NP-Completeness

A decision problem L is NP-Hard if

L' ≤p L for all L' ϵ NP.

**Definition:** L is NP-complete if

1. L ϵ NP and
2. L' ≤ p L for some known NP-complete problem L.' Given this formal definition, the complexity classes are:

**P:** is the set of decision problems that are solvable in polynomial time.

**NP:** is the set of decision problems that can be verified in polynomial time.

**NP-Hard:** L is NP-hard if for all L' ϵ NP, L' ≤p L. Thus if we can solve L in polynomial time, we can solve all NP problems in polynomial time.

**NP-Complete** L is NP-complete if

1. L ϵ NP and
2. L is NP-hard

If any NP-complete problem is solvable in polynomial time, then every NP-Complete problem is also solvable in polynomial time. Conversely, if we can prove that any NP-Complete problem cannot be solved in polynomial time, every NP-Complete problem cannot be solvable in polynomial time.

## Reductions

**Concept:** - If the solution of NPC problem does not exist then the conversion from one NPC problem to another NPC problem within the polynomial time. For this, you need the concept of reduction. If a solution of the one NPC problem exists within the polynomial time, then the rest of the problem can also give the solution in polynomial time (but it's hard to believe). For this, you need the concept of reduction.

**Example:** - Suppose there are two problems, **A** and **B**. You know that it is impossible to solve problem **A** in polynomial time. You want to prove that B cannot be solved in polynomial time. So you can convert the problem **A** into problem **B** in polynomial time.

## Example of NP-Complete problem

**NP problem:** - Suppose a DECISION-BASED problem is provided in which a set of inputs/high inputs you can get high output.

**Criteria to come either in NP-hard or NP-complete.**

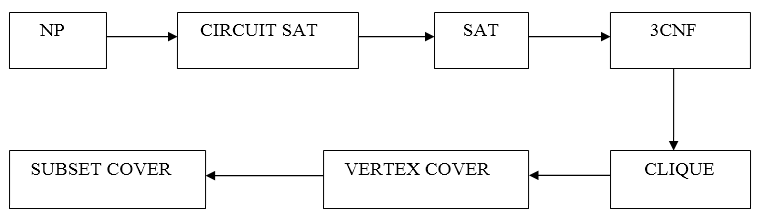
1. The point to be noted here, the output is already given, and you can verify the output/solution within the polynomial time but can't produce an output/solution in polynomial time.
2. Here we need the concept of reduction because when you can't produce an output of the problem according to the given input then in case you have to use an emphasis on the concept of reduction in which you can convert one problem into another problem.

#### Note1:- If you satisfy both points then your problem comes into the category of NP-complete class

#### Note2:- If you satisfy the only 2nd points then your problem comes into the category of NP-hard class

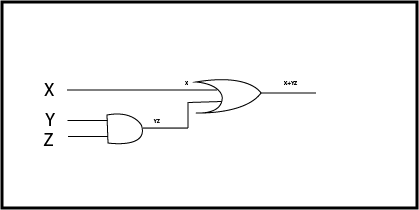
So according to the given decision-based NP problem, you can decide in the form of yes or no. If, yes then you have to do verify and convert into another problem via reduction concept. If you are being performed, both then decision-based NP problems are in NP compete.

**Here we will emphasize NPC.**



# CIRCUIT SAT

According to given decision-based NP problem, you can design the CIRCUIT and verify a given mentioned output also within the P time. The CIRCUIT is provided below:-



#### Note:- You can design a circuit and verified the mentioned output within Polynomial time but remember you can never predict the number of gates which produces the high output against the set of inputs/high inputs within a polynomial time. So you verified the production and conversion had been done within polynomial time. So you are in NPC.

## SAT (Satisfiability):-

A Boolean function is said to be SAT if the output for the given value of the input is true/high/1

F=X+YZ (Created a Boolean function by CIRCUIT SAT)

**These points you have to be performed for NPC**

1. CONCEPTS OF SAT
2. CIRCUIT SAT≤ρ SAT
3. SAT≤ρ CIRCUIT SAT
4. SAT ϵ NPC
5. **CONCEPT:** - A Boolean function is said to be SAT if the output for the given value of the input is true/high/1.
6. **CIRCUIT SAT≤ρ SAT:** - In this conversion, you have to convert CIRCUIT SAT into SAT within the polynomial time as we did it
7. **SAT≤ρ CIRCUIT SAT:** - For the sake of verification of an output you have to convert SAT into CIRCUIT SAT within the polynomial time, and through the CIRCUIT SAT you can get the verification of an output successfully
8. **SAT ϵ NPC:** - As you know very well, you can get the SAT through CIRCUIT SAT that comes from NP.

**Proof of NPC:** - Reduction has been successfully made within the polynomial time from CIRCUIT SAT TO SAT. Output has also been verified within the polynomial time as you did in the above conversation.

So concluded that SAT ϵ NPC.

# 3CNF SAT

**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) ᶺ (XvYvZ) ᶺ (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
5. **CONCEPT:** - In 3CNF SAT, you have at least 3 clauses, and in clauses, you will have almost 3 literals or constants.
6. **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)
7. **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
8. **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 that you can easily convert a Boolean function of SAT into 3CNF SAT and satisfied the concept of 3CNF SAT also within polynomial time through Reduction concept.
2. If you want to verify the output in 3CNF SAT then perform the Reduction and convert into SAT and CIRCUIT also to check the output

If you can achieve these two points that means 3CNF SAT also in NPC

# Clique

**To Prove:** - Clique is an NPC or not?

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

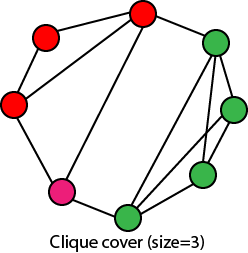
1. Clique
2. 3CNF ≤ρ Clique
3. Clique ≤ρ 3CNF≤SAT
4. Clique ϵ NP

### 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 verticesV1, V2...VK, 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.**



### 2)3CNF ≤ρ Clique

**Proof:**-For the successful conversion from 3CNF to Clique, you 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

### Clique ≤ρ 3CNF

**Proof:** - As you 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, 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**.

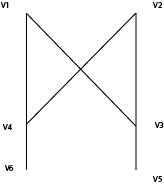
# Vertex Cover

1. Vertex Cover Definition
2. Vertex Cover ≤ρ Clique
3. Clique ≤ρ Vertex Cover
4. Vertex Cover ϵ NP

### 1) Vertex Cover:

**Definition:** - It represents a set of vertex or node in a graph G (V, E), which gives the connectivity of a complete graph

According to the graph G of vertex cover which you have created, **the size of Vertex Cover =2**



### 2) Vertex Cover ≤ρ Clique

In a graph G of Vertex Cover, you have N vertices which contain a Vertex Cover K. There must exist of Clique Size of size N-K in its complement.

**According to the graph G, you have  
Number of vertices=6  
Size of Clique=N-K=4**

You can also create the Clique by complimenting the graph G of Vertex Cover means in simpler form connect the vertices in Vertex Cover graph G through edges where edges don?t exist and remove all the existed edges

You will get the graph G with Clique Size=4

### 3) Clique ≤ρ Vertex Cover

Here through the Reduction process, you can get the Vertex Cover form Clique by just complimenting the Clique graph G within the polynomial time.

### 4) Vertex Cover ϵ NP

As you know very well, you can get the Vertex Cover through Clique and to convert the decision-based NP problem into Clique firstly you have to convert into 3CNF and 3CNF into SAT and SAT into CIRCUIT SAT that comes from NP.

**Proof of NPC:-**

1. Reduction from Clique to Vertex Cover has been made within the polynomial time. In the simpler form, you can convert into Vertex Cover from Clique within the polynomial time
2. And verification has also been done when you convert Vertex Cover to Clique and Clique to 3CNF and satisfy/verified the output within a polynomial time also, so it concluded that Reduction and Verification had been done in the polynomial time that means **Vertex Cover also comes in NPC**

# Subset Cover

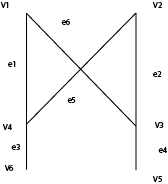
**To Prove:-**

1. Subset Cover
2. Vertex Cover ≤ρ Subset Cover
3. Subset Cover≤ρ Vertex Cover
4. Subset Cover ϵ NP

### 1) Subset Cover

**Definition:** - Number of a subset of edges after making the union for a get all the edges of the complete graph G, and that is called Subset Cover.

**According to the graph G, which you have created the size of Subset Cover=2**



1. v1{e1,e6}     v2{e5,e2}     v3{e2,e4,e6}     v4{e1,e3,e5}     v5{e4}     v6{e3}
2. v3Uv4= {e1, e2, e3, e4, e5, e6} complete set of edges after the union of vertices.

### 2) Vertex Cover ≤ρ Subset Cover

In a graph G of vertices N, if there exists a Vertex Cover of size k, then there must also exist a Subset Cover of size k even. If you can achieve after the Reduction from Vertex Cover to Subset Cover within a polynomial time, which means you did right.

### 3) Subset Cover ≤ρ Vertex Cover

Just for verification of the output perform the Reduction and create Clique and via an equation, N-K verifies the Clique also and through Clique you can quickly generate 3CNF and after solving the Boolean function of 3CNF in the polynomial time. You will get output. It means the output has been verified.

### 4) Subset Cover ϵ NP:-

**Proof:** - As you know very well, you can get the Subset-Cover through Vertex Cover and Vertex Cover through Clique and to convert the decision-based NP problem into Clique firstly you have to convert into3CNF and 3CNF into SAT and SAT into CIRCUIT SAT that comes from NP.

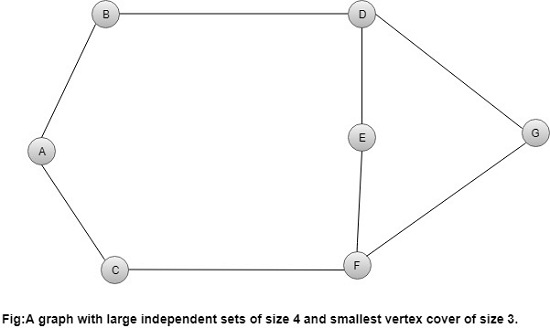
**Proof of NPC:-**

The Reduction has been successfully made within the polynomial time form Vertex Cover to Subset Cover

Output has also been verified within the polynomial time as you did in the above conversation so, concluded that **SUBSET COVER also comes in NPC**.

## Independent Set:

An independent set of a graph G = (V, E) is a subset V'⊆V of vertices such that every edge in E is incident on at most one vertex in V.' The independent-set problem is to find a largest-size independent set in G. It is not hard to find small independent sets, e.g., a small independent set is an individual node, but it is hard to find large independent sets.



# Approximate Algorithms

## Introduction:

An Approximate Algorithm is a way of approach **NP-COMPLETENESS** for the optimization problem. This technique does not guarantee the best solution. The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at the most polynomial time. Such algorithms are called approximation algorithm or heuristic algorithm.

* For the traveling salesperson problem, the optimization problem is to find the shortest cycle, and the approximation problem is to find a short cycle.
* For the vertex cover problem, the optimization problem is to find the vertex cover with fewest vertices, and the approximation problem is to find the vertex cover with few vertices.

## 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

Approximate Algorithm

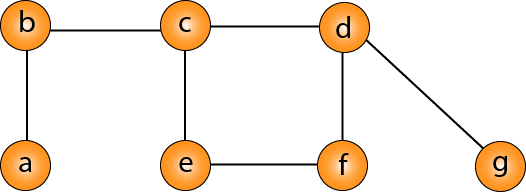
Intuitively, the approximation ratio measures how bad the approximate solution is distinguished with the optimal solution. A large (small) approximation ratio measures the solution is much worse than (more or less the same as) an optimal solution.

      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.

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.

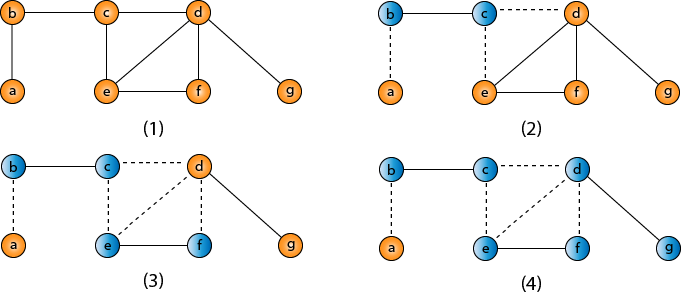
The decision vertex-cover problem was proven NPC. Now, we want to solve the optimal version of the vertex cover problem, i.e., we want to find a minimum size vertex cover of a given graph. We call such vertex cover an optimal vertex cover C\*.



An approximate algorithm for vertex cover:

1. Approx-Vertex-Cover (G = (V, E))
2. {
3. C = empty-set;
4. E'= E;
5. While E' is not empty **do**
6. {
7. Let (u, v) be any edge in E': (\*)
8. Add u and v to C;
9. Remove from E' all edges incident to
10. u or v;
11. }
12. Return C;
13. }

The idea is to take an edge (u, v) one by one, put both vertices to C, and remove all the edges incident to u or v. We carry on until all edges have been removed. C is a VC. But how good is C?



VC = {b, c, d, e, f, g}

# Traveling-salesman Problem

In the traveling salesman Problem, a salesman must visits n cities. We can say that salesman wishes to make a tour or Hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is a non-negative cost c (i, j) to travel from the city i to city j. The goal is to find a tour of minimum cost. We assume that every two cities are connected. Such problems are called Traveling-salesman problem (TSP).

We can model the cities as a complete graph of n vertices, where each vertex represents a city.

It can be shown that TSP is NPC.

If we assume the cost function c satisfies the triangle inequality, then we can use the following approximate algorithm.

## Triangle inequality

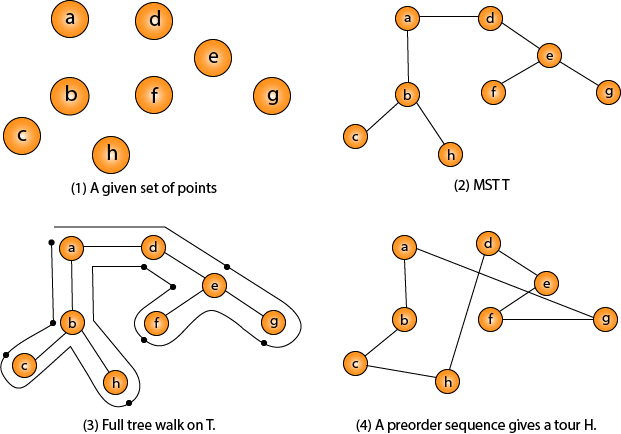
Let u, v, w be any three vertices, we have

Traveling-salesman Problem

One important observation to develop an approximate solution is if we remove an edge from H\*, the tour becomes a spanning tree.

1. Approx-TSP (G= (V, E))
2. {
3. 1. Compute a MST T of G;
4. 2. Select any vertex r is the root of the tree;
5. 3. Let L be the list of vertices visited in a preorder tree walk of T;
6. 4. Return the Hamiltonian cycle H that visits the vertices in the order L;
7. }

**Traveling-salesman Problem**



Intuitively, Approx-TSP first makes a full walk of MST T, which visits each edge exactly two times. To create a Hamiltonian cycle from the full walk, it bypasses some vertices (which corresponds to making a shortcut)

# String Matching Introduction

String Matching Algorithm is also called "String Searching Algorithm." This is a vital class of string algorithm is declared as "this is the method to find a place where one is several strings are found within the larger string."

Given a text array, T [1.....n], of n character and a pattern array, P [1......m], of m characters. The problems are to find an integer s, called **valid shift** where 0 ≤ s < n-m and T [s+1......s+m] = P [1......m]. In other words, to find even if P in T, i.e., where P is a substring of T. The item of P and T are character drawn from some finite alphabet such as {0, 1} or {A, B .....Z, a, b..... z}.

Given a string T [1......n], the **substrings** are represented as T [i......j] for some 0≤i ≤ j≤n-1, the string formed by the characters in T from index i to index j, inclusive. This process that a string is a substring of itself (take i = 0 and j =m).

The **proper substring** of string T [1......n] is T [1......j] for some 0<i ≤ j≤n-1. That is, we must have either i>0 or j < m-1.

Using these descriptions, we can say given any string T [1......n], the substrings are

1. T [i.....j] = T [i] T [i +1] T [i+2]......T [j] **for** some 0≤i ≤ j≤n-1.

And proper substrings are

1. T [i.....j] = T [i] T [i +1] T [i+2]......T [j] **for** some 0≤i ≤ j≤n-1.

#### Note: If i>j, then T [i.....j] is equal to the empty string or null, which has length zero.

## Algorithms used for String Matching:

There are different types of method is used to finding the string

1. The Naive String Matching Algorithm
2. The Rabin-Karp-Algorithm
3. Finite Automata
4. The Knuth-Morris-Pratt Algorithm
5. The Boyer-Moore Algorithm

# The Naive String Matching Algorithm

The naïve approach tests all the possible placement of Pattern P [1.......m] relative to text T [1......n]. We try shift s = 0, 1.......n-m, successively and for each shift s. Compare T [s+1.......s+m] to P [1......m].

The naïve algorithm finds all valid shifts using a loop that checks the condition P [1.......m] = T [s+1.......s+m] for each of the n - m +1 possible value of s.

**NAIVE-STRING-MATCHER (T, P)**

1. n ← length [T]

2. m ← length [P]

3. for s ← 0 to n -m

4. do if P [1.....m] = T [s + 1....s + m]

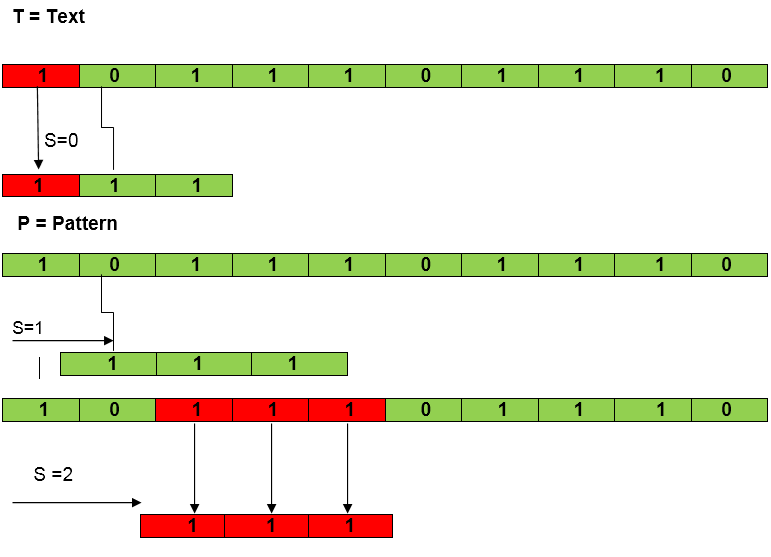
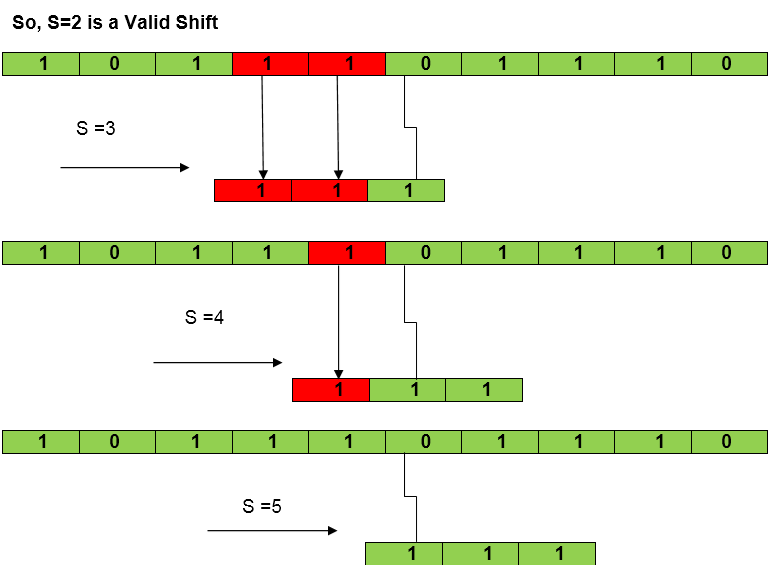
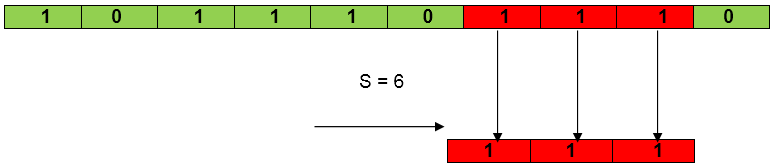
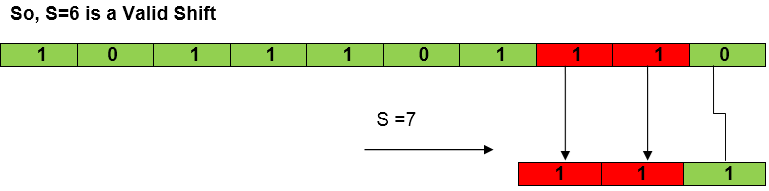
5. then print "Pattern occurs with shift" s

**Analysis:** This for loop from 3 to 5 executes for n-m + 1(we need at least m characters at the end) times and in iteration we are doing m comparisons. So the total complexity is O (n-m+1).

### Example:

1. Suppose T = 1011101110
2. P = 111
3. Find all the Valid Shift

**Solution:**

# The Rabin-Karp-Algorithm

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.

**RABIN-KARP-MATCHER (T, P, d, q)**

1. n ← length [T]

2. m ← length [P]

3. h ← dm-1 mod q

4. p ← 0

5. t0 ← 0

6. for i ← 1 to m

7. do p ← (dp + P[i]) mod q

8. t0 ← (dt0+T [i]) mod q

9. for s ← 0 to n-m

10. do if p = ts

11. then if P [1.....m] = T [s+1.....s + m]

12. then "Pattern occurs with shift" s

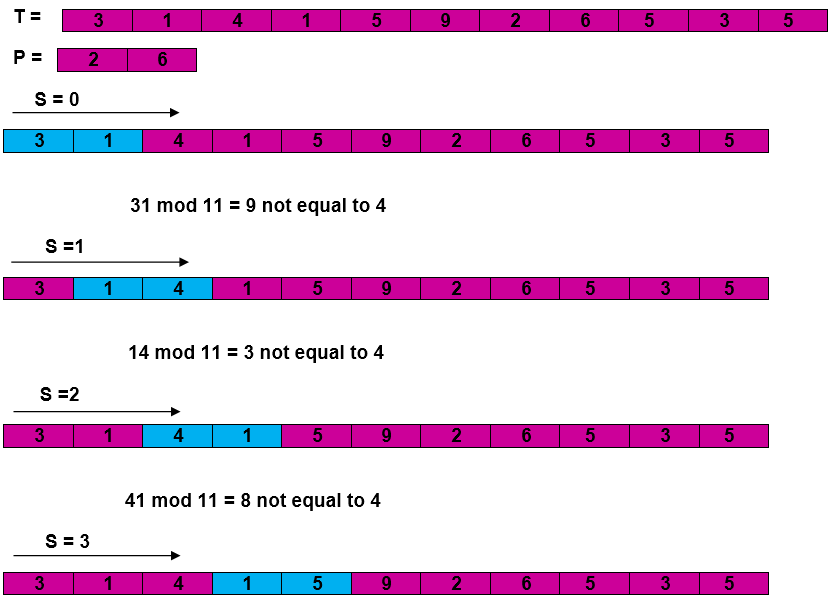
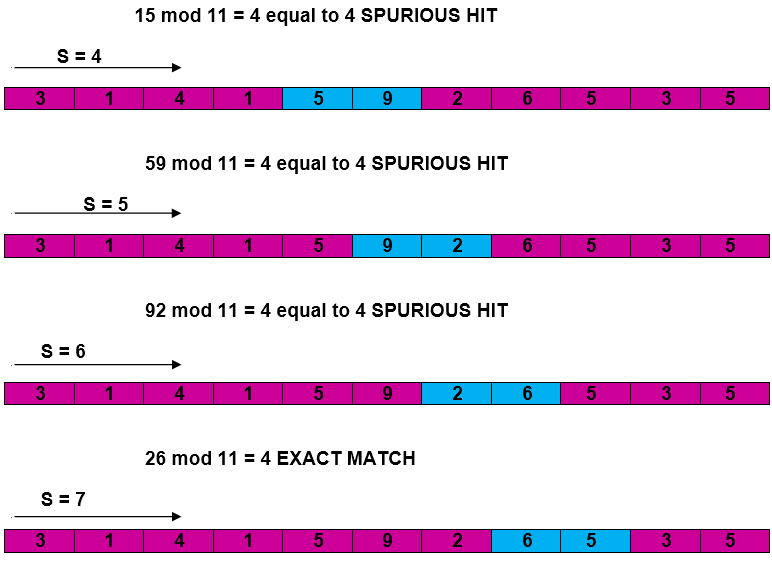
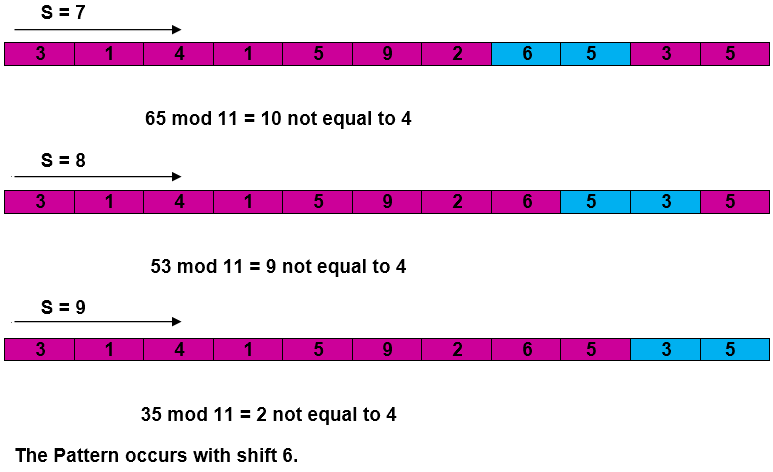
13. If s < n-m

14. then ts+1 ← (d (ts-T [s+1]h)+T [s+m+1])mod q

**Example:** For string matching, working module q = 11, how many spurious hits does the Rabin-Karp matcher encounters in Text T = 31415926535.......

1. T = 31415926535.......
2. P = 26
3. Here T.Length =11 so Q = 11
4. And P mod Q = 26 mod 11 = 4
5. Now find the exact match of P mod Q...

**Solution:**

## Complexity:

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. If the expected number of strong shifts is small **O (1)** and prime q is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time **O (n+m)** plus the time to require to process spurious hits.

# String Matching with Finite Automata

The string-matching automaton is a very useful tool which is used in string matching algorithm. It examines every character in the text exactly once and reports all the valid shifts in O (n) time. The goal of string matching is to find the location of specific text pattern within the larger body of text (a sentence, a paragraph, a book, etc.)

## Finite Automata:

A finite automaton **M** is a 5-tuple **(Q, q0,A,∑δ)**, where

* Q is a finite set of **states**,
* q0 ∈ Q is the **start state**,
* A ⊆ Q is a notable set of **accepting states**,
* ∑ is a **finite input alphabet**,
* δ is a function from **Q x ∑** into **Q** called the **transition function** of **M**.

The finite automaton starts in state **q0** and reads the characters of its input string one at a time. If the automaton is in state q and reads input character a, it moves from state q to state δ (q, a). Whenever its current state q is a member of A, the machine M has accepted the string read so far. An input that is not allowed is **rejected**.

A finite automaton M induces a function ∅ called the called the **final-state function**, from ∑\* to Q such that ∅(w) is the state M ends up in after scanning the string w. Thus, M accepts a string w if and only if ∅(w) ∈ A.

The function f is defined as

∅ (∈)=q0

∅ (wa) = δ ((∅ (w), a) for w ∈ ∑\*,a∈ ∑)

**FINITE- AUTOMATON-MATCHER (T,δ, m),**

1. n ← length [T]

2. q ← 0

3. for i ← 1 to n

4. do q ← δ (q, T[i])

5. If q =m

6. then s←i-m

7. print "Pattern occurs with shift s" s

The primary loop structure of FINITE- AUTOMATON-MATCHER implies that its running time on a text string of length n is O (n).

**Computing the Transition Function:** The following procedure computes the transition function δ from given pattern P [1......m]

**COMPUTE-TRANSITION-FUNCTION (P, ∑)**

1. m ← length [P]

2. for q ← 0 to m

3. do for each character a ∈ ∑\*

4. do k ← min (m+1, q+2)

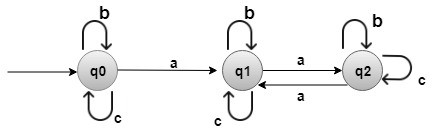
5. repeat k←k-1

6. Until

7. δ(q,a)←k

8. Return δ

**Example:** Suppose a finite automaton which accepts even number of a's where ∑ = {a, b, c}



**Solution:**

q0 is the initial state.

