Module - 2
Probabilistic Analysis and Randomized Algorithms.

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 random number to pick the next pivot.

→ Algorithms where some of the actions are dependent on chance are generally termed as probabilistic algorithms or randomized algorithms.

→ For many problems, randomized algorithms run faster than the known best deterministic algorithms. Moreover, these algorithms are simpler to describe and implement than the deterministic algorithms.

→ The results of randomized algorithms may not always be 100% correct. Correctness of the output of these algorithms is characterized by some probability. This is done by running the algorithms for a longer time.

→ probabilistic algorithms react differently for the same input at different times. The execution time and the result may be different from one use to the next. This is because of their dependance on the use of random variables.

→ A randomized algorithm is one that makes use of a randomizer (such as a random number generator). Some of the decisions made in the algorithm depends on the output of the randomizer.

→ Randomized algorithms are classified in two categories:-

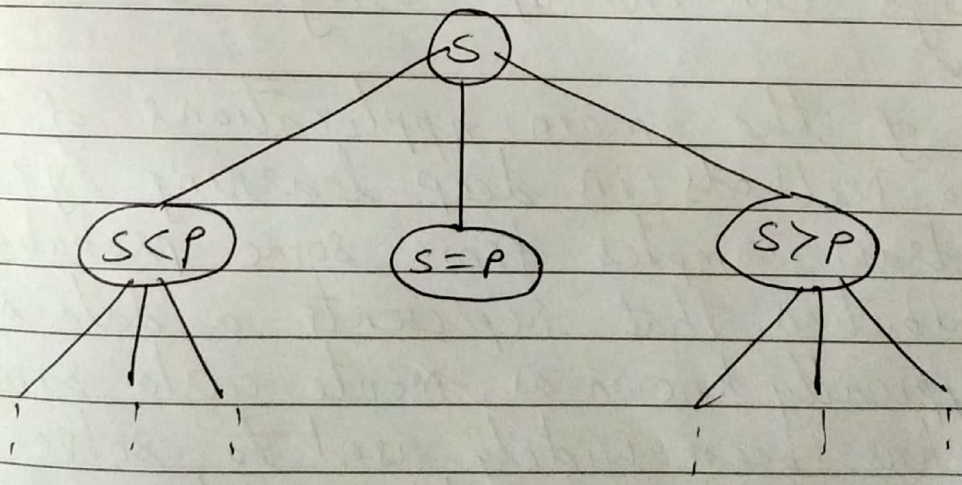
1) Las Vegas algorithms
2) Monte Carlo algorithms.

1) Las Vegas algorithms →

→ These algorithms always produce correct or optimum result. Time complexity of these algorithms is based on a random value.

eg) Randomized quick sort algorithm picks a pivot at random, and then partitions the elements into three sets:-

a) all the elements less than the pivot.
b) all the elements equal to the pivot.
c) all the elements greater than the pivot.



→ The randomized quick sort tends to consume a lot of resources but guarantees an ~~extra answer~~, exact answer. Thus, Las Vegas method tends to be recommended in scenarios with a small number of potential answers.

2) Monte Carlo Algorithms →

→ These algorithms produces 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.

→ There is also generally a probability guarantee that the answer is right.

→ For example:- If you used a non-perfect hash to assign hash values to two different strings and then try to see if the strings are the same or not by comparing the hash values, then this is like a Monte Carlo algorithm. You will mostly get the right answer but sometimes two different strings can end up having the same hash value.

→ One of the main applications of Monte Carlo methods in deep learning systems is to draw samples from some probability distribution that represents a data set. This is typically known as Monte carlo sampling and has been widely used to solve highly complex data estimation problems.

→ There are several monte Carlo techniques that have been widely implemented in modern deep learning platforms. The best known member of the Monte Carlo family is a technique that brings Markov chains into the world of randomness and is commonly known as Markov chain Monte carlo methods (MCMC)

Comparison:-

→ Las Vegas methods tend to always provide an exact answer while Monte Carlo methods returns answers with a random amount of error. Thus, the degree of error in Monte Carlo system decreases with the increase in resources such as data or computation models.

→ Las Vegas models can never expect to produce an exact answer. Monte Carlo techniques addresses some of the limitations of Las Vegas algorithms by improving the efficiency of the computation graph, introducing certain level of randomness in the answers.

* Hiring problem →

→ Consider the problem of hiring an office assistant. We interview candidates on a rolling basis, and at any given point we want to hire the best candidate we have seen so far. If a better candidate comes along, we immediately fire the old one and hire the new one.

Let, n → number of candidates for the office assistant job.

c_i → cost associated with interviewing the people.

c_h → larger cost associated with hiring the people.

Algorithm →

```
HIRE_ASSISTANT (n)
{
    best = 0   // candidate 0 is a least-
               qualified dummy candidate
    for i = 1 to n
    {
        interview candidate i
        if candidate i is better than candidate best
        {
            best = i
            hire candidate i
        }
    }
}
```

→ The cost of the above algorithm is

$$O(C_i n + C_h m)$$

where $n \rightarrow$ total number of applicants
& $m \rightarrow$ number of times we hire a new person

* Worst Case Analysis →

→ In the worst case scenario, the candidates come in order of increasing quality, and we hire every person that we interview. Then the hiring cost is $O(C_h n)$ & the total cost is -

$$O[(C_i + C_h) n]$$

Smita Attarde

* Average Case Analysis →

→ We mainly focus on the worst case cost of algorithms. Worst case analysis is very important, but sometimes algorithms with poor worst-case performance may be useful in practice.

→ Defining a average-case requires certain assumptions on how often different types of inputs come up.

→ One possible assumption is that all permutations of candidates are equally likely. But this assumption might not always be true.

For example:- Candidates may be sourced through a recruiter, who sends the strongest candidates first because those are the candidates who are most likely to get him a referral bonus. Without knowing the distribution of input, its hard to perform average-case analysis.

→ If we know the distribution of inputs, we can compute the average case of an algorithm by finding the cost of each input, and then averaging the costs together in accordance with how likely the input is to come up.

Example:- Suppose the algorithms costs are as follows:-

input	probability that the input happens	cost of the algorithm when run on that input
A	0.5	1
B	0.3	2
C	0.2	3

Then the average-cost of the algorithm is just the expected value of the cost, which is-

$1 * 0.5 + 2 * 0.3 + 3 * 0.2$

* **Indicator random variables →**

→ An indicator random variable that indicates whether an event is happening.

→ If A is an event, then the indicator random variable I_A is defined as-

$$I_A = \begin{cases} 1 & \text{if } A \text{ occurs} \\ 0 & \text{if } A \text{ does not occur.} \end{cases}$$

Example: Suppose we are flipping a coin n times.

Let X_i → indicator random variable associated with the coin coming up heads on the i^{th} coin flip.

So,

$$X_i = \begin{cases} 1 & \text{if } i^{th} \text{ coin is heads} \\ 0 & \text{if } i^{th} \text{ coin is tails} \end{cases}$$

→ By summing the values of X_i, we can get the total number of heads accross the n coin flips.

→ To find the expected number of heads,

ie. $$E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i]$$

Now,

$$\begin{aligned} E[X_i] &= 1 * P(X_i = 1) + 0 * P(X_i = 0) \\ &= 1 * (1/2) + 0 * (1/2) \\ &= 1/2 \end{aligned}$$

Smita Attarde

So, the expected number of heads is-

$$\sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} (1/2) = n/2$$

* Analysis of Hire Assistant →

→ Let,

X_i → Indicator random variable that indicates whether candidate i is hired.

Let $X_i = \begin{cases} 1 & \text{if candidate } i \text{ is hired} \\ 0 & \text{otherwise} \end{cases}$

→ Let $X = \sum_{i=1}^{n} X_i$ be the number of times we hire a new candidate. We want to find $E[X]$, which is- $\sum_{i=1}^{n} E[X_i]$

Now, we know, $E[X_i] = P$ (candidate i is hired)
So we just need to find this probability.
To do this, we assume that the candidates are interviewed in a random order.

→ Candidate i is hired when candidate i is better than all of the candidates 1 through i-1. Now, consider only the first i candidates, which must appear in a random order. Anyone of them is equally likely to be the best-qualified so far. So, the probability that candidate i is better than candidates 1 through i-1 is just 1/i.

Smita Attarde

$\therefore E[X_i] = 1/i$

$\& E[X] = \sum_{i=1}^{n} 1/i = \log n + O(1)$

So, the expected number of candidates hired is $O(\log n)$ & the expected hiring cost is ~~$O(C_h \log n)$~~ $O(C_h \log n)$.

Module 6

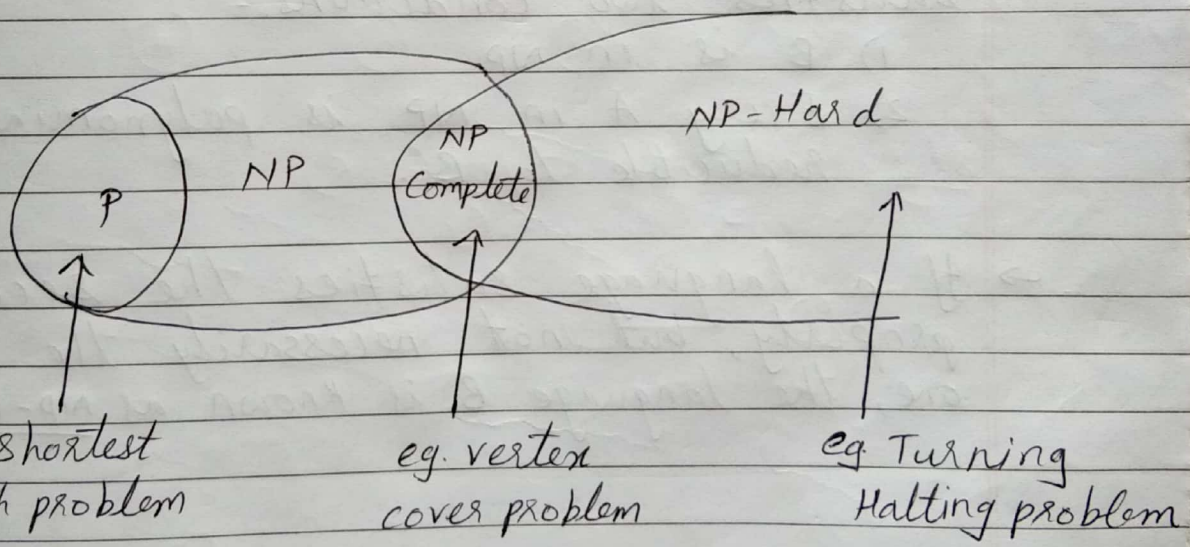
* **Optimization problems →**
Optimization problems are those for which the objective is to maximize or minimize some values.
For example:-
1) Finding the minimum number of colors needed to color a given graph.
2) Finding the shortest path between two vertices in a graph.

* **Decision problems →**
There are many problems for which the answer is a Yes or a No. These types of problems are known as decision problems.
For example:-
1) Whether a given graph can be colored by only 4 colors.
2) Finding Hamiltonian cycle in a graph is not a decision problem, whereas checking a graph is Hamiltonian or not is a decision problem.

* P-class →
The class P etc contains those problems that are solvable in polynomial time ie. these problems can be solved in time O(nk) in worst case, where k is constant.

* NP-class →
→ The class NP consists of those problems that can be verified in polynomial time
→ NP is the class of decision problems for which it is easy to check the correctness of a answer.
→ Every problem in this class can be solved in exponential time.

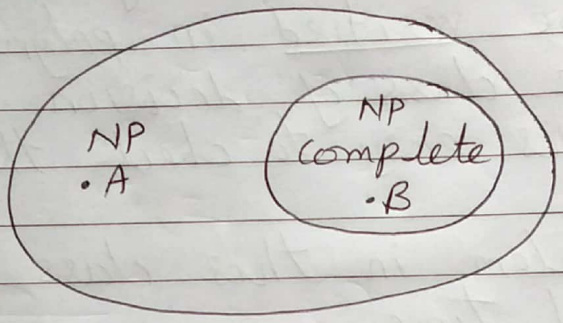
* P versus NP →



* NP-complete class →
The problems in NP, which can be solved in polynomial time are called as NP-complete problems.

Smita Attarde

* NP-Hard →
→ A problem is NP-Hard if an algorithm for solving it can be translated into one for solving any NP problem.
→ Thus, NP-Hard means "at least as hard as any NP-problem".



→ A language B is NP-complete, if it satisfies two conditions:-
1) B is in NP
2) Every A in NP is polynomial time reducible to B.

→ If a language satisfies the second property, but not necessarily the first one, the language B is known as NP-Hard.

* Vertex Cover problem →

→ A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either 'u' or 'v' is in the vertex cover, ~~there~~ where $(u, v) \in E$

→ Although the name is vertex cover, the set covers all the edges of the given graph.

→ Thus, the vertex cover problem is to find minimum size vertex cover.

→ Vertex cover is a subset of vertices, $V' \subseteq V$, such that if the edge $(u, v) \in E$, then $u \in V'$ or $v \in V'$

→ The size of the cover is the number of vertices in V'.

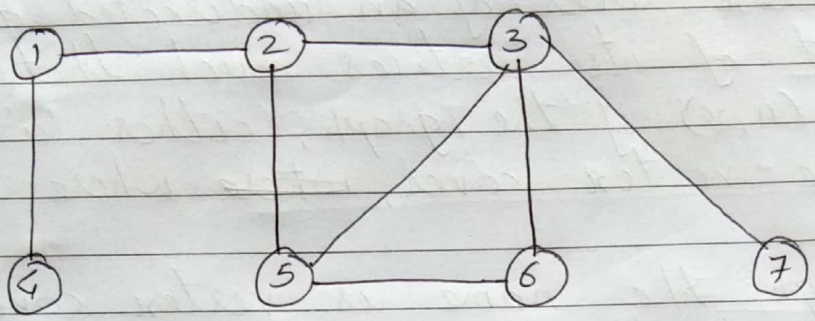
→ The vertex cover problem is a NP-complete problem ie. there is no polynomial-time solution for this, unless $P = NP$.

→ The simplest way of finding vertex cover of graph $G = (V, E)$ is to randomly select the edge $(u, v) \in E$ and then delete the adjacent edges of u and v.

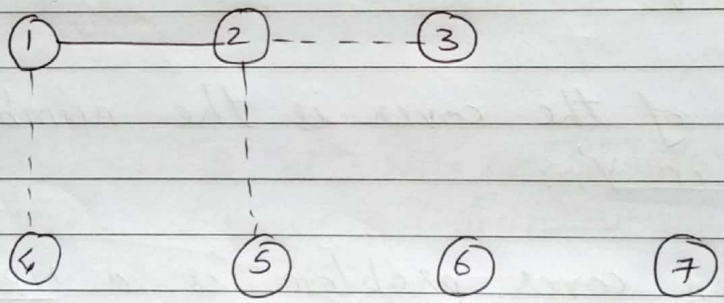
→ Repeat the procedure until the cover is found.

Smita Attarde

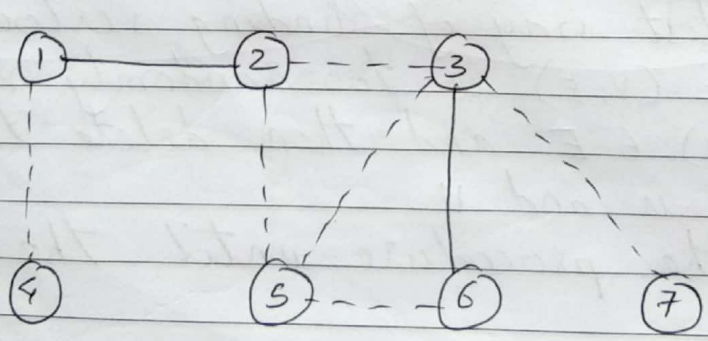
eg) Consider the graph:-



step 1:- Select any random edge
eg) select edge (1,2).
Remove all adjacent edges of vertices 1 and 2
Thus, $V' = \{1, 2\}$



step 2:- select edge (3,6)
$V' = \{1, 2, 3, 6\}$



→ Thus, all the edges are now covered.
→ Thus, the vertex cover for the given graph is $\{1, 2, 3, 6\}$