in such a way that no two adjacent vertices have the same color. This is also called the **vertex coloring** problem.

- If coloring is done using at most k colors, it is called **k-coloring**.
- The smallest number of colors required for coloring graph is called its **chromatic number**.
- The chromatic number is denoted by X(G). Finding the chromatic number for the graph is NP-complete problem.
- Graph coloring problem is both, decision problem as well as an optimization problem. A decision problem is stated as, "With given M colors and graph G, whether such color scheme is possible or not?".
- The optimization problem is stated as, "Given M colors and graph G, find the minimum number of colors required for graph coloring."

Applications of Graph Coloring Problem

- Design a timetable.
- Sudoku
- Register allocation in the compiler
- Map coloring
- Mobile radio frequency assignment:

The input to the graph is an adjacency matrix representation of the graph. Value M(i, j) = 1 in the matrix represents there exists an edge between vertex i and j. A graph and its adjacency matrix representation are shown in Figure (a)

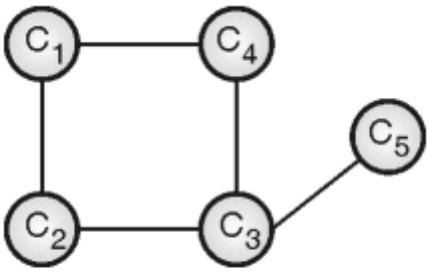


(a). Graph G

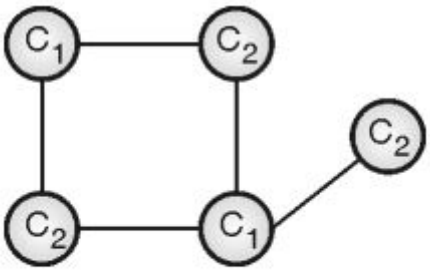
	C_1	C_2	C_3	C_4	C_5
C_1	0	1	0	1	0
C_2	1	0	1	0	0
C_3	0	1	0	1	1
C_4	1	0	1	0	1
C_5	0	0	1	0	0

Adjacency matrix for graph G

The problem can be solved simply by assigning a unique color to each vertex, but this solution is not optimal. It may be possible to color the graph with colors less than |V|. Figure (b) and figure (c) demonstrate both such instances. Let Ci denote the i^{th} color.



(b). Nonoptimal solution (uses 5 colors)



(c). Optimal solution (uses 2 colors)

- This problem can be solved using backtracking algorithms as follows:
  - List down all the vertices and colors in two lists
  - Assign color 1 to vertex 1
  - If vertex 2 is not adjacent to vertex 1 then assign the same color, otherwise assign color 2.
  - Repeat the process until all vertices are colored.
- Algorithm backtracks whenever color i is not possible to assign to any vertex k and it selects next color i + 1 and test is repeated. Consider the graph shown in Figure (d).

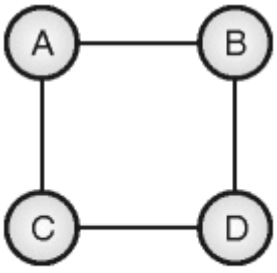


Figure (d)

If we assign color 1 to vertex A, the same color cannot be assigned to vertex B or C. In the next step, B is assigned some different colors 2. Vertex A is already colored and vertex D is a neighbor of B, so D cannot be assigned color 2. The process goes on. State-space tree is shown in Figure (e)

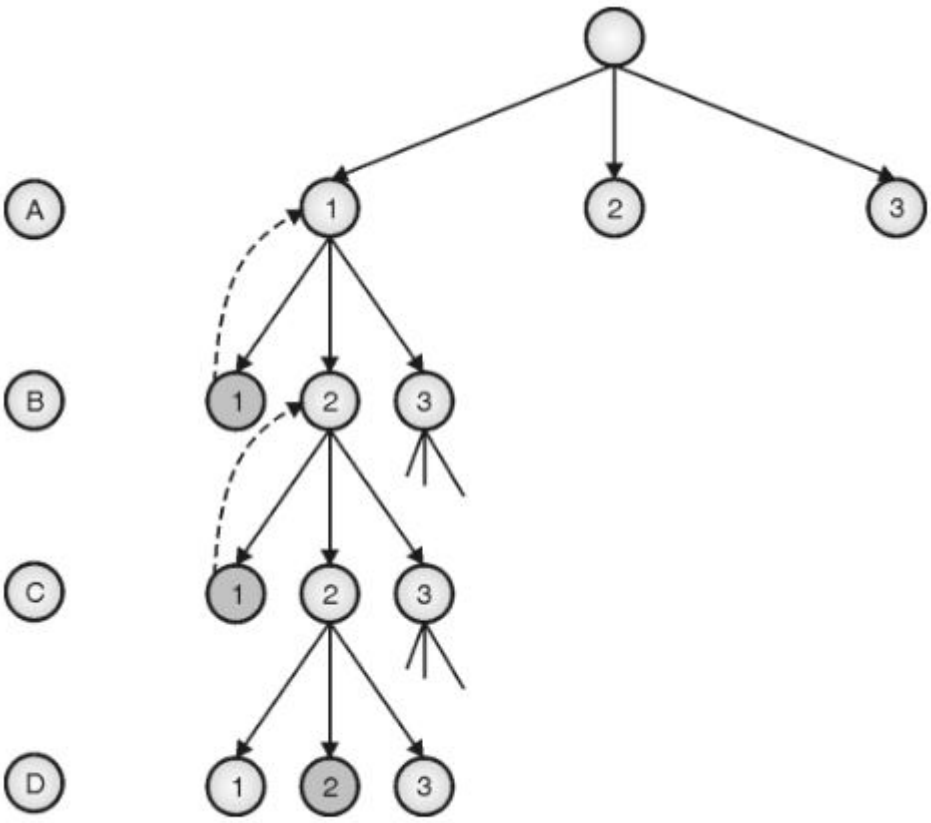


Figure (e): State-space tree of the graph of Figure (d)

Thus, vertices A and C will be colored with color 1, and vertices B and D will be colored with color 2.

Complexity Analysis

The number of anode increases exponentially at every level in state space tree. With M colors and n vertices, total number of nodes in state space tree would be $1 + M + M^2 + M^3 + \ldots + M^n$

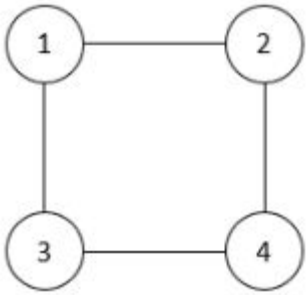
Hence, $T(n) = 1 + M + M^2 + M^3 + \ldots + M^n$

So, $T(n) = O(M^n)$.

Thus, the graph coloring algorithm runs in exponential time.

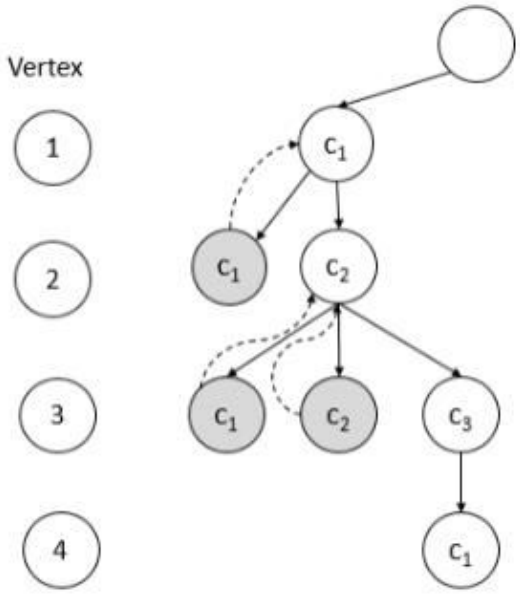
Examples on Graph Coloring Problem

Example: Apply backtrack on the following instance of graph coloring problem of 4 nodes and 3 colors



Solution:

This problem can be solved using backtracking algorithms. The formal idea is to list down all the vertices and colors in two lists. Assign color 1 to vertex 1. If vertex 2 is not adjacent to vertex 1 then assign the same color, otherwise assign color 2. The process is repeated until all vertices are colored. The algorithm backtracks whenever color i is not possible to assign to any vertex k and it selects the next color i + 1 and the test is repeated. This graph can be colored with 3 colors. The solution tree is shown in Fig. (i):

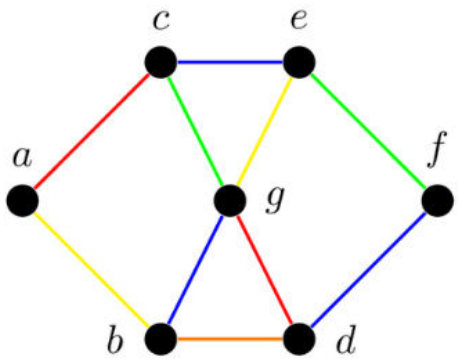


Flg. (i): Solution tree

Vertex	Assigned Color
1	C_1
2	C_2
3	C_3
4	C_1

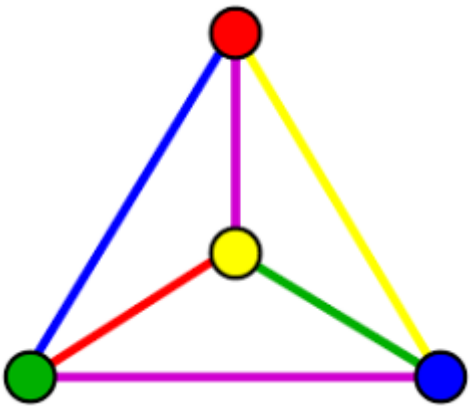
Edge coloring

An **edge coloring** of a graph is a proper coloring of the edges, meaning an assignment of colors to edges so that no vertex is incident to two edges of the same color. An edge coloring with k colors is called a **k-edge-coloring** and is equivalent to the problem of partitioning the edge set into k matchings. The smallest number of colors needed for an edge coloring of a graph G is the chromatic index, or edge chromatic number, $\chi'(G)$.



Total coloring

Total coloring is a type of coloring on the vertices and edges of a graph. A total coloring is that no adjacent vertices, no adjacent edges, and no edge and its end-vertices are assigned the same color. The total chromatic number $\chi''(G)$ of a graph G is the fewest colors needed in any total coloring of G.



Randomized Algorithms

An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm. For example, in Randomized Quick Sort, we use a random number to pick the next pivot (or we randomly shuffle the array). Typically, this randomness is used to reduce time complexity or space complexity in other standard algorithms.

Randomized algorithms are classified in **two** categories.

Las Vegas: These algorithms always produce correct or optimum result. Time complexity of these algorithms is based on a random value and time complexity is evaluated as expected value. For example, Randomized QuickSort always sorts an input array and expected worst case time complexity of QuickSort is O(nLogn).

Monte Carlo: Produce correct or optimum result with some probability. These algorithms have deterministic running time and it is generally easier to find out worst case time complexity. For example this implementation of Karger's Algorithm produces minimum cut with probability greater than or equal to $1/n^2$ (n is number of vertices) and has worst case time complexity as O(E).

Example to Understand Classification:

Consider a binary array where exactly half elements are 0 and half are 1. The task is to find index of any 1.

A Las Vegas algorithm for this task is to keep picking a random element until we find a 1. A Monte Carlo algorithm for the same is to keep picking a random element until we either find 1 or we have tried maximum allowed times say k.
The Las Vegas algorithm always finds an index of 1, but time complexity is determined as expect value. The expected number of trials before success is 2, therefore expected time complexity is O(1). The Monte Carlo Algorithm finds a 1 with probability $[1 – (1/2)^k]$. Time complexity of Monte Carlo is O(k) which is deterministic.

Randomized version of QuickSort algorithm with analysis

A **Central Pivot** is a pivot that divides the array in such a way that one side has at-least 1/4 elements.

// Sorts an array arr[low..high]

randQuickSort(arr[], low, high)

1. If low >= high, then EXIT.

2. While pivot 'x' is not a Central Pivot.

(i) Choose uniformly at random a number from [low..high].
Let the randomly picked number be **x**.

(ii) Count elements in arr[low..high] that are smaller than arr[x]. Let this count be **sc**.

(iii) Count elements in arr[low..high] that are greater than arr[x]. Let this count be **gc**.

(iv) Let **n** = (high-low+1). If sc >= n/4 and gc >= n/4, then x is a central pivot.

3. Partition arr[low..high] around the pivot x.

4. // Recur for smaller elements
randQuickSort(arr, low, sc-1)

5. // Recur for greater elements
randQuickSort(arr, high-gc+1, high)

In this analysis, the time taken by step 2 is O(n).

How many times while loop runs before finding a central pivot?
The probability that the randomly chosen element is central pivot is 1/n.
Therefore, expected number of times the while loop runs is n.
Thus, the expected time complexity of step 2 is O(n).

What is overall Time Complexity in Worst Case?
In worst case, each partition divides array such that one side has n/4 elements and other side has 3n/4 elements. The worst case height of recursion tree is $\text{Log}_{3/4}$ n which is O(Log n).

$T(n) < T(n/4) + T(3n/4) + O(n)$
$T(n) < 2T(3n/4) + O(n)$
Solution of above recurrence is O(n Log n)

Applications and Scope:

- Randomized algorithms have huge applications in Cryptography.
- Load Balancing.
- Data Structures: Hashing, Sorting, Searching, Order Statistics and Computational Geometry.
- Mathematical programming: Faster algorithms for linear programming, Rounding linear program solutions to integer program solutions
- Graph algorithms: Minimum spanning trees, shortest paths, minimum cuts.
- Counting and enumeration: Matrix permanent Counting combinatorial structures.
- Parallel and distributed computing: Deadlock avoidance distributed consensus.
- Probabilistic existence proofs: Show that a combinatorial object arises with non-zero probability among objects drawn from a suitable probability space.

UNIVERSITY QUESTIONS

1. Compare Tractable and Intractable problems

Tractable problems	Intractable problems
– Can be solved in polynomial time. – Can be verified in exponential time – It is easy to solve	– Can be solved in exponential time – Can be verified in exponential time – It is not easy to solve

2. What do you mean by intractable problems?(2 Marks)

   Answer key: Intractable problems – 2 Marks

3. State the difference between P, NP, NP Hard and NP Complete classes.(4 Marks)

   Answer key: Each definition carries 1 Marks

4. Differentiate between P and NP problems.
5. Define class P and class NP. (2 Marks)
6. Define NP Hard and NP Complete problems. (4 Marks)

   Answer key: NP Hard – 2 Marks

   NP Complete – 2 Marks

7. With examples explain polynomial time reducibility.(4 Marks)

   Answer key: polynomial time reducibility – 2 Marks

   Example – 2 Marks

8. Specify the relevance of approximation algorithms.
9. What are the steps used to show a given problem is NP Complete.(4 Marks)
10. Consider the following algorithm to determine whether or not an undirected graph has a clique of size k. First, generate all subsets of the vertices containing exactly k vertices. Next check whether any of the subgraphs induced by these subsets is complete (i.e. forms a clique). Why is this not a polynomial-time algorithm for the clique problem, thereby implying that P=NP?(4 Marks)

    Answer key: If the algorithm runs in polynomial time, then, because the clique problem is NP complete, this would imply that P=NP. (2 marks)

    Showing that the algorithm does not run in polynomial time. (2 Marks)

11. Prove that Clique problem is NP- Complete.(4 Marks)

    Answer key: Showing CLIQUE problem is in NP. (2 Marks)

Giving a polynomial time reduction from a known NP Complete problem to CLIQUE + showing that the reduction is polynomial time.(2 Marks)

12. With the help of suitable code sequence convince Vertex Cover Problem is an example of NP- Complete Problem.

13. Explain Vertex Cover Problem using an example. Suggest an algorithm for finding Vertex Cover of a graph.

14. Write a short note on Approximation algorithms.

15. State bin packing problem? Explain the first fit decreasing strategy.

16. Explain the need for randomized algorithms. Differentiate Las Vegas and Monte Carlo algorithms.

A **randomized algorithm** is a technique that uses a source of randomness as part of its logic. It is typically used to reduce either the running time, or time complexity; or the memory used, or space complexity, in a standard algorithm. The algorithm works by generating a random number, *r*, within a specified range of numbers, and making decisions based on *r*'s value.

A randomized algorithm could help in a situation of doubt by flipping a coin or a drawing a card from a deck in order to make a decision. Similarly, this kind of algorithm could help speed up a brute force process by randomly sampling the input in order to obtain a solution that may not be totally optimal, but will be good enough for the specified purposes.

17. Compare Conventional quicksort algorithm and Randomized quicksort with the help of a suitable example.

18. Explain randomized quicksort and analyse the expected running time of randomized quicksort with the help of suitable example.