P AND NP PROBLEMS

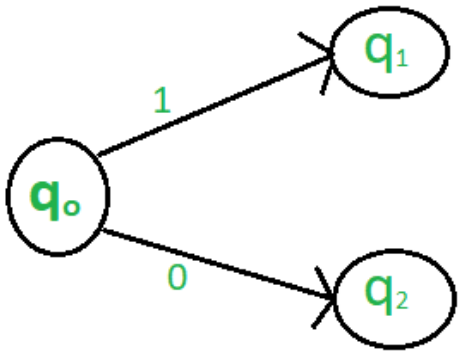
Basic Concepts::

Deterministic Algorithm::

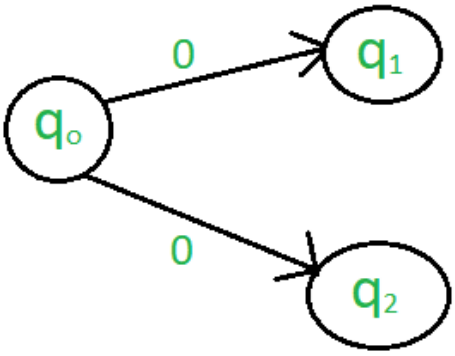
- For a given particular input, the computer will always produce the same output going through the same states.
- i.e., For a particular input the computer will give always produces the same output.
- Can solve the problem in polynomial time.
- Can determine the next step of execution.

Non-Deterministic Algorithm::

- For the same input, the compiler may produce different output in different runs.
- In fact non-deterministic algorithms can't solve the problem in polynomial time and can't determine what is the next step.
- The non-deterministic algorithms can show different behaviors for the same input on different execution.



Deterministic Algorithm



Non-Deterministic Algorithm

- To implement a non-deterministic algorithm, we have a couple of languages like Prolog but these don't have standard programming language operators and these operators are not a part of any standard programming languages.

Some of the terms related to the non-deterministic algorithm are defined below:

- **choice(X) :** chooses any value randomly from the set X.
- **failure() :** denotes the unsuccessful solution.
- **success() :** Solution is successful and current thread terminates.

Example :

Problem Statement : *Search an element x on A[1:n] where n>=1, on successful search return j if a[j] is equals to x otherwise return 0.*

Non-deterministic Algorithm for this problem :

```
1.j= choice(a, n)
2.if(A[j]==x) then
  {
    write(j);
    success();
  }
3.write(0);
4.  failure();
```

Deterministic Algorithm	**Non-deterministic Algorithm**
➢ For a particular input the computer will give always same output.	➢ For a particular input the computer will give different output on different execution.
➢ Can solve the problem in polynomial time.	➢ Can't solve the problem in polynomial time.
➢ Can determine the next step of execution.	➢ Cannot determine the next step of execution due to more than one path the algorithm can take.

Based on difficulty level, problems can be categorized into 2 types

1 Analytically hard problems.

2 Computationally hard problems.

1 Analytically hard problems:

- Either do not have a proper mathematical formulation or require enormous amount of intelligence. Ex:: the game of chess.

2 Computationally hard problems:

- Cannot be solved using available algorithm. Computational complexity focusses on these problems. These are categorized into groups called complexity classes.
- Complexity classes are a set of problems that have similar run times of growth.
- These are categorized into 2. A) Class P B) Class NP

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.

3. 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.

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:

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 problems are again classified into 2 types. I) NP Hard II) NP Complete.

NP-hard class

- An NP-hard problem is at least as hard as the hardest problem in NP and it is the class of the problems such that every problem in NP reduces to NP-hard.

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 hardest problems in NP.
- In mathematics, if all statements of a system can be explained, it is said to be complete.
- NP complete problems are powerful problems that can solve all problems of class NP and hence these problems are said to be NP complete.

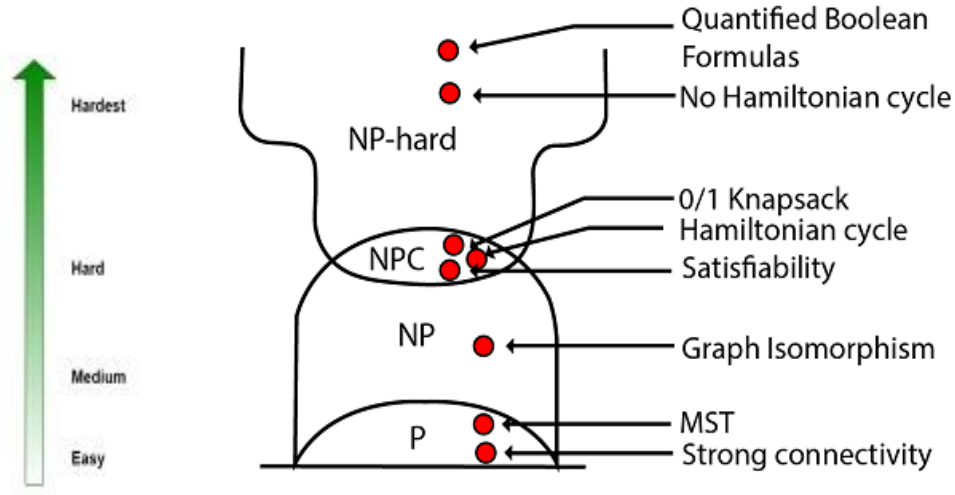
Features:

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.**

Complexity Class	Characteristic feature
P	Easily solvable in polynomial time.
NP	Yes, answers can be checked in polynomial time.
NP-hard	All NP-hard problems are not in NP and it takes a long time to check them.
NP-complete	A problem that is NP and NP-hard is NP-complete.



Cook's theorem

- In computational complexity theory, the Cook–Levin theorem, also known as Cook's theorem, states that the Boolean satisfiability problem is NP-complete.
- That is, it is in NP, and any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the Boolean satisfiability problem.
- *Stephen Arthur Cook and L.A. Levin in 1973 independently proved that the* ***satisfiability problem(SAT)*** *is NP-complete.*

SAT ***is a significant problem and can be stated as follows:***

- Given a Boolean expression **F** having **n** variables $x_1, x_2, \ldots, x_n$, and Boolean operators, is it possible to have an assignment for variables true or false such that binary expression **F** is true?
- This problem is also known as the **formula – SAT**.
- An **SAT(formula-SAT or simply SAT)** takes a Boolean expression F and checks whether the given expression(or formula) is satisfiable.
- A Boolean expression is said to be satisfactory for some valid assignments of variables if the evaluation comes to be true.

Some important terminologies of a Boolean expression:

- **Boolean variable:** A variable, say **x**, that can have only two values, true or false, is called a boolean variable.

- **Literal:** A literal can be a logical variable, say **x**, or the negation of it, that is **x** or $\bar{x}$; **x** is called a positive literal, and $\bar{x}$ is called the negative literal.

- **Clause:** A sequence of variables$(x_1, x_2, \ldots, x_n)$ that can be separated by a logical **OR** operator is called a clause. For example, $(x_1 \vee x_2 \vee x_3)$ is a clause of three literals.

- **Expressions:** One can combine all the preceding clauses using a Boolean operator to form an expression.

- **CNF form:** An expression is in CNF form(conjunctive normal form) if the set of clauses are separated by an **AND (^),** operator, while the literals are connected by an **OR (v)** operator.

  The following is an example of an expression in the CNF form:

  $$f = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee \bar{x}_3 \vee x_2)$$

- **3 – CNF:** An expression is said to be in 3-CNF if it is in the conjunctive normal form, and every clause has exact three literals.

  - Thus, an **SAT** is one of the toughest problems, as there is no known algorithm other than the brute force approach.
  - A brute force algorithm would be an exponential-time algorithm, as 2^n possible assignments need to be tried to check whether the given Boolean expression is true or not. Stephen Cook and Leonid Levin proved that the **SAT** is NP-complete.

3-CNF-SAT(3-SAT)

- This problem contains the expression is in a conjunctive normal form and that every clause should contain exactly three literals.
- This problem is also about assigning **n** assignments of truth values to **n** variables of the Boolean expression such that the output of the expression is true.
- In simple words, given an expression in 3-CNF, a 3-SAT problem is to check whether the given expression is satisfiable.
- If we consider,

$$f = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3)$$

it is the Boolean satisfiability problem of 3 literals.. So every literal can take either 0 or 1 as input Values.

i.e., $x_1 =$ 0 or 1; $x_2 = 0$ or 1; $x_3 = 0$ or 1.. we are having 3 literals and every literal with 2 inputs . So we can have a total of $2^n = 2^3 = 8$ possible outcomes.

For inputs $x_1 = 0$; $x_2 = 0$; $x_3 = 0 \Rightarrow f = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3)$

$\Rightarrow f = (0 \vee \bar{0} \vee 0) \wedge (0 \vee 0 \vee \bar{0}) \Rightarrow 1 \wedge 1 = 1$ *i.e, True..*

$x_1 = 1$; $x_2 = 1$; $x_3 = 1 \Rightarrow f = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3)$

$\Rightarrow f = (1 \vee \bar{1} \vee 1) \wedge (1 \vee 1 \vee \bar{1}) \Rightarrow 1 \wedge 1 = 1$ *i.e, True..*

Likewise, for all the values , the solution will be true..

The Boolean expression satisfiable for all the valid assignments of variables and the evaluation comes to be true.*Hence, the theorem is proved..*

String Matching

Introduction::

- String matching algorithms have greatly influenced computer science and play an essential role in various real-world problems.
- It helps in performing time-efficient tasks in multiple domains.
- These algorithms are useful in the case of searching a string within another string.
- String matching is also used in the Database schema, NetworkSystems.
- String Matching Algorithms can broadly be classified into two types of algorithms –
    A. Exact String Matching Algorithms
    B. Approximate String Matching Algorithms

Exact String Matching Algorithms:

- Exact string matching algorithms is to find one, several, or all occurrences of a defined string (pattern) in a large string (text or sequences) such that each matching is perfect.
- All alphabets of patterns must be matched to corresponding matched subsequence.
- These are further classified into four categories:

1. Algorithms based on character comparison:
    - **Naive Algorithm**: It slides the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.
    - **KMP (Knuth Morris Pratt) Algorithm:** The idea is whenever a mismatch is detected, we already know some of the characters in the text of the next window. So, we take advantage of this information to avoid matching the characters that we know will anyway match.
    - **Boyer Moore Algorithm:** This algorithm uses best heuristics of Naive and KMP algorithm and starts matching from the last character of the pattern.
    - **Using the Trie data structure**: It is used as an efficient information retrieval data structure. It stores the keys in form of a balanced BST.

2. Deterministic Finite Automaton (DFA) method:
   - **Automaton Matcher Algorithm**: It starts from the first state of the automata and the first character of the text. At every step, it considers next character of text, and look for the next state in the built finite automata and move to a new state.

3. Algorithms based on Bit (parallelism method):
   - **Aho-Corasick Algorithm:** It finds all words in O(n + m + z) time where n is the length of text and m be the total number characters in all words and z is total number of occurrences of words in text. This algorithm forms the basis of the original Unix command fgrep.

4. Hashing-string matching algorithms:
   - **Rabin Karp Algorithm:** It matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters.

Approximate String Matching Algorithms:

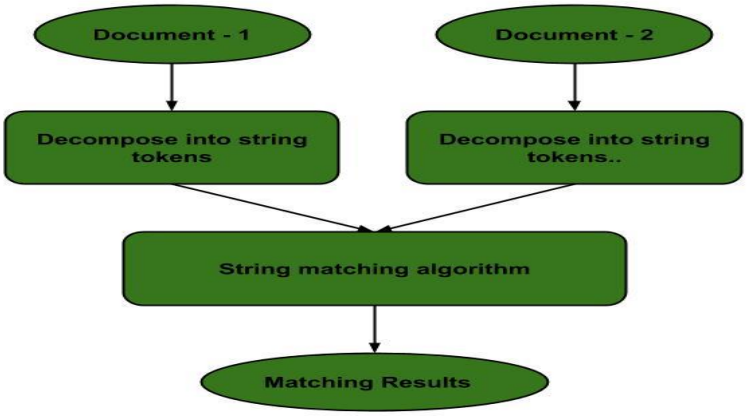
- Approximate String Matching Algorithms (also known as Fuzzy String Searching) searches for substrings of the input string.

- These techniques are used when the quality of the text is low, there are spelling errors in the pattern or text, finding DNA subsequences after mutation, heterogeneous databases, etc.
- Some approximate string matching algorithms are:

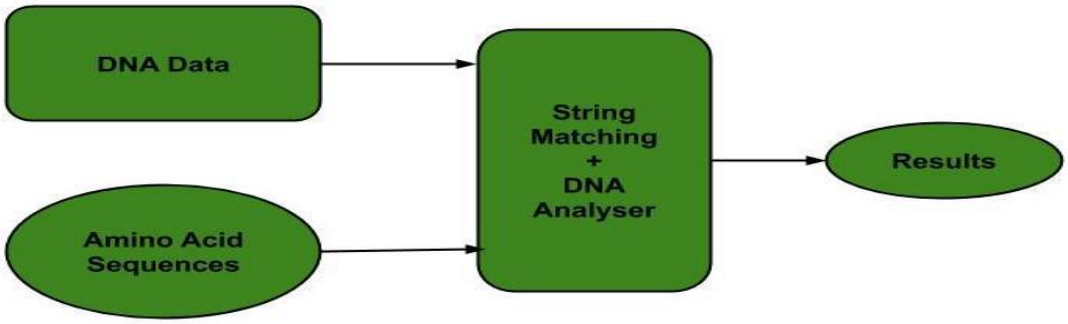
1. **Naive Approach:** It slides the pattern over text one by one and check for approximate matches. If they are found, then slides by 1 again to check for subsequent approximate matches.
2. Sellers Algorithm (Dynamic Programming)
3. Shift or Algorithm (Bitmap Algorithm)

Applications of String Matching Algorithms:

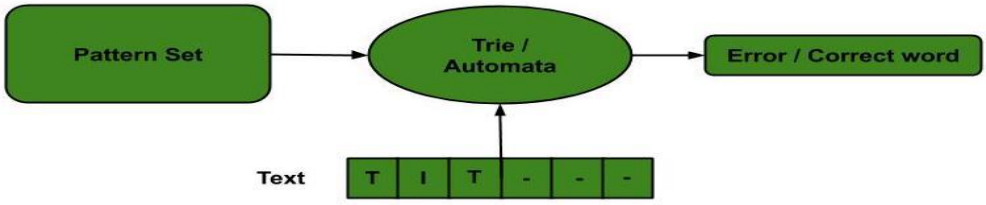
- **Plagiarism Detection:** The documents to be compared are decomposed into string tokens and compared using string matching algorithms. Thus, these algorithms are used to detect similarities between them and declare if the work is plagiarized or original.



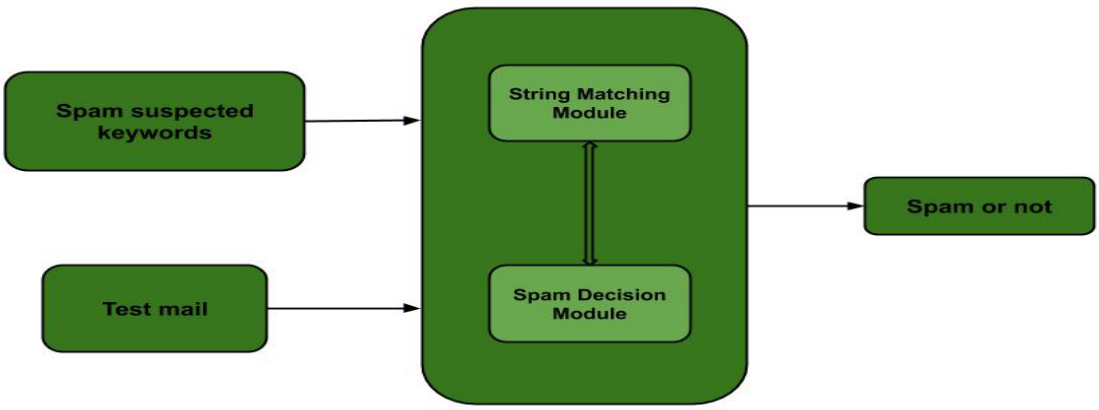
Bioinformatics and DNA Sequencing: Bioinformatics involves applying information technology and computer science to problems involving genetic sequences to find DNA patterns. String matching algorithms and DNA analysis are both collectively used for finding the occurrence of the pattern set.



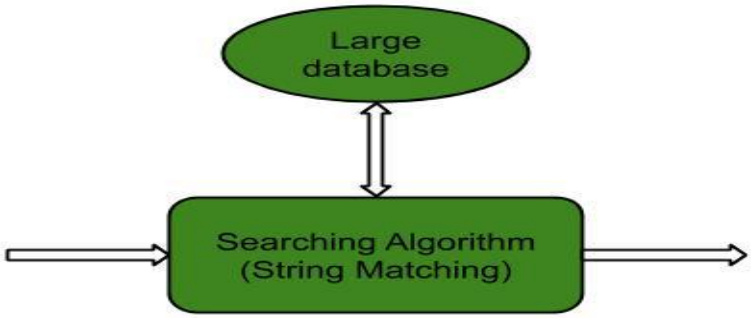
- **Digital Forensics:** String matching algorithms are used to locate specific text strings of interest in the digital forensic text, which are useful for the investigation.
- **Spelling Checker:** Trie is built based on a predefined set of patterns. Then, this trie is used for string matching. The text is taken as input, and if any such pattern occurs, it is shown by reaching the acceptance state.



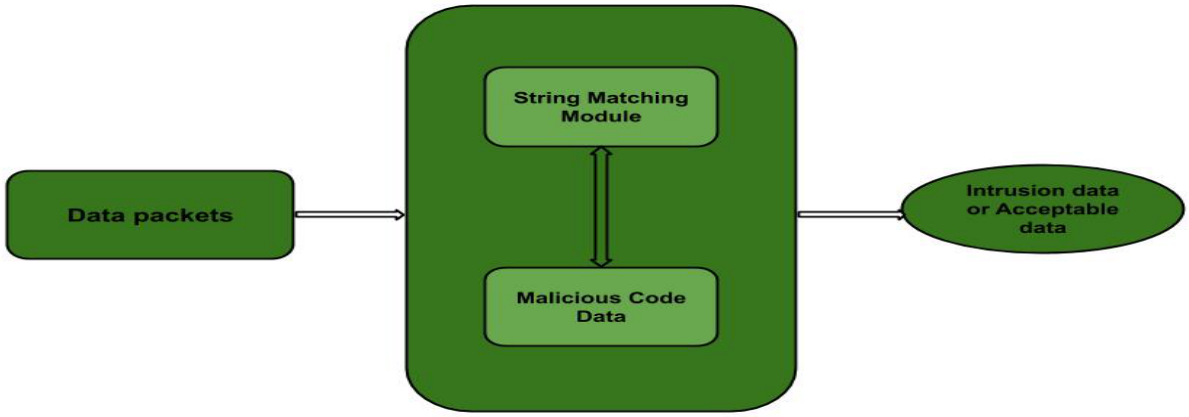
- **Spam filters:** Spam filters use string matching to discard the spam. For example, to categorize an email as spam or not, suspected spam keywords are searched in the content of the email by string matching algorithms. Hence, the content is classified as spam or not.



- **Search engines or content search in large databases:** To categorize and organize data efficiently, string matching algorithms are used. Categorization is done based on the search keywords. Thus, string matching algorithms make it easier for one to find the information they are searching for.



- **Intrusion Detection System:** The data packets containing intrusion-related keywords are found by applying string matching algorithms. All the malicious code is stored in the database, and every incoming data is compared with stored data. If a match is found, then the alarm is generated. It is based on exact stringmatching algorithms where each intruded packet must be detected.



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)

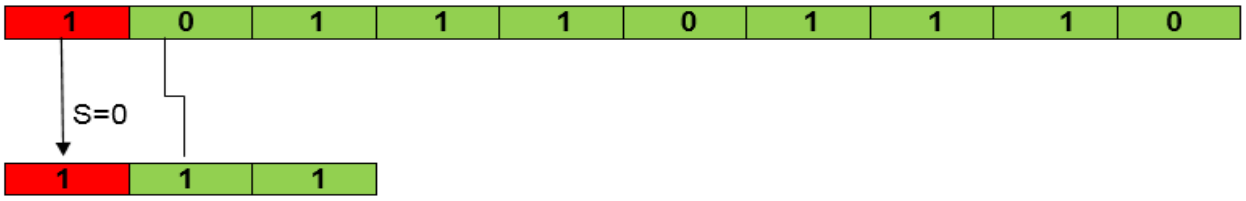
```
1. n ← length [T]
2. m ← length [P]
3. for s ← 0 to n -m
4. do if P [1.....m] = T [s + 1....s + m]
5. then print "Pattern occurs with shift" s
6. else print "Pattern not found"
```

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).

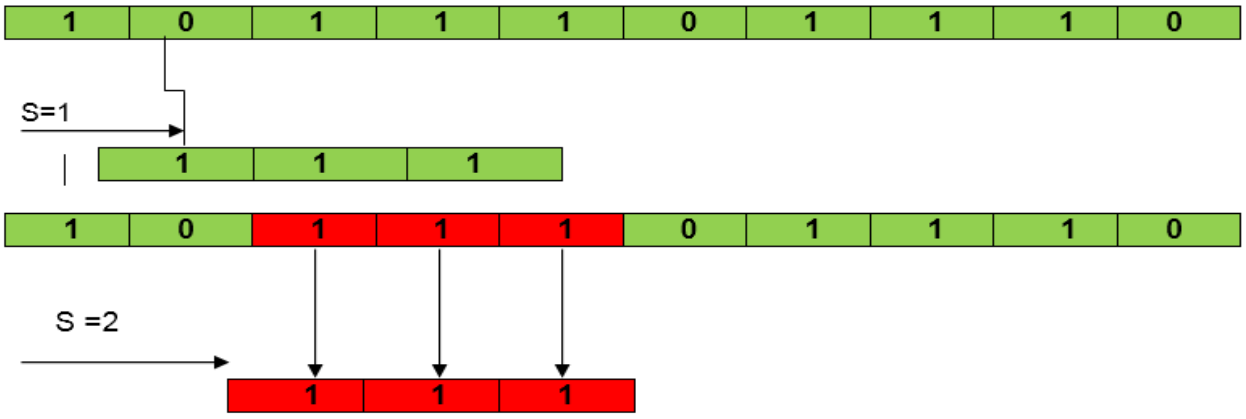
Example: Suppose T = 1011101110 P = 111 Find all the Valid Shifts

Solution:

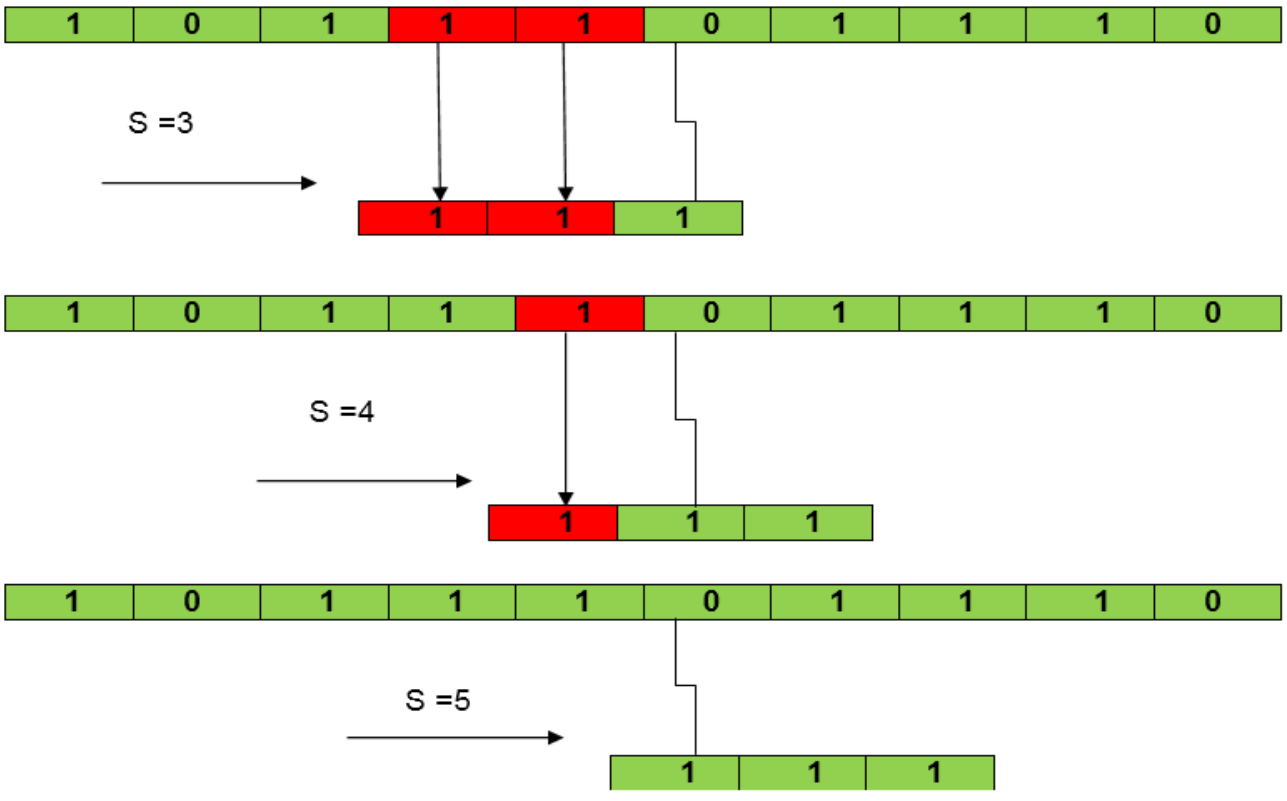
T = Text

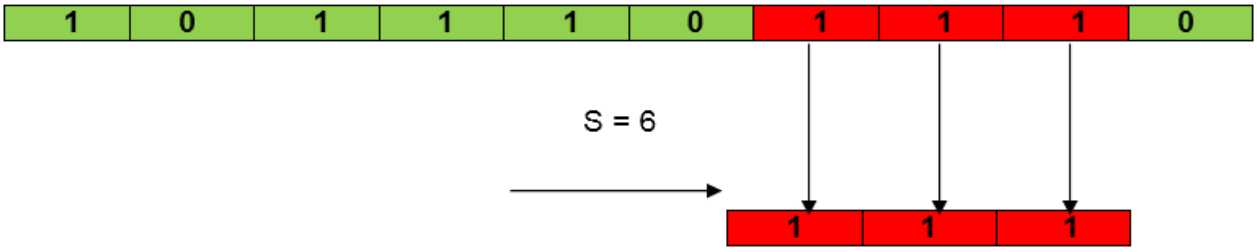


P = Pattern

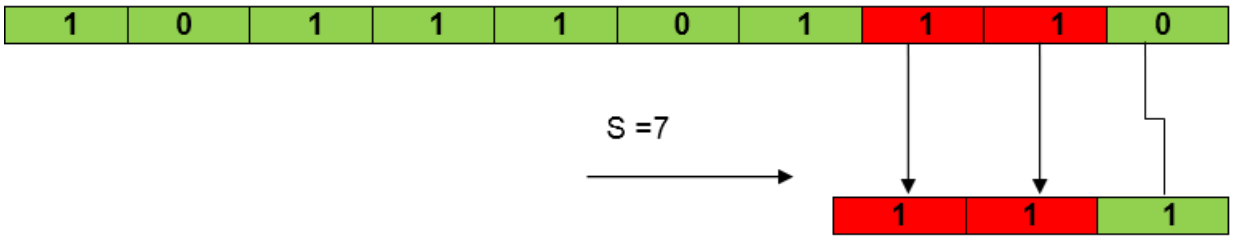


So, S=2 is a Valid Shift





So, S=6 is a Valid Shift



Bestcase::
The best case occurs when the first character of the pattern is not present in text at all.

txt[] = "AABCCAADDEE";

pat[] = "FAA";

The number of comparisons in best case is O(n).

Worstcase::
The worst case of Naive Pattern Searching occurs in following scenarios.

1) When all characters of the text and pattern are same.

txt[] = "AAAAAAAAAAAAAAAAAA";

pat[] = "AAAAA";

2) Worst case also occurs when only the last character is different.

txt[] = "AAAAAAAAAAAAAAAAAB"; pat[] = "AAAAB";

- The number of comparisons in the worst case is O(m*(n-m+1)
- The KMP matching algorithm improves the worst case to O(n).

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)

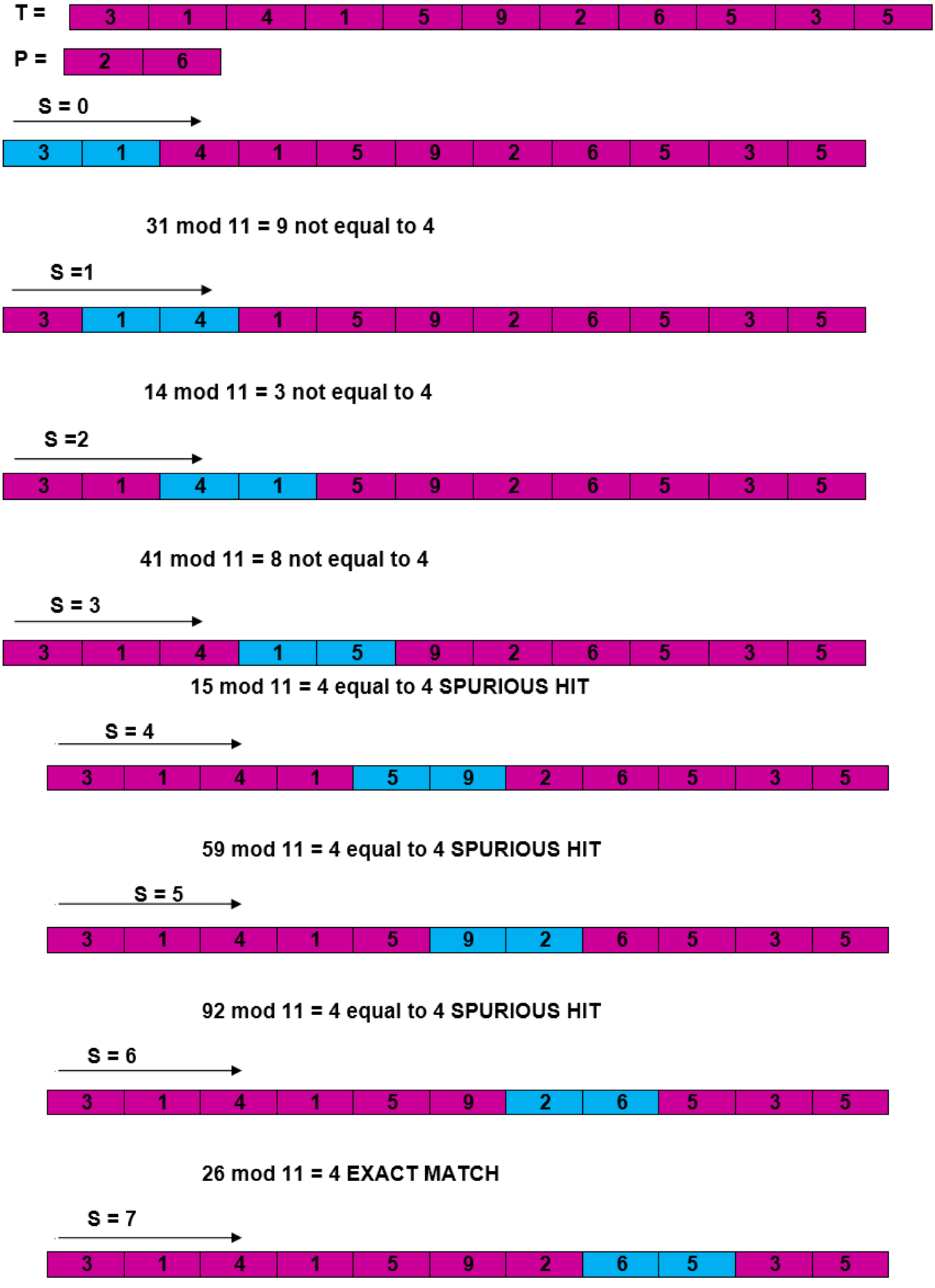
```
1. n ← length [T]
 2. m  ← length [P]
 3. h  ←  d^(m-1) mod q
 4. p ←  0
 5. t0 ←  0
 6. for i ← 1 to m
 7. do p ←  (dp + P[i]) mod q
 8. t0 ← (dt0+T [i]) mod q
 9. for s  ←  0 to n-m
 10. do if p = ts
 11. then if P [1.....m] = T [s+1.....s + m]
 12. then "Pattern occurs with shift" s
 13. If s < n-m
 14. then ts+1 ←  (d (ts-T [s+1]h)+T [s+m+1])mod q
```

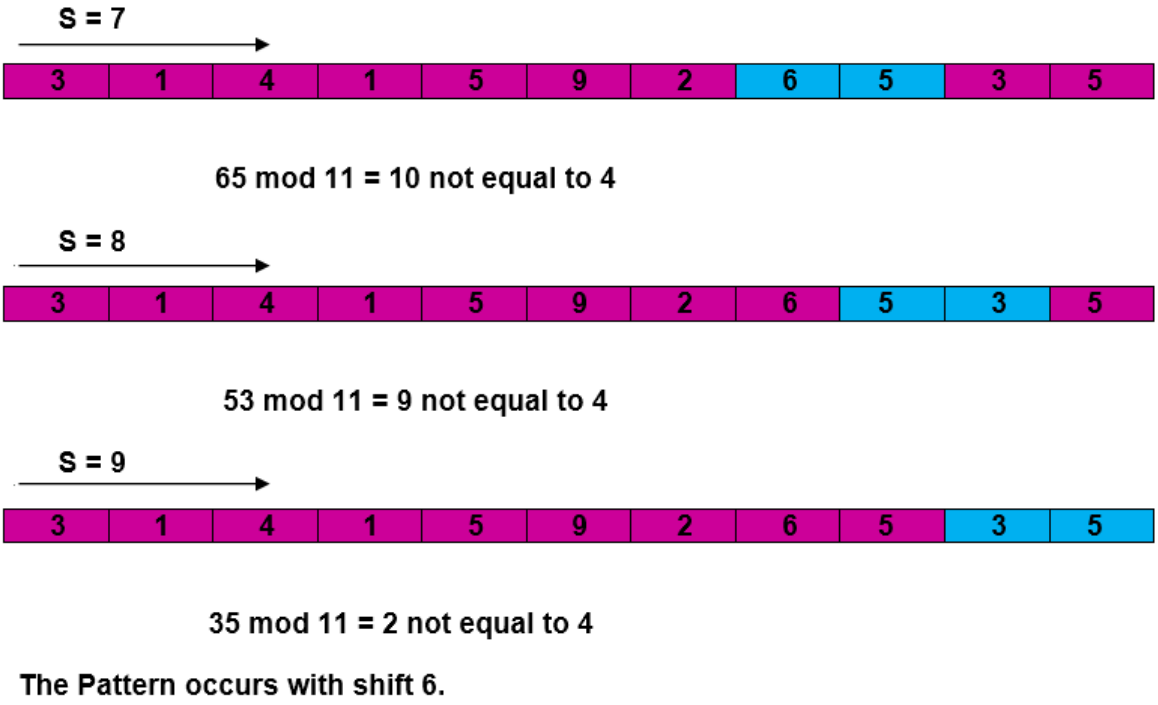
Example:

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:


T = 3 1 4 1 5 9 2 6 5 3 5
P = 2 6
S = 0
3 1 4 1 5 9 2 6 5 3 5
31 mod 11 = 9 not equal to 4
S =1
3 1 4 1 5 9 2 6 5 3 5
14 mod 11 = 3 not equal to 4
S =2
3 1 4 1 5 9 2 6 5 3 5
41 mod 11 = 8 not equal to 4
S = 3
3 1 4 1 5 9 2 6 5 3 5
15 mod 11 = 4 equal to 4 SPURIOUS HIT
S = 4
3 1 4 1 5 9 2 6 5 3 5
59 mod 11 = 4 equal to 4 SPURIOUS HIT
S = 5
3 1 4 1 5 9 2 6 5 3 5
92 mod 11 = 4 equal to 4 SPURIOUS HIT
S = 6
3 1 4 1 5 9 2 6 5 3 5
26 mod 11 = 4 EXACT MATCH
S = 7
3 1 4 1 5 9 2 6 5 3 5



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.

The Knuth-Morris-Pratt (KMP) Algorithm

- Knuth-Morris and Pratt introduce a linear time algorithm for the string matching problem.
- A matching time of O (n) is achieved by avoiding comparison with an element of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs.

Components of KMP Algorithm:

1. The Prefix Function (Π): The Prefix Function, Π for a pattern encapsulates knowledge about how the pattern matches against the shift of itself. This information can be used to avoid a useless shift of the pattern 'p.' In other words, this enables avoiding backtracking of the string 'S.'

2. The KMP Matcher: With string 'S,' pattern 'p' and prefix function 'Π' as inputs, find the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrences are found.

The Prefix Function (Π)

Following pseudo code compute the prefix function, Π:

COMPUTE- PREFIX- FUNCTION (P)

```
1. m ←length [P]          //'p' pattern to be matched
 2. Π [1] ← 0
 3. j ← 0
 4. for q ← 2 to m
 5. do while j > 0 and P [j + 1] ≠ P [q]
 6. do j ← Π [j]
 7. If P [j + 1] = P [q]
 8. then j← j + 1
 9. Π [q] ← j
 10. Return Π
```

Running Time Analysis:

- In the above pseudo code for calculating the prefix function, the for loop from step 4 to step 10 runs 'm' times. Step1 to Step3 take constant time. Hence the running time of computing prefix function is O (m).

Example: Compute Π for the pattern 'p' below:

P :

a	b	a	b	a	c	a

Solution: Initially: m = length [p] = 7

```
Π [1] = 0; k = 0
```

Step 1: q = 2, k = 0

Π [2] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0					

Step 2: q = 3, k = 0

Π [3] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1				

Step3: q =4, k =1

Π [4] = 2

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
π	0	0	1	2			

Step4: q = 5, k =2

Π [5] = 3

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3		

Step5: q = 6, k = 3

Π [6] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	

Step6: q = 7, k = 1

Π [7] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	1

After iteration 6 times, the prefix function computation is complete:

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

The KMP Matcher:

The KMP Matcher with the pattern 'p,' the string 'S' and prefix function 'Π' as input, finds a match of p in S. Following pseudo code compute the matching component of KMP algorithm:

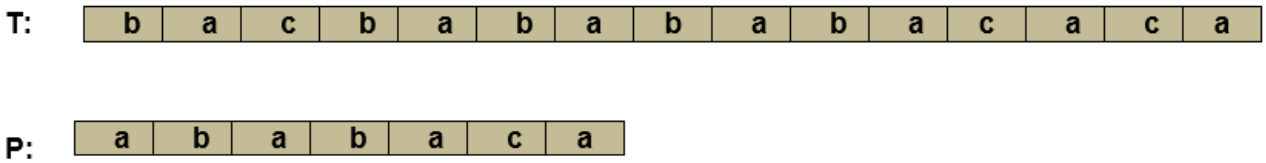
KMP-MATCHER (T, P)

```
1. n ← length [T]
 2. m ← length [P]
 3. Π← COMPUTE-PREFIX-FUNCTION (P)
 4. q ← 0          // numbers of characters matched
 5. for i ← 1 to n// scan S from left to right
 6. do while q > 0 and P [q + 1] ≠ T [i]
 7. do q ← Π [q]          // next character does not match
 8. If P [q + 1] = T [i]
 9. then q ← q + 1        // next character matches
 10. If q = m                              // is all of p matched?
 11. then print "Pattern occurs with shift" i - m
 12. q ← Π [q]                          // look for the next match
```

Running Time Analysis:

The for loop beginning in step 5 runs 'n' times, i.e., as long as the length of the string 'S.' Since step 1 to step 4 take constant times, the running time is dominated by this for the loop. Thus running time of the matching function is O (n).

Example: Given a string 'T' and pattern 'P' as follows:



Let us execute the KMP Algorithm to find whether 'P' occurs in 'T.'

For 'p' the prefix function, ? was computed previously and is as follows:

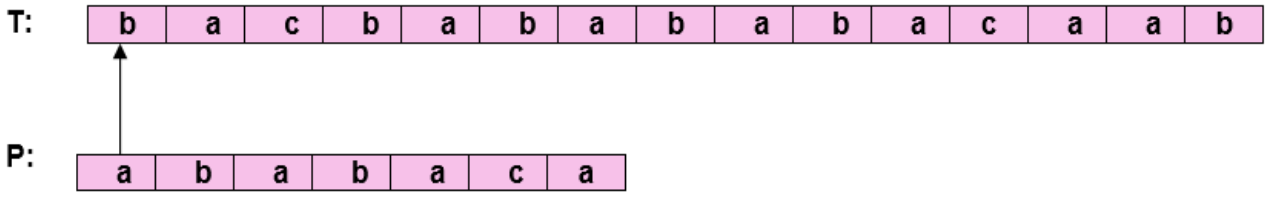
q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

Solution:

```
Initially: n = size of T = 15 ;
m = size of P = 7
```

Step1: i=1, q=0

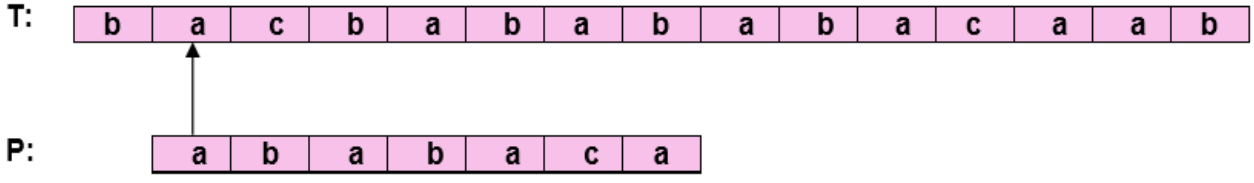
Comparing P [1] with T [1]



P [1] does not match with T [1]. 'p' will be shifted one position to the right.

Step2: i = 2, q = 0

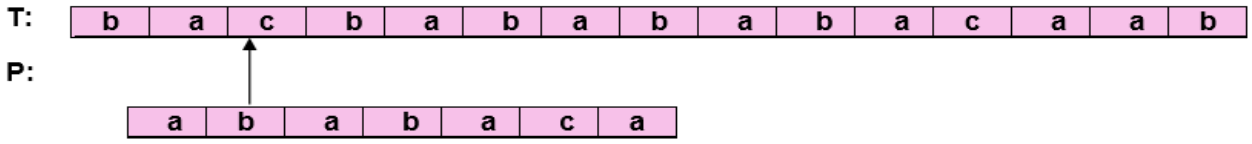
Comparing P [1] with T [2]



P [1] matches T [2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

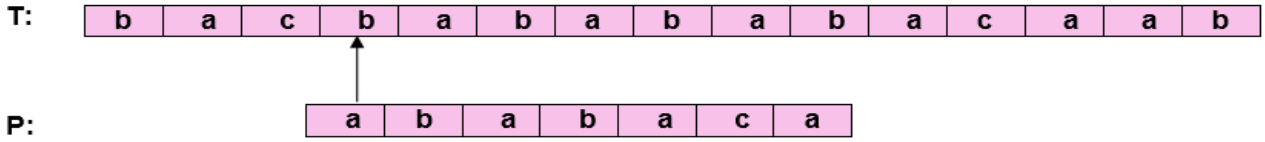
Comparing P [2] with T [3] P [2] doesn't match with T [3]



Backtracking on p, Comparing P [1] and T [3]

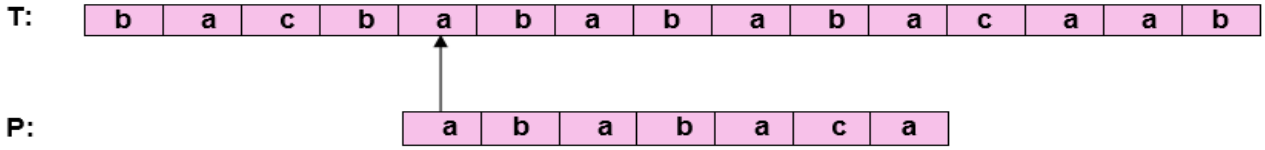
Step4: i = 4, q = 0

Comparing P [1] with T [4] P [1] doesn't match with T [4]



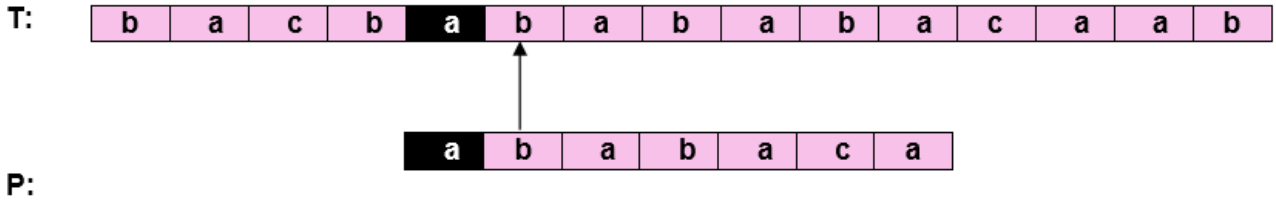
Step5: i = 5, q = 0

Comparing P [1] with T [5] P [1] match with T [5]



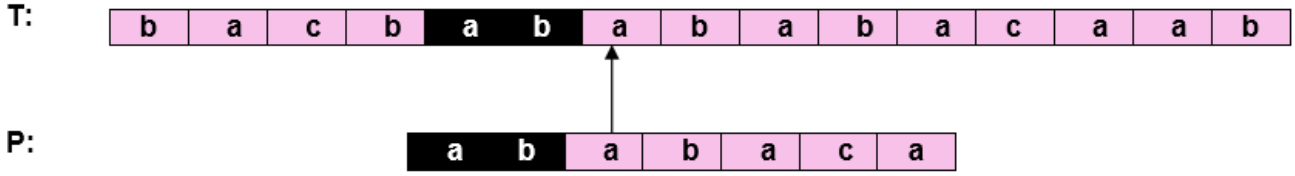
Step6: i = 6, q = 1

Comparing P [2] with T [6] P [2] matches with T [6]



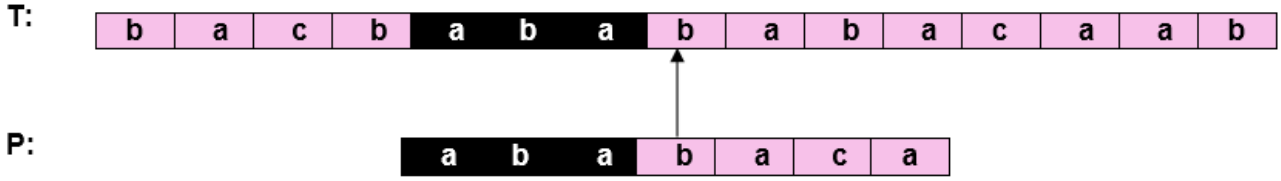
Step7: i = 7, q = 2

Comparing P [3] with T [7] P [3] matches with T [7]



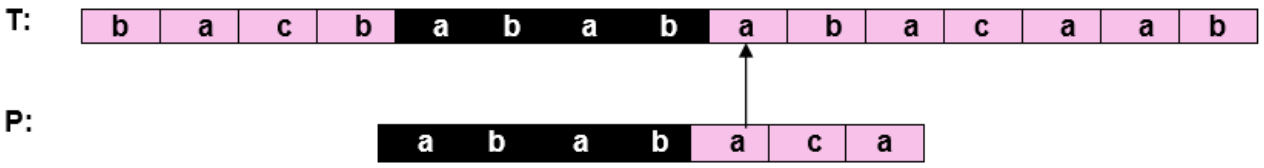
Step8: i = 8, q =3

Comparing P [4] with T [8] P [4] matches with T [8]



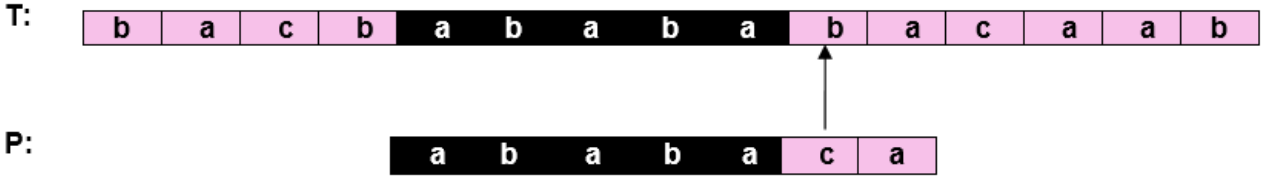
Step9: i = 9, q = 4

Comparing P [5] with T [9] P [5] matches with T [9]



Step10: i = 10, q = 5

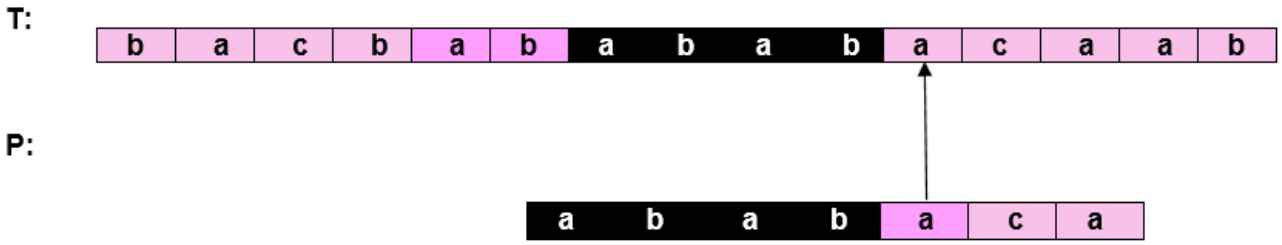
Comparing P [6] with T [10] P [6] doesn't match with T [10]



Backtracking on p, Comparing P [4] with T [10] because after mismatch q = π [5] = 3

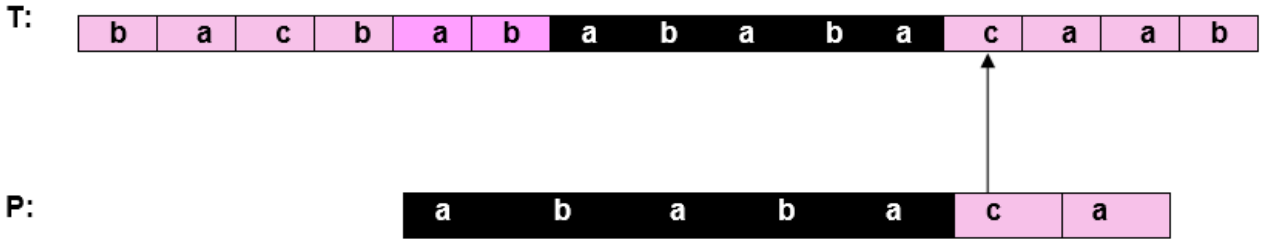
Step11: i = 11, q =4

Comparing P [5] with T [11] P [5] match with T [11]



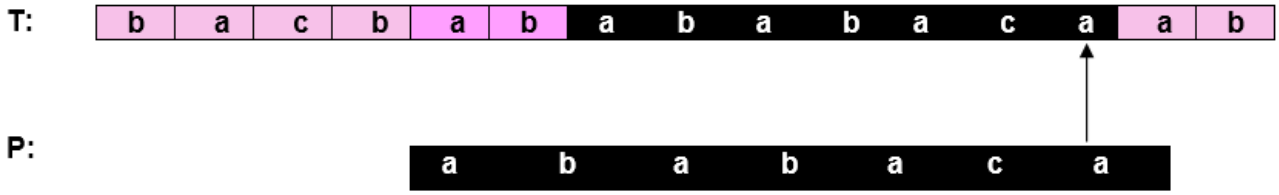
Step12: i = 12, q = 5

Comparing P [6] with T [12] P [6] matches with T [12]



Step13: i = 3, q = 6

Comparing P [7] with T [13] P [7] matches with T [13]



Pattern 'P' has been found to complexity occur in a string 'T.' The total number of shifts that took place for the match to be found is i-m = 13 - 7 = 6 shifts.

Polynomial Time Verification

Definition :Polynomial time verification :

If x is a problem and the solution given to the problem x is S the Algorithm A has a polynomial time to verify that the given solution S is correct or not for the problem x then this situation is called polynomial time verification.

The verification algorithm :

- The verification algorithm A is having two arguments
- One argument is an ordinary input string x
- another argument is a binary string y called as **certificate**
- a two argument algorithm A **verifies** an input string x if there exists a certificate y such that A(x,y)=1

- the language verified by a verification algorithm A is
  L={x€{0,1}*:there exists y€{0,1}* such that A(x,y)=1}

An algorithm A verifies a Language (problem)L if for any string x€L,there is a certificate y that A can use to prove that x€L.

EXAMPLE :Hamiltonian Cycle problem

HAM-CYCLE ={<G>:G is a Hamiltonian graph}

One possible decision algorithm list all permutations (paths) of the vertices of G and check each permutation to see if it is a Hamiltonian path.

the running time of this algorithm is

there are m! possible permutations

therefore the running time is Ω(m!)=Ω(srqrt(n!)=Ω(pow(2,sqrt(n))

this is exponential time there fore HAM-CYCLE is not solvable in polynomial time it is Np problem

verification of algorithm:

certificate y is : a path is given that is a Hamiltonian cycle

the verification algorithm A can easily verify that provided cycle is Hamiltonian by checking whether it is a permutation of the vertices of v and whether each of the consecutive edges along the cycle exists in the graph.

This verification algorithm can be implemented in O(n2) where n is the no of vertices in the graph.

Therefore the Hamiltonian cycle exists in the graph is verified in polynomial time .The Hamiltonian cycle problem can't be solved in polynomial time ,but it can be verifiable in polynomial time .

Approximation Algorithm:

- 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.

Travelling salesman problem?

Travelling Salesman Problem is based on a real life scenario, where a salesman from a company has to start from his own city and visit all the assigned cities exactly once and return to his home till the end of the day.

"Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point."
There are two important things to be cleared about in this problem statement,

- Visit every city exactly once
- Cover the shortest path

The approximate algorithms for TSP works only if the problem instance satisfies **Triangle-Inequality**.

Why are we using triangle inequality ?

"The least distant path to reach a vertex j from i is always to reach j directly from i, rather than through some other vertex k (or vertices)" i.e.,
`dis(i, j) <= dis(i, k) + dist(k, j)`
dis(a,b) = diatance between a & b, i.e. the edge weight.
The Triangle-Inequality holds in many practical situations.

Now our problem is approximated as we have tweaked the cost function/condition to triangle inequality.

The Algorithm:

- Let 0 be the starting and ending point for salesman.
- Construct Minimum Spanning Tree from with 0 as root using **Prim's Algorithm**.
- List vertices visited in preorder walk/Depth First Search of the constructed MST and add source node at the end.

Why 2 approximate ?

Following are some important points that maybe taken into account,

1. The cost of best possible Travelling Salesman tour is never less than the cost of MST. (The definition of MST says, it is a **minimum cost tree** that connects all vertices).

2. The total cost of full walk is at most twice the cost of MST (Every edge of MST is visited at-most twice)
3. The output of the above algorithm is less than the cost of full walk.

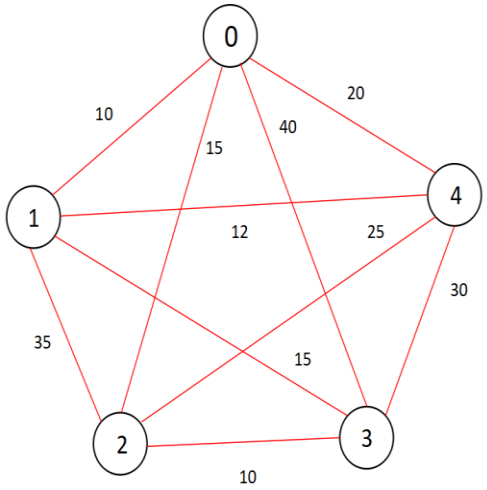
Step - 1 - Constructing The Minimum Spanning Tree

Step - 2 - Getting the preorder walk/ Defth first search walk

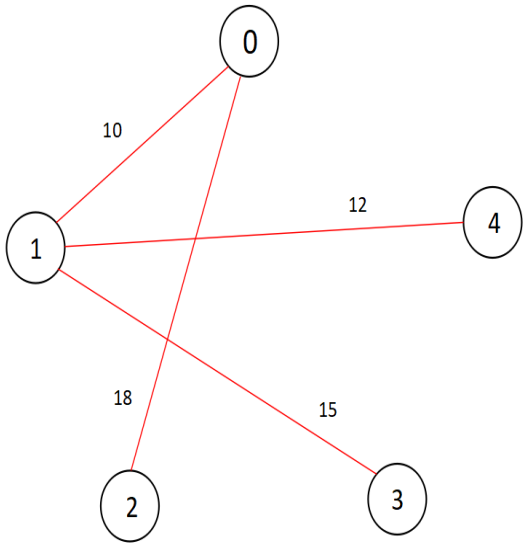
Step – 3 – Connect the vertices in the order.

Depth First Search Algorithm:

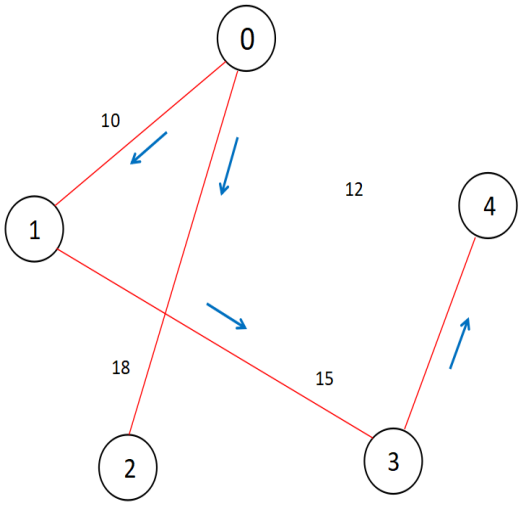
Let's have a look at the graph(adjacency matrix) given as input,



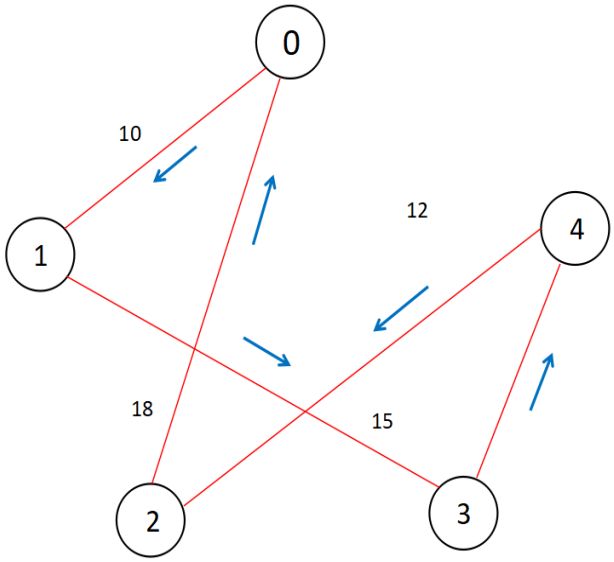
After performing **step-1**, we will get a Minimum spanning tree as below,



Performing DFS, we can get something like this,



Final step, connecting DFS nodes and the source node,



Hence we have the optimal path according to the approximation algorithm, i.e. **0-1-3-4-2-0**.

Complexity Analysis:

- The time complexity for obtaining MST from the given graph is **O(V^2)** where V is the number of nodes.
- The worst case space complexity for the same is **O(V^2)**, as we are constructing a `vector<vector<int>>` data structure to store the final MST.
- The time complexity for obtaining the DFS of the given graph is **O(V+E)** where V is the number of nodes and E is the number of edges.
  The space complexity for the same is **O(V)**.
- Hence the overall time complexity is **O(V^2)** and the worst case space somplexity of this algorithm is **O(V^2)**.