UNIT V
Introduction to randomization and approximation algorithm

Syllabus

Introduction –

- Randomization Algorithm
- Randomized hiring problem
- Randomized quick sort and its time complexity
- String matching algorithm
  - Rabin Karp algorithm for string matching

- Approximation algorithm
- Introduction Complexity classes
- P type problems
- Introduction to NP type problems
- Vertex covering
- Satisfiability problem
- Hamiltonian cycle problem

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 (or we randomly shuffle the array).

- Classification of Randomized algorithms
    - Las Vegas
    - Monte Carlo

RANDOMIZED ALGORITHMS

- **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. Output may be incorrect for certain probability. These algorithms have deterministic running time and it is generally easier to find out worst case time complexity.

Example to Understand Classification

- Example:
- **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.
  - 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

Applications and Scope

- Randomized algorithms have huge applications in Cryptography.
- Load Balancing.
- Data Structures: Hashing, Sorting, Searching, Order Statistics and Computational Geometry.
- Algebraic identities: Polynomial and matrix identity verification. Interactive proof systems.
- 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.
- etc…

HIRING PROBLEM

- We will now begin our investigation of randomized algorithms with a toy problem:
  - **You want to hire an office assistant from an employment agency.**
    - You want to interview candidates and determine if they are better than the current assistant and if so replace the current assistant.
    - You are going to eventually interview every candidate from a pool of n candidates.
    - You want to always have the best person for this job, so you will replace an assistant with a better one as soon as you are done the interview.
    - However, there is a cost to fire and then hire someone.
    - You want to know the expected price of following this strategy until all n candidates have been interviewed.

Steps involved in HIRING PROBLEM

Hire-Assistant(n)

```
1 best ← 0        candidate 0 is a least-qualified dummy candidate
2 for i ← 1 to n
3     do interview candidate i in random permutation
4         if candidate i is better than candidate best
5             then best ← i
6                 hire candidate i
```

Total Cost and Cost of Hiring

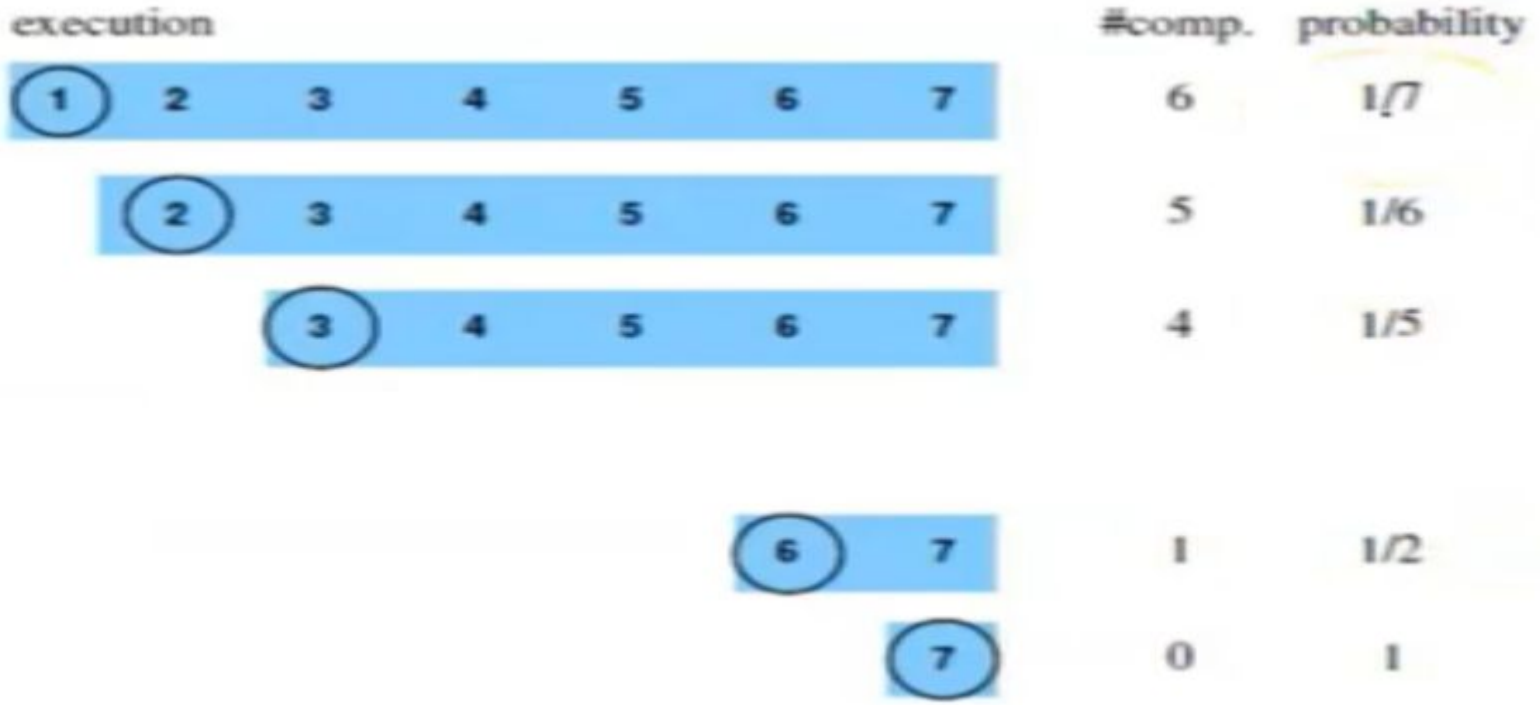
Total Cost and Cost of Hiring

- Interviewing has a low cost $\boldsymbol{ci}$.
- Hiring has a high cost $\boldsymbol{c_h}$.
- Let **n** be the number of candidates to be interviewed andlet **m** be the number of people hired.
- The total cost then goes as $O(\boldsymbol{n} * \boldsymbol{ci} + \boldsymbol{m} * \boldsymbol{ch})$
- The number of candidates is fixed so the part of thealgorithm we want to focus on is the $\boldsymbol{m} * \boldsymbol{ch}$term.
- This term governs the cost of hiring.

Worst-case Analysis

- In the worst case, everyone we interview turns out to be better than the person we currently have.
- In this case, the hiring cost for the algorithm will be O(n*ch).
- This bad situation presumably doesn't typically happen so it is interesting to ask what happens in the average case.

Time complexity – Execution - Comparison



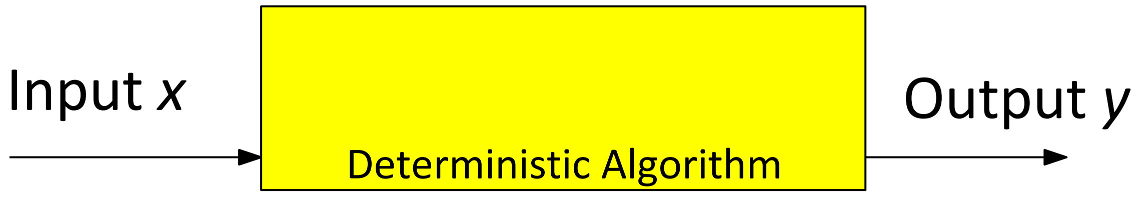
execution	#comp.	probability
1 2 3 4 5 6 7	6	1/7
2 3 4 5 6 7	5	1/6
3 4 5 6 7	4	1/5
6 7	1	1/2
7	0	1

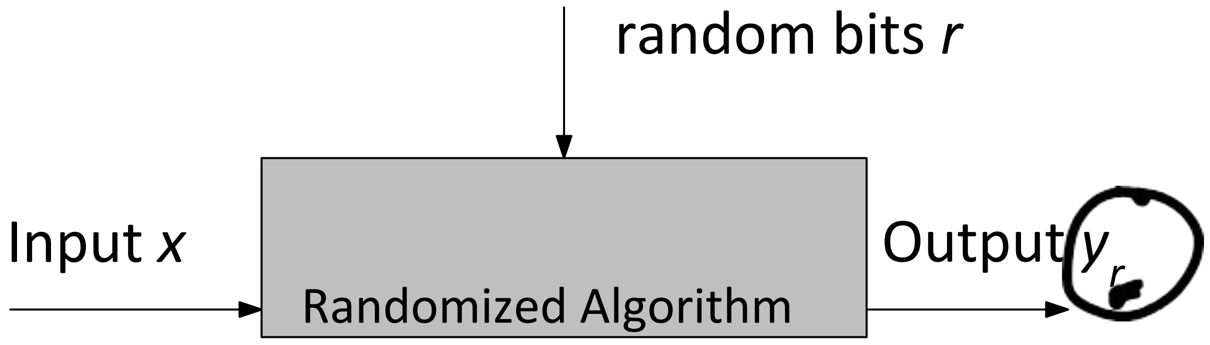
#comparisons: $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)$

Randomized quick sort

- In this article we will discuss how to implement QuickSort using random pivoting. In QuickSort we first partition the array in place such that all elements to the left of the pivot element are smaller, while all elements to the right of the pivot are greater that the pivot. Then we recursively call the same procedure for left and right subarrays.
- Unlike merge sort we don’t need to merge the two sorted arrays. Thus Quicksort requires lesser auxillary space than Merge Sort, which is why it is often preferred to Merge Sort.Using a randomly generated pivot we can further improve the time complexity of QuickSort.

Randomized Algorithms





Example: Randomized QuickSort

QuickSort

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
3. Recursively sort the subarrays, and concatenate them.

Randomized

1. Pick a pivot element **uniformly at random** from the array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
3. Recursively sort the subarrays, and concatenate them.

Example: Randomized QuickSort

Recall: **QuickSort** can take $\Omega(n^2)$ time to sort array of size n.

Theorem

Randomized **QuickSort** *sorts a given array of length* n *in* $O(n \log n)$ *expected time.*

Note: On *every* input randomized **QuickSort** takes $O(n \log n)$ time in expectation. On *every* input it may take $\Omega(n^2)$ time with some small probability.

Algorithm for Randomized quick sort

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if $p < r$
2      $q$ = RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, $q - 1$)
4      RANDOMIZED-QUICKSORT(A, $q + 1$, r)
```

```
RANDOMIZED-PARTITION(A, p, r)
1  $i$ = RANDOM(p, r)
2  exchange $A[r]$ with $A[i]$
3  return PARTITION(A, p, r)
```

Algorithm for Randomized quick sort

```
// 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 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)
```

Rabin-Karp algorithm

- Rabin-Karp algorithm is an algorithm used for searching/matching patterns in the text using a hash function.
- **How Rabin-Karp Algorithm Works?**
- A sequence of characters is taken and checked for the possibility of the presence of the required string. If the possibility is found then, character matching is performed.
- The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

Example using Rabin-Karp algorithm

Question:
Text = 31415926535, Pattern = 26 , find whether the pattern matching and also find how many spurious hits does the Rabin-Karp matcher encounters in Text T

Solution:

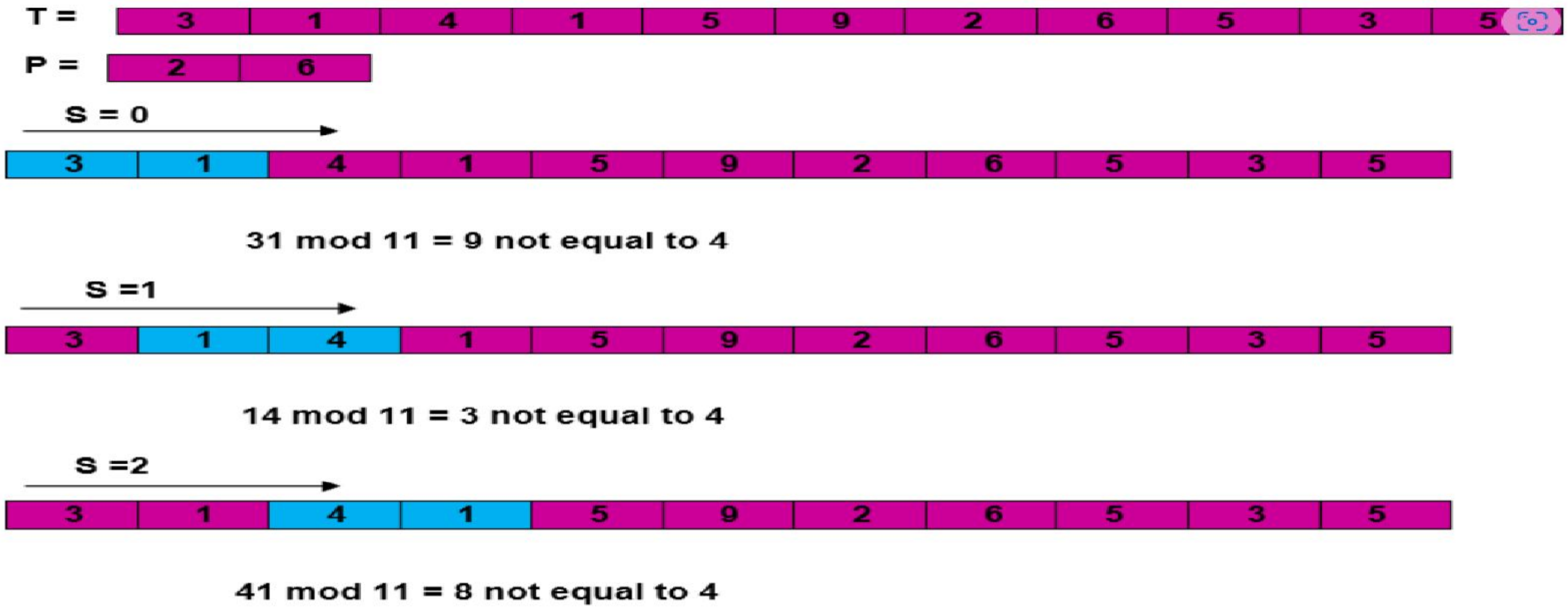
Example using Rabin-Karp algorithm

Question: Text = 31415926535, Pattern = 26 , find whether the pattern matching and also find how many spurious hits does the Rabin-Karp matcher encounters in Text T

Solution:

1. Here T.Length =11 so Q = 11
2. And P mod Q = 26 mod 11 = 4

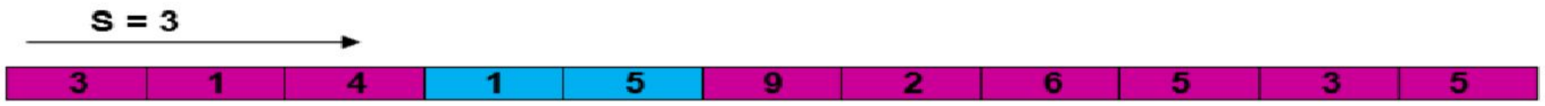
Now find the exact match of P mod Q...



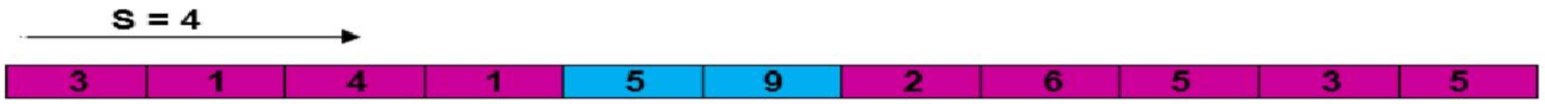
31 mod 11 = 9 not equal to 4

14 mod 11 = 3 not equal to 4

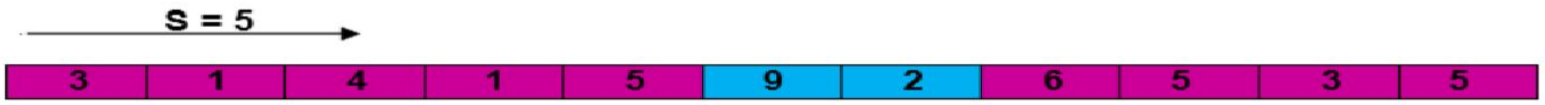
41 mod 11 = 8 not equal to 4



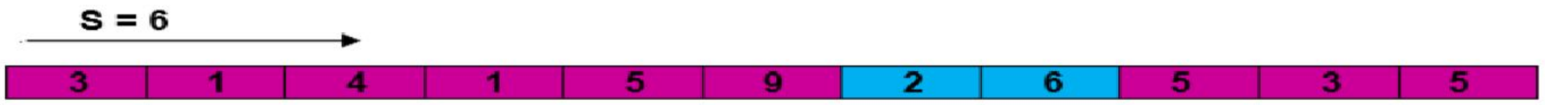
15 mod 11 = 4 equal to 4 SPURIOUS HIT



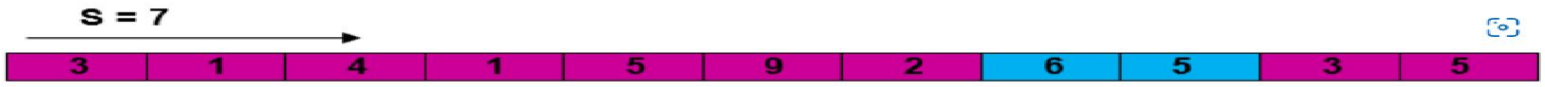
59 mod 11 = 4 equal to 4 SPURIOUS HIT



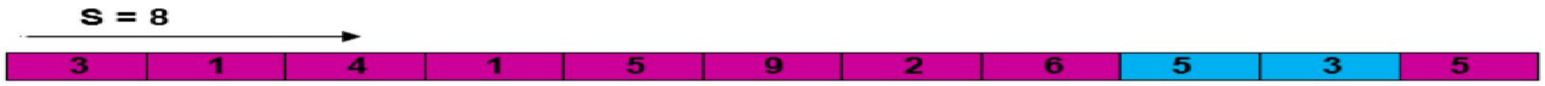
92 mod 11 = 4 equal to 4 SPURIOUS HIT



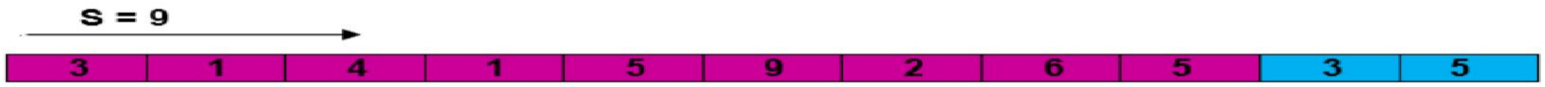
26 mod 11 = 4 EXACT MATCH



65 mod 11 = 10 not equal to 4



53 mod 11 = 9 not equal to 4

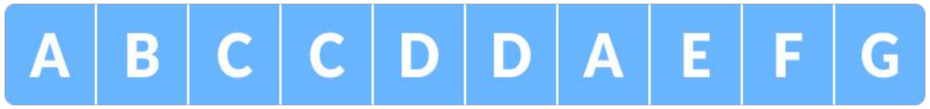


35 mod 11 = 2 not equal to 4

The Pattern occurs with shift 6.

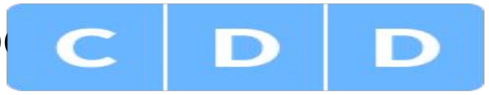
Example using Rabin-Karp algorithm

- Let the text be:



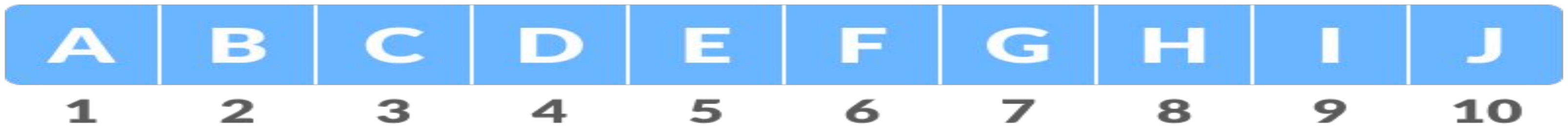
A	B	C	C	D	D	A	E	F	G

- And the string to be searched in the above text b



C	D	D

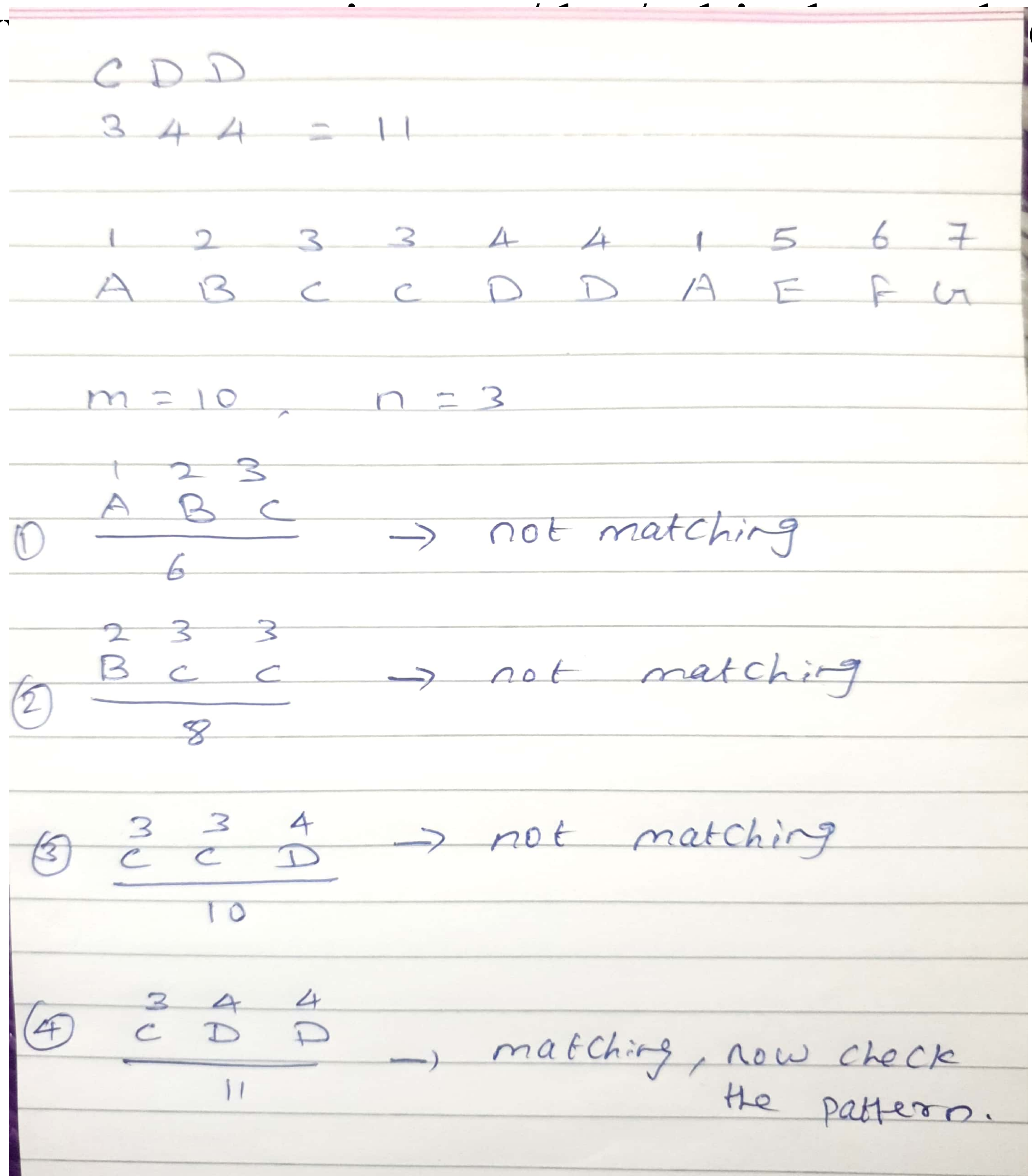
- Let us assign a numerical value(v)/weight for the characters we will be using in the problem. Here, we have taken first ten alphabets only (i.e. A to J).



A	B	C	D	E	F	G	H	I	J
1	2	3	4	5	6	7	8	9	10

- n be the length of the pattern and m be the length of the text. Here, m = 10 and n = 3.

- It is matching at 4th shift https://w orithm



C D D

3 4 4 = 11

1	2	3	3	4	4	1	5	6	7
A	B	C	C	D	D	A	E	F	G

m = 10 , n = 3

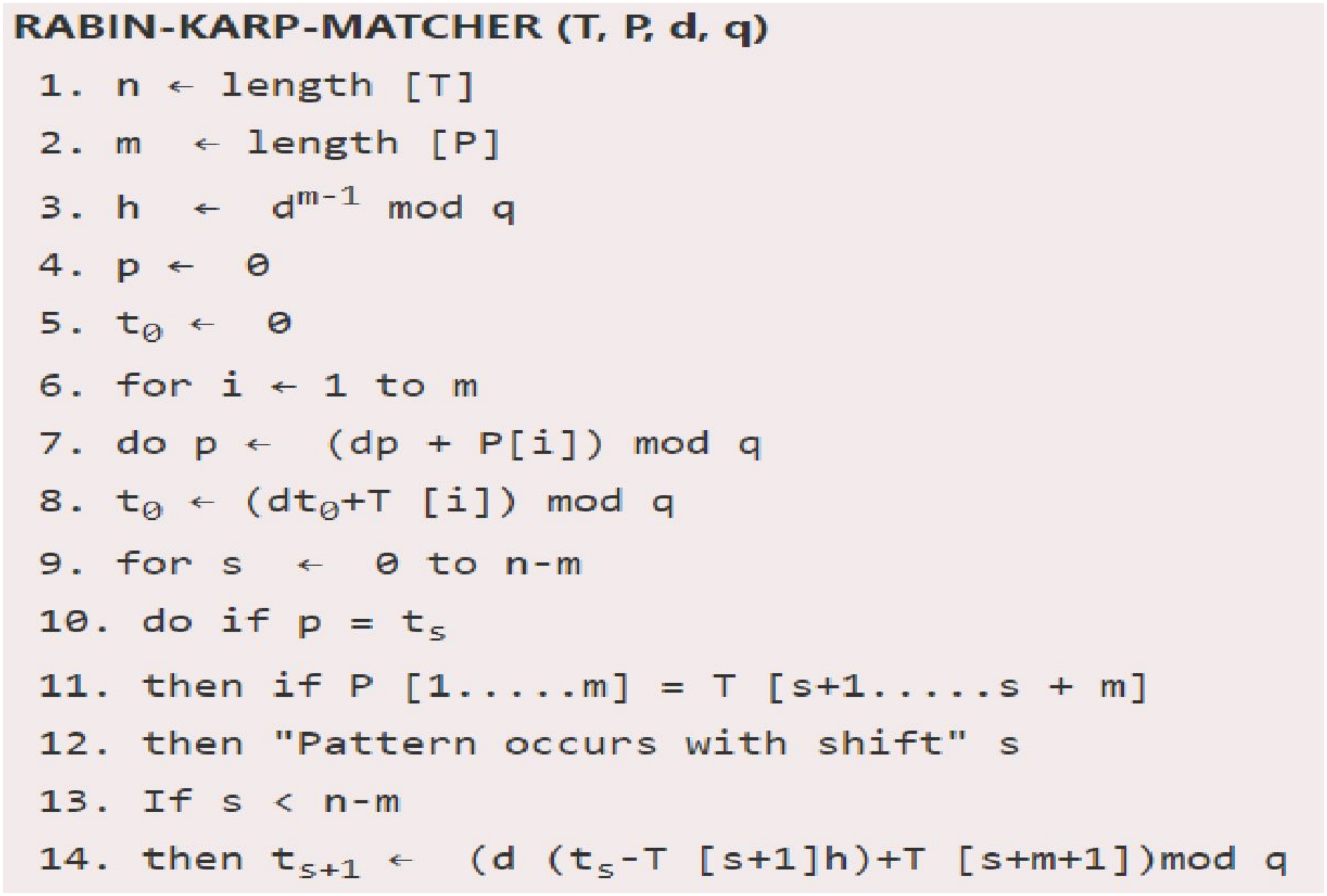
① $\underline{\overset{1}{A}\ \overset{2}{B}\ \overset{3}{C}}$ 6 → not matching

② $\underline{\overset{2}{B}\ \overset{3}{C}\ \overset{3}{C}}$ 8 → not matching

③ $\underline{\overset{3}{C}\ \overset{3}{C}\ \overset{4}{D}}$ 10 → not matching

④ $\underline{\overset{3}{C}\ \overset{4}{D}\ \overset{4}{D}}$ 11 → matching, now check the pattern.

Rabin-Karp algorithm



```
RABIN-KARP-MATCHER (T, P, d, q)
1. n ← length [T]
2. m ← length [P]
3. h ← $d^{m-1}$ mod q
4. p ← 0
5. $t_0$ ← 0
6. for i ← 1 to m
7.     do p ← (dp + P[i]) mod q
8.        $t_0$ ← ($dt_0$+T [i]) mod q
9. for s ← 0 to n-m
10.    do if p = $t_s$
11.          then if P [1.....m] = T [s+1.....s + m]
12.                  then "Pattern occurs with shift" s
13.       If s < n-m
14.          then $t_{s+1}$ ← (d ($t_s$-T [s+1]h)+T [s+m+1])mod q
```

Time Complexity - Rabin-Karp algorithm

- The running time of RABIN-KARP-MATCHER in the worst case scenario **O ((n-m+1)** m but it has a good average case running time.

Approximation algorithm

- An Approximate Algorithm is a way of approach that 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 the most polynomial time. Such algorithms are called approximation algorithm or heuristic algorithm.

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. **Sorting**
2. **Searching**

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.
- **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.

Example:

Shortest Path

Travelling sales man

Relation of P and NP classes

- **P contains in NP**
- **P=NP**
- Observe that P contains in NP. In other words, if we can solve a problem in polynomial time, we can indeed verify the solution in polynomial time. More formally, we do not need to see a certificate (there is no need to specify the vertex/intermediate of the specific path) to solve the problem; we can explain it in polynomial time anyway.
- However, it is not known whether P = NP. It seems you can verify and produce an output of the set of decision-based problems in NP classes in a polynomial time which is impossible because according to the definition of NP classes you can verify the solution within the polynomial time. So this relation can never be held.

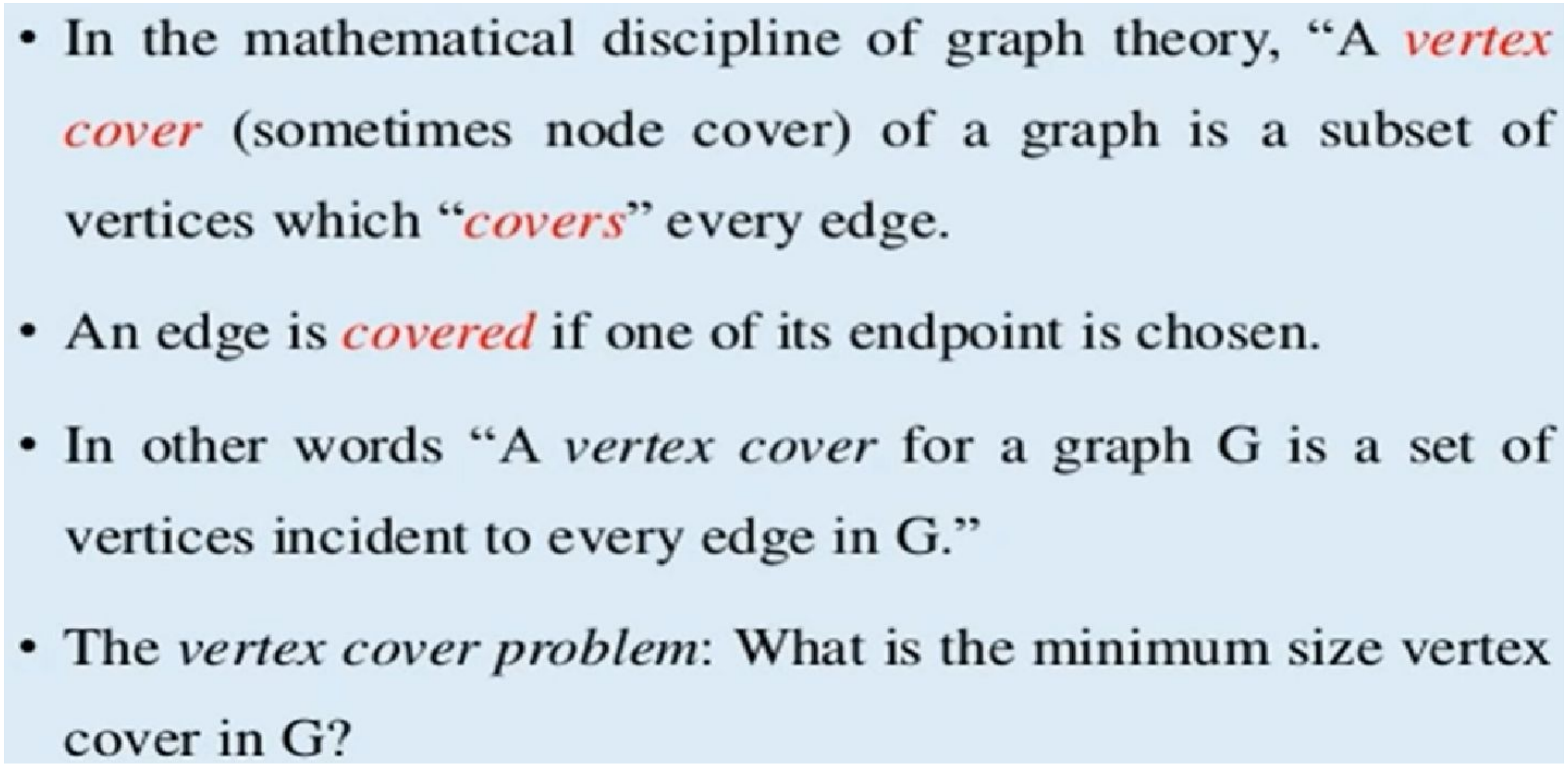
NP complete (NPC)

- A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hardest problems in NP.
- The class NP-complete (NPC) problems consist of a set of decision problems (a subset of class NP) that no one knows how to solve efficiently. But if there were a polynomial solution for even a single NP-complete problem, then every problem in NPC will be solvable in polynomial time.
- Some example problems include:

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

Vertex Cover

- A Vertex Cover of a graph G is a set of vertices such that each edge in G is incident to at least one of these vertices.
- The decision vertex-cover problem was proven NPC. Now, we want to solve the optimal version of the vertex cover problem, i.e., we want to find a minimum size vertex cover of a given graph. We call such vertex cover an optimal vertex cover C*.

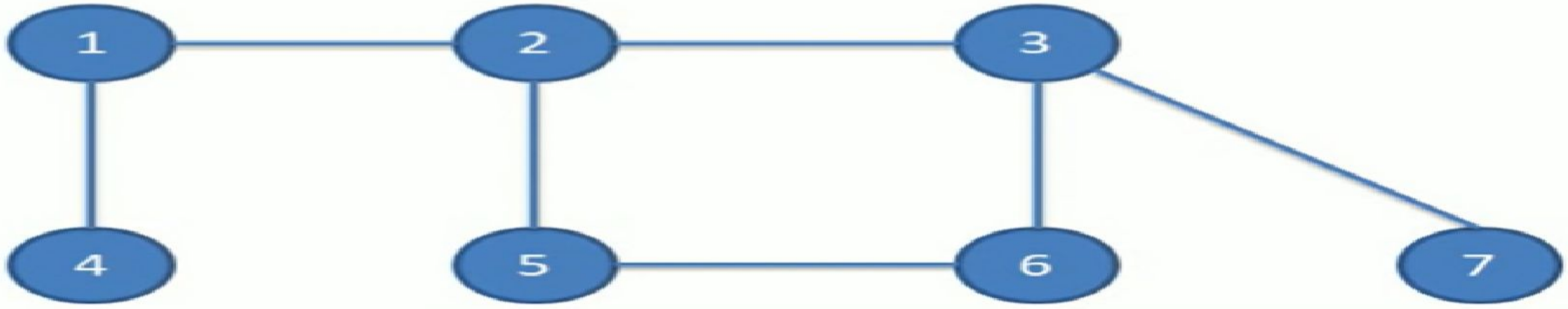


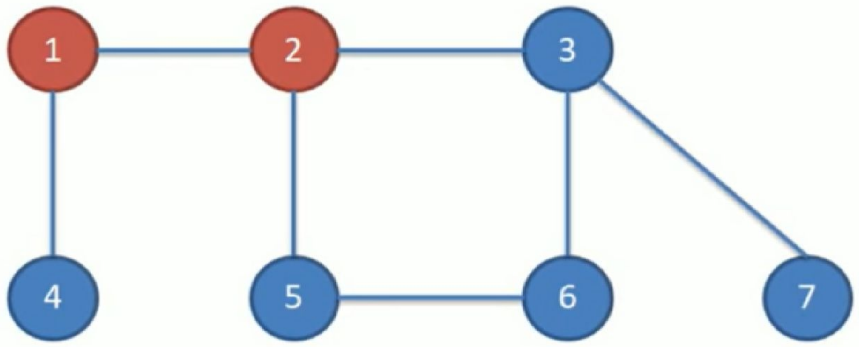
- In the mathematical discipline of graph theory, "A *vertex cover* (sometimes node cover) of a graph is a subset of vertices which "*covers*" every edge.
- An edge is *covered* if one of its endpoint is chosen.
- In other words "A *vertex cover* for a graph G is a set of vertices incident to every edge in G."
- The *vertex cover problem*: What is the minimum size vertex cover in G?

Vertex Cover

Vertex-cover problem

Given a undirected graph, find a vertex cover with minimum size.

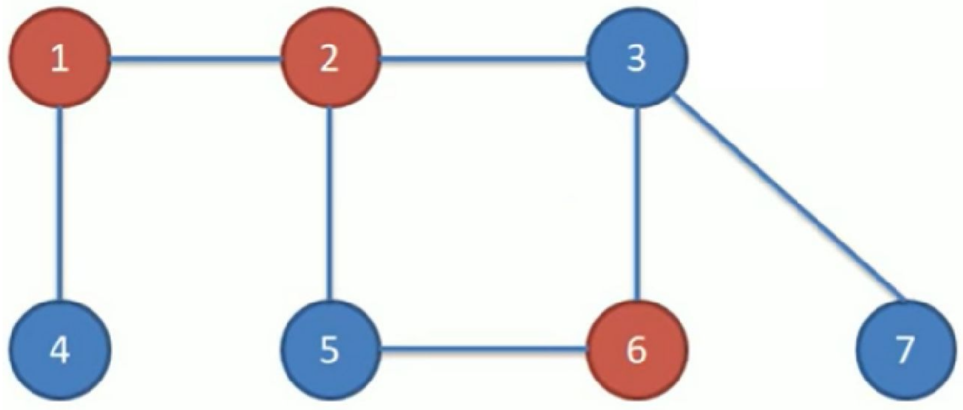




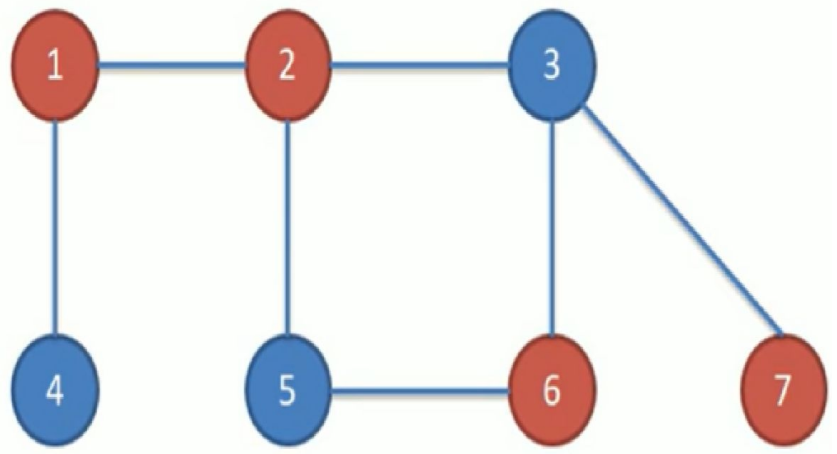
Are the red vertices a vertex-cover?

No. why?

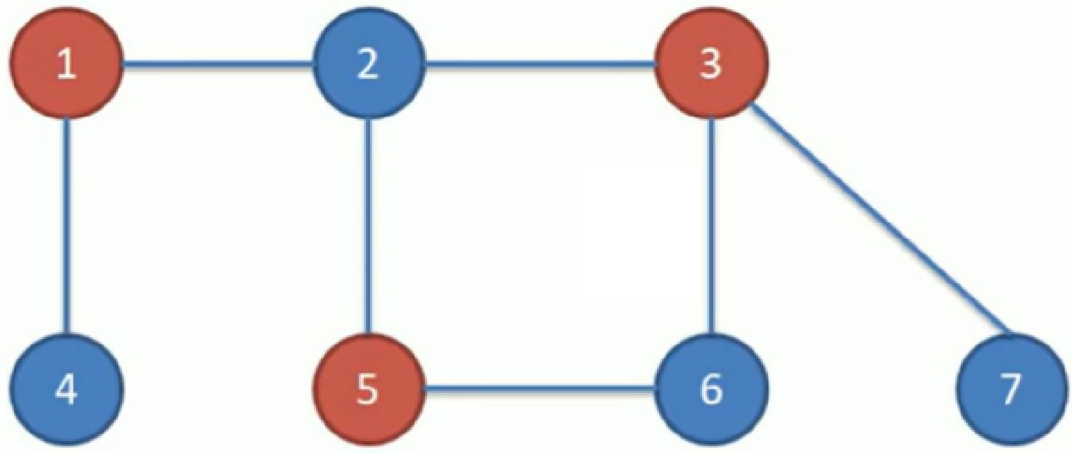
Edges (5, 6), (3, 6) and (3, 7) are not covered by it



Are the red vertices a vertex-cover?
No. why? Edge (3, 7) is not covered by it

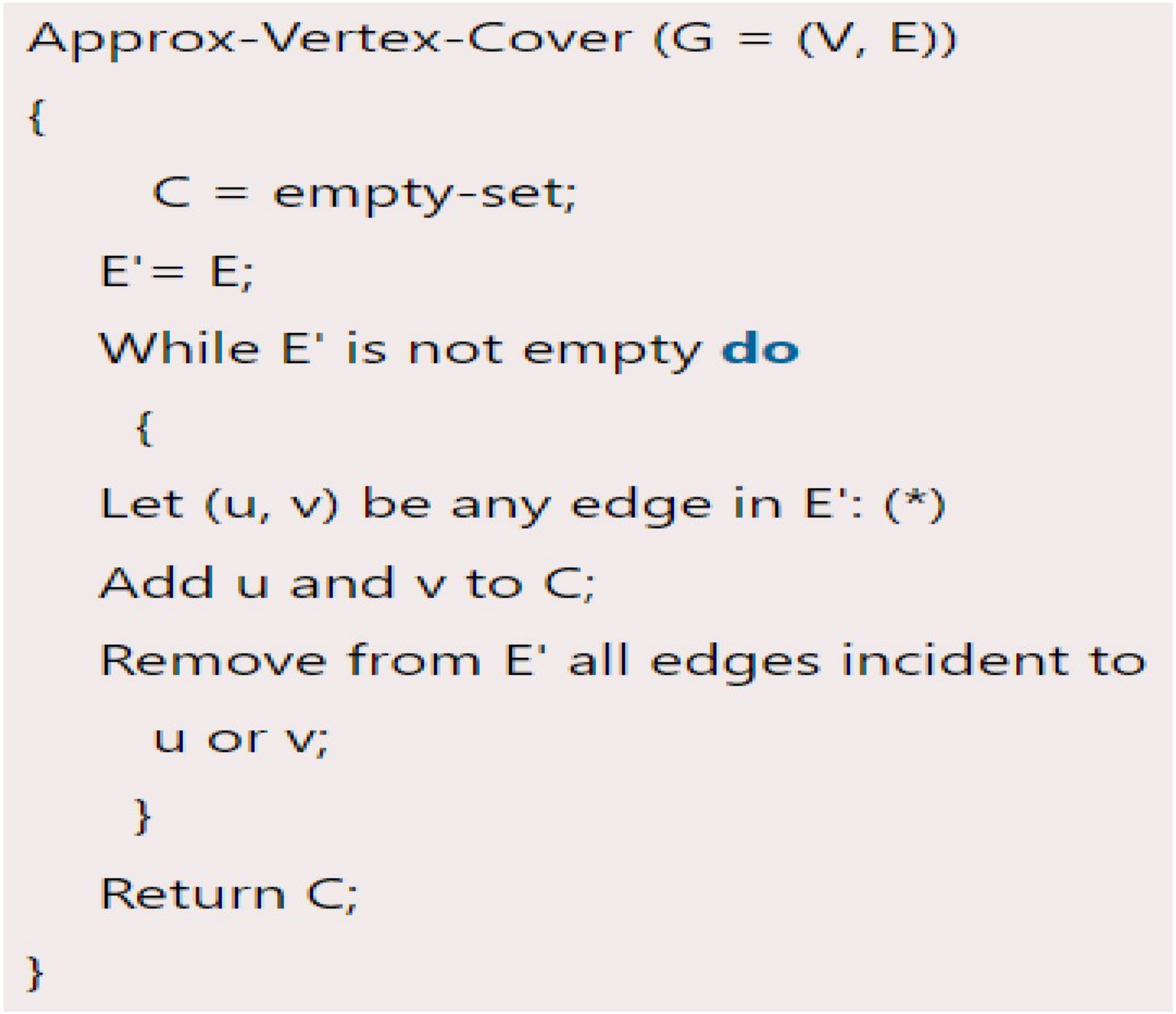


Are the red vertices a vertex-cover?
Yes What is the size? 4



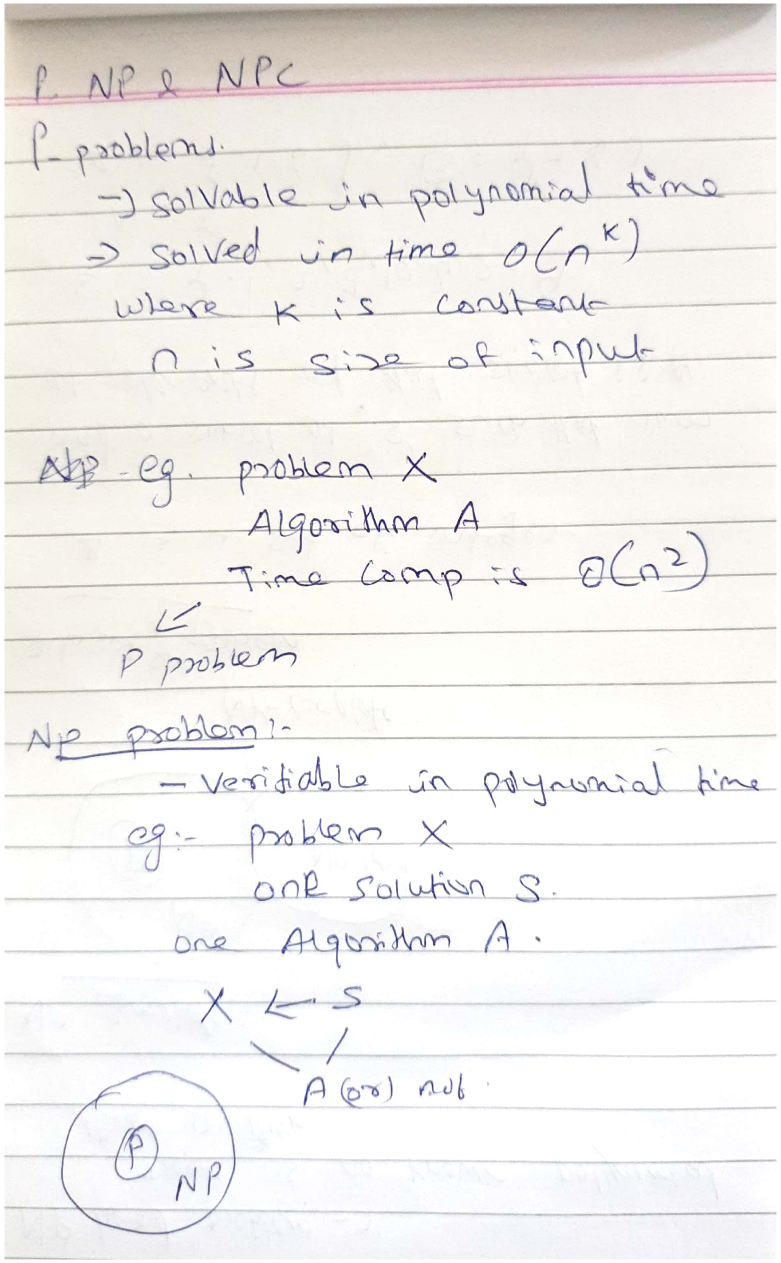
Are the red vertices a vertex-cover?
Yes What is the size? 3

Vertex Cover - Algorithm



```
Approx-Vertex-Cover (G = (V, E))
{
        C = empty-set;
    E'= E;
    While E' is not empty do
      {
    Let (u, v) be any edge in E': (*)
    Add u and v to C;
    Remove from E' all edges incident to
       u or v;
      }
    Return C;
}
```

P, NP and NP Complete



P, NP & NPC

P-problems.

-) solvable in polynomial time

-> solved in time $O(n^k)$

where k is constant

n is size of input

~~NP~~ eg. problem X

Algorithm A

Time Comp is $O(n^2)$

$\swarrow$

P problem

NP problem:-

- verifiable in polynomial time

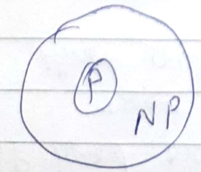
eg:- problem X

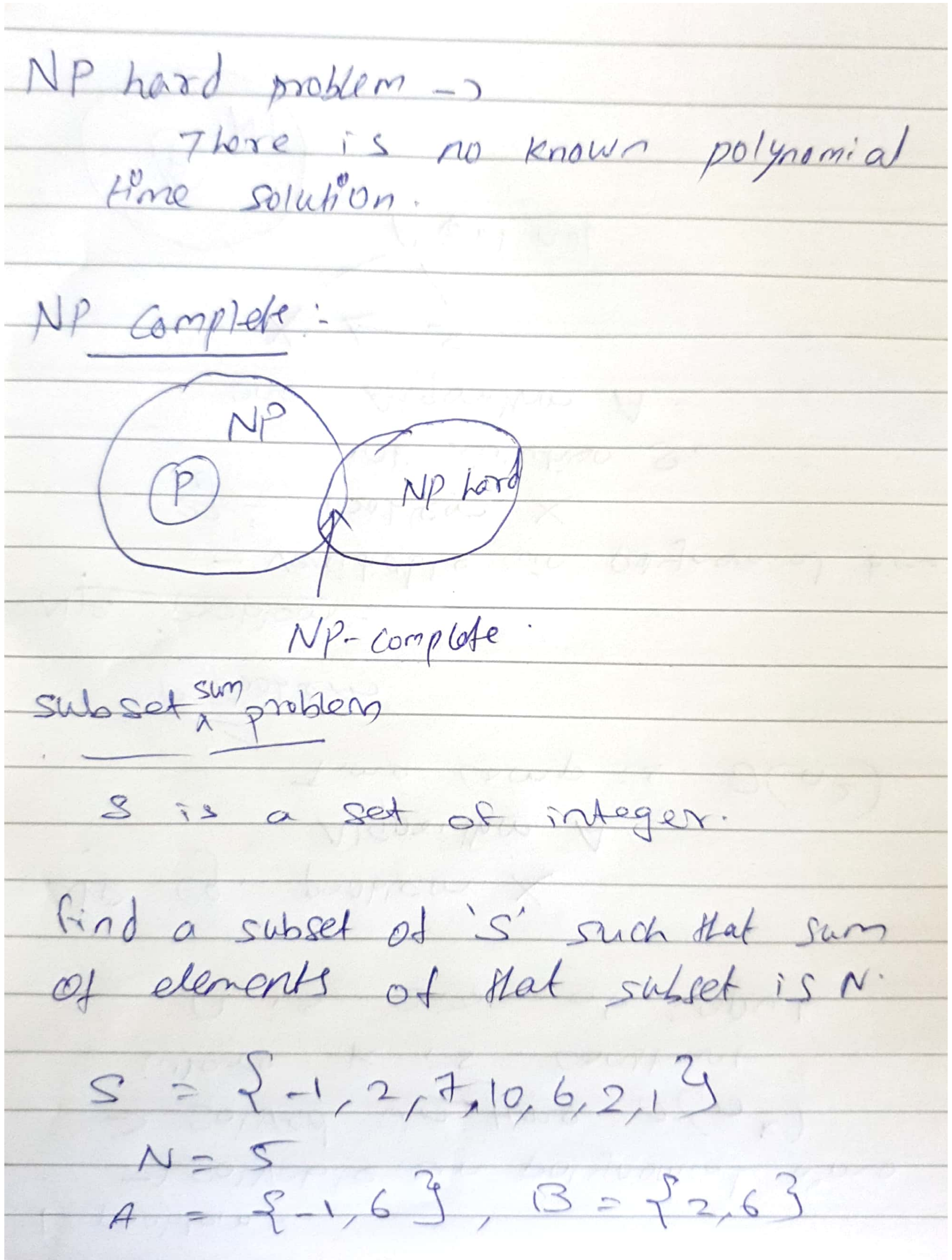
one solution S.

one Algorithm A.

$X \leftarrow S$

A (or) not.

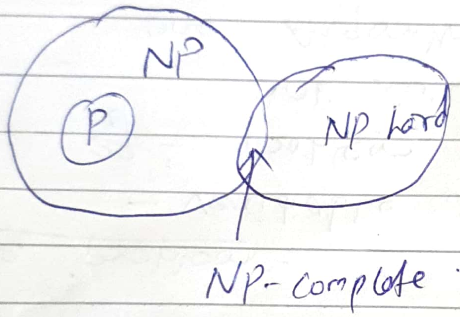




NP hard problem ->

There is no known polynomial time solution.

NP Complete:-



Subset $\overset{sum}{\wedge}$ problem

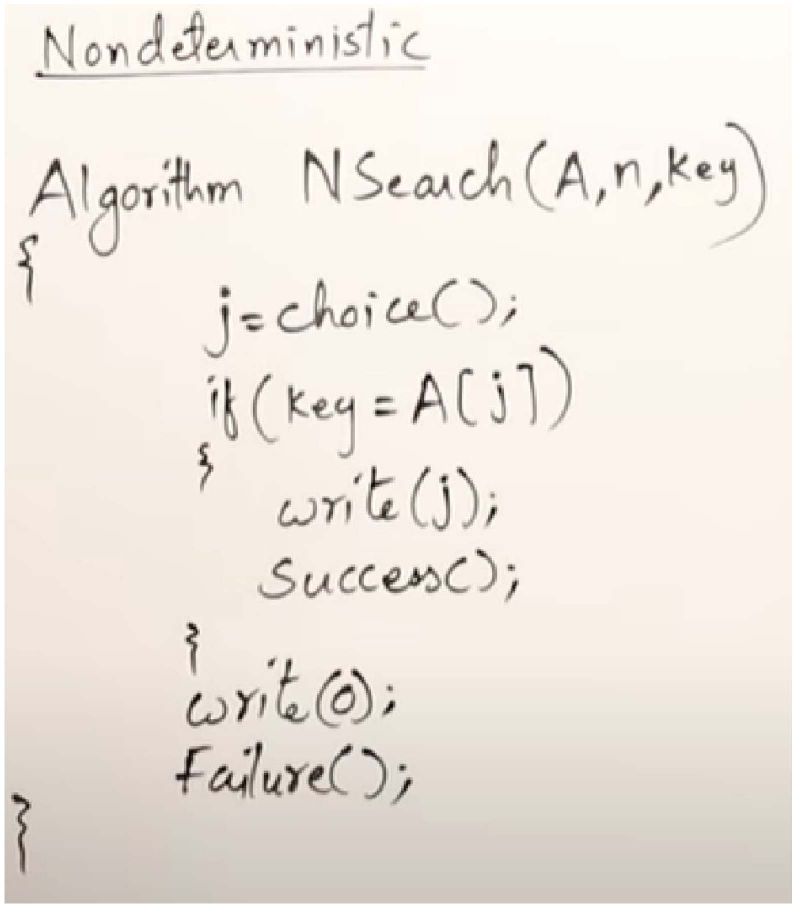
S is a set of integer.

Find a subset of 'S' such that sum of elements of that subset is N.

$S = \{-1, 2, 7, 10, 6, 2, 1\}$

$N = 5$

$A = \{-1, 6\}$, $B = \{2, 6\}$



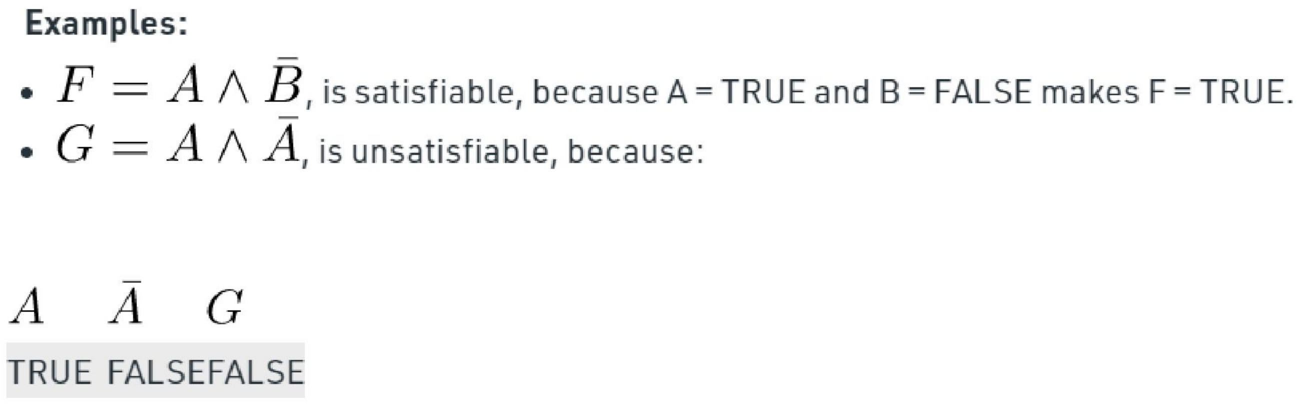
Nondeterministic

```
Algorithm NSearch(A,n,key)
{
        j=choice();
        if(key=A[j])
        {
            write(j);
            Success();
        }
        write(0);
        Failure();
}
```

Satisfiability problem

- **Boolean Satisfiability Problem**
- Boolean Satisfiability or simply **SAT** is the problem of determining if a Boolean formula is satisfiable or unsatisfiable.

- **Satisfiable :** If the Boolean variables can be assigned values such that the formula turns out to be TRUE, then we say that the formula is satisfiable.
- **Unsatisfiable :** If it is not possible to assign such values, then we say that the formula is unsatisfiable.



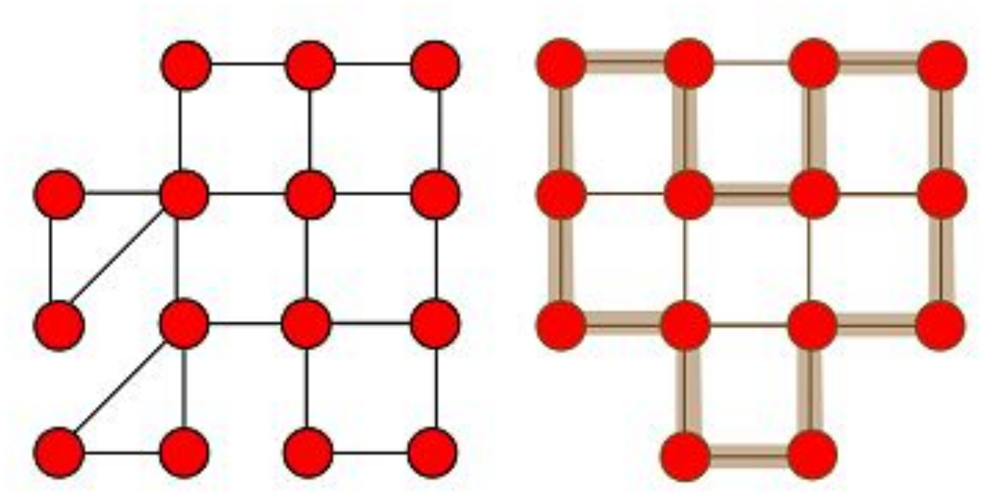
Examples:

- $F = A \wedge \bar{B}$, is satisfiable, because A = TRUE and B = FALSE makes F = TRUE.
- $G = A \wedge \bar{A}$, is unsatisfiable, because:

A	$\bar{A}$	G
TRUE	FALSE	FALSE

Hamiltonian cycle problem

- Consider the Hamiltonian cycle problem. Given an undirected graph G, does G have a cycle that visits each vertex exactly once? There is no known polynomial time algorithm for this dispute.


Non-Hamiltonian
Hamiltonian