UNIT- V

Graphs–Definitions, Terminology, Applications and more definitions, Properties, Graph ADT, Graph Representations- Adjacency matrix, Adjacency lists, Graph Search methods - DFS and BFS, Spanning Trees and Minimum Cost Spanning Trees. String Matching- Introduction, String matching algorithms-Brute force, the Boyer–Moore algorithm, the KnuthMorris- Pratt algorithm.

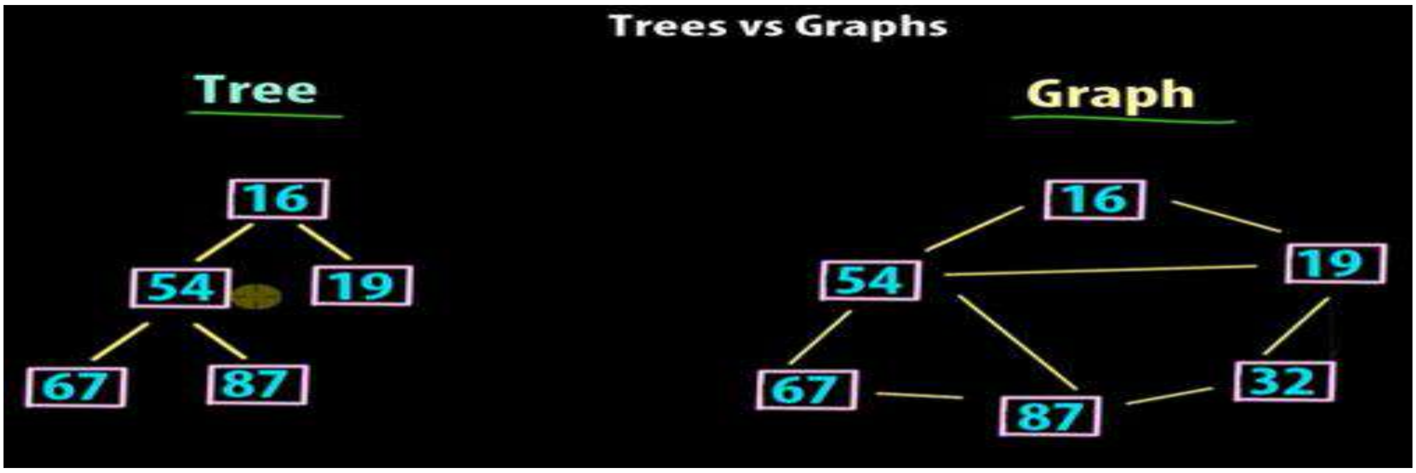
==

Graph Definition

Graph consists of a finite set of vertices and set of Edges (or Links) which connect to a pair of nodes.

A Graph is a non-linear data structure of nodes and edges.

	Trees	Graphs
1	Only one path | edge between two nodes.	Multiple paths | edges | links among two nodes.
2	Has a root node	No root node
3	Don't have loops	Can have loops
4	Have N-1 edges (N→ No of node)	No of edges not defined
5	Hierarchical Model	Network Model



A tree is an undirected graph.

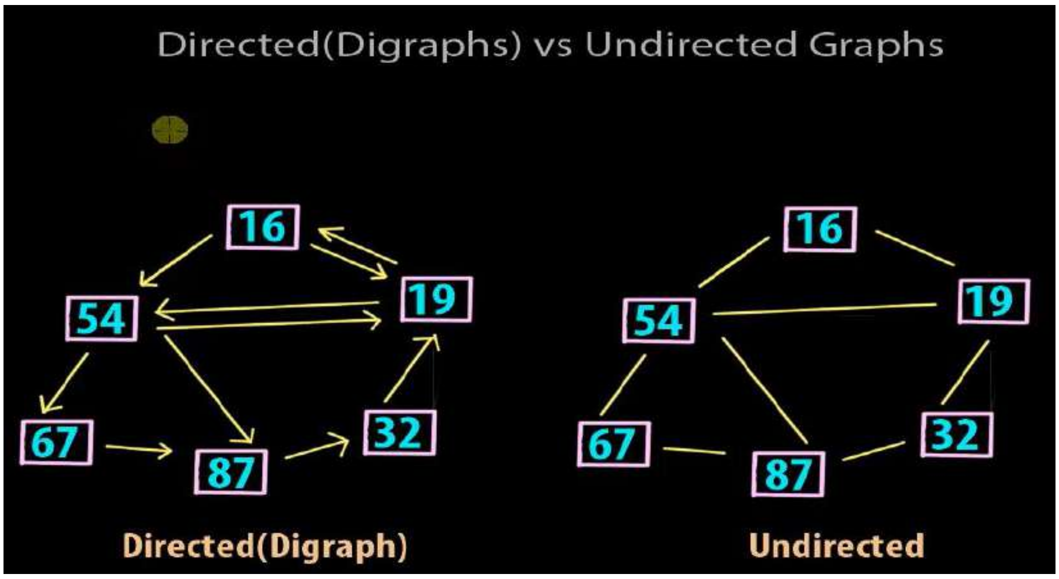
Directed Graph (Digraph)

A directed graph is a set of vertices (nodes) connected by edges, with each node having a direction associated with it.

Edges are usually represented by arrows pointing in the direction the graph can be traversed.

Undirected Graph

In an undirected graph the edges are bidirectional, with no direction associated with them. Hence, the graph can be traversed in either direction. The absence of an arrow tells us that the graph is undirected.

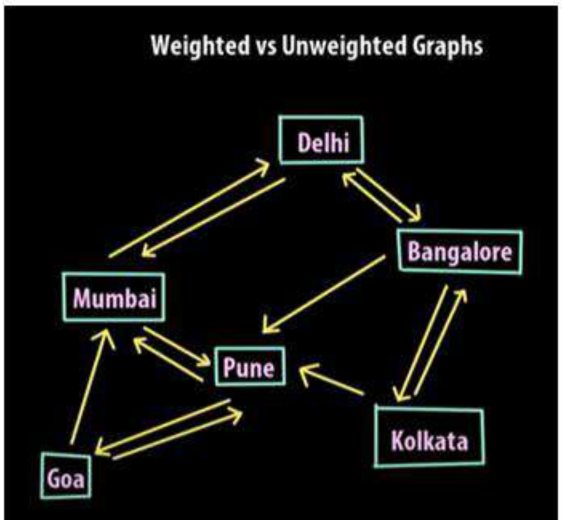


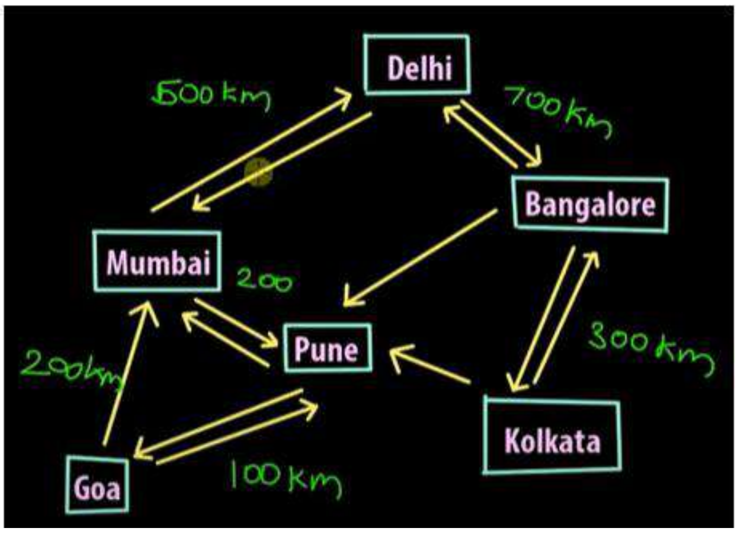
Unweighted Graph

- An unweighted graph is a graph in which all edges | paths are considered to have same weight.

Weighted Graph

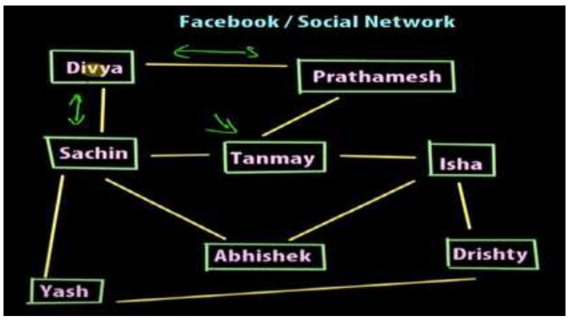
- A weighted graph is graph in which each branch is given a numerical weight.





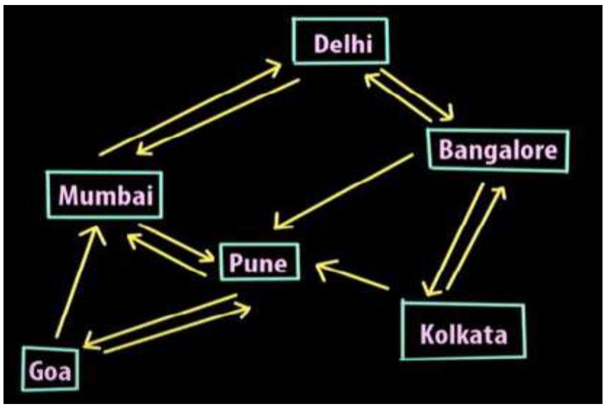
Examples of Graphs

- Social Networks

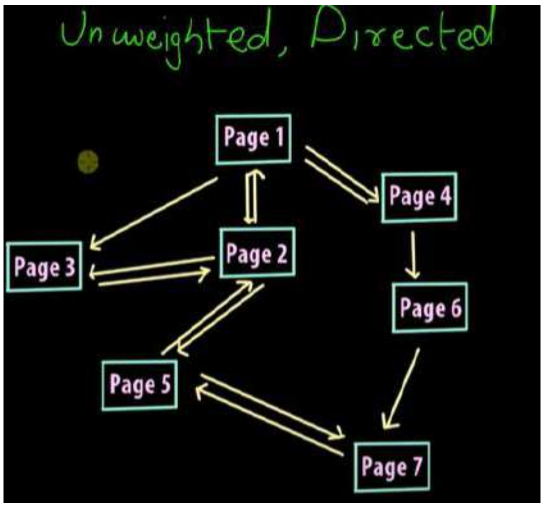


Give other names

- Maps (Google maps | GPS | Navigation | Flight



- World Wide Web



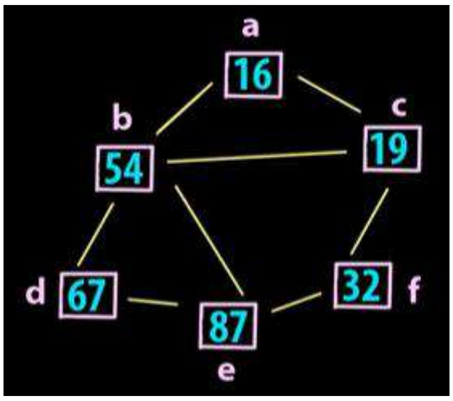
Adjacency Matrix

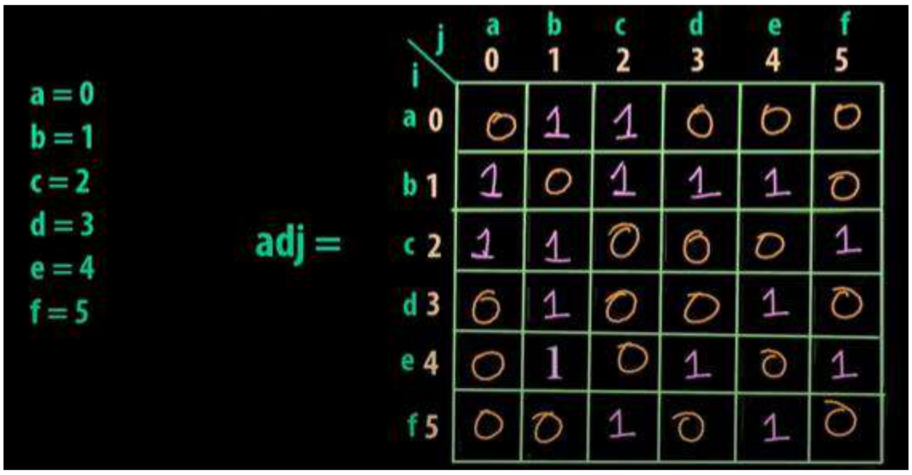
An adjacency matrix is a way of representing a graph as a matrix of Booleans (0s and 1s). A finite graph can be represented in the form of a square matrix on a computer where the Boolean value of the matrix indicates if there is a direct path between two vertices.

Let's assume the n X n matrix as adj[n][n].

If there is an edge from vertex i to j, mark adj[i][j] as 1. i.e. adj[i][j] == 1

If there is no edge from vertex i to j, mark adj[i][j] as 0. i.e. adj[i][j] == 0




a=0
b=1
c=2
d=3
e=4
f=5

adj =

i \ j	a 0	b 1	c 2	d 3	e 4	f 5
a 0	0	1	1	0	0	0
b 1	1	0	1	1	1	0
c 2	1	1	0	0	0	1
d 3	0	1	0	0	1	0
e 4	0	1	0	1	0	1
f 5	0	0	1	0	1	0

No need to create a separate list | hash table for searching in adjacency matrix. Since each element in a row represents a vertex, a linear search for the vertex with the required criterial can be used to find that vertex in O(n) linear time.

Adjacency Matrix Implementation of Graph is preferred only when the Graph is DENCE and the number of vertices is Relatively Less.

Time Complexity

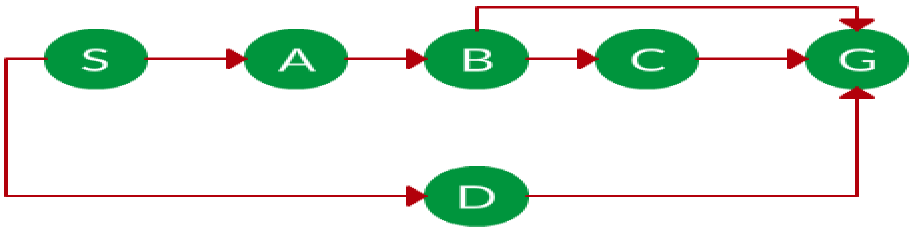
1. Find Node in Graph → O(1) | O(n)
2. Find all adjacent nodes of a node → O(n)
3. Check if 2 nodes are connected → O(1)

Space Complexity – O(n^2)

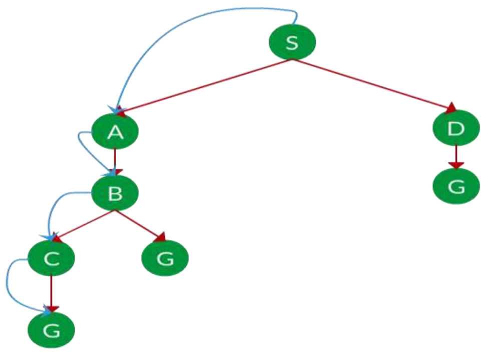
Depth-first search (DFS)

DFS is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. It uses last in- first-out strategy and hence it is implemented using a stack.

Question. *Which solution would DFS find to move from node S to node G if run on the graph below?*



Solution. The equivalent search tree for the above graph is as follows. As DFS traverses the tree "deepest node first", it would always pick the deeper branch until it reaches the solution (or it runs out of nodes, and goes to the next branch). The traversal is shown in blue arrows.



Path: *S -> A -> B -> C -> G*

= the depth of the search tree = the number of levels of the search tree.

= number of nodes in level .

Time complexity: *Equivalent to the number of nodes traversed in DFS*

Space complexity: *Equivalent to how large can the fringe get.*

Completeness: *DFS is complete if the search tree is finite, meaning for a given finite search tree, DFS will come up with a solution if it exists.*

Optimality: *DFS is not optimal, meaning the number of steps in reaching the solution, or the cost spent in reaching it is high.*

A C++ program to print DFS traversal for a given given graph

```
#include <bits/stdc++.h>

using namespace std;

class Graph

{


        // A function used by DFS

        void DFSUtil(int v);


public:

        map<int, bool> visited;

        map<int, list<int> > adj;

        // function to add an edge to graph

        void addEdge(int v, int w);


        // prints DFS traversal of the complete graph

        void DFS();
```

```
};

void Graph::addEdge(int v, int w)
{
        adj[v].push_back(w); // Add w to v's list.
}
void Graph::DFSUtil(int v)
{
        // Mark the current node as visited and print it
        visited[v] = true;
        cout << v << " ";

        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for (i = adj[v].begin(); i != adj[v].end(); ++i)
                if (!visited[*i])
                        DFSUtil(*i);
}

// The function to do DFS traversal. It uses recursive DFSUtil()
void Graph::DFS()
{
        // Call the recursive helper function to print DFS traversal starting from
        //all vertices one by one
        for (auto int i : adj)
                if (visited[i.first] == false)
                        DFSUtil(i.first);
}

int main()
{
        // Create a graph given in the above diagram
        Graph g;
        g.addEdge(0, 1);
        g.addEdge(0, 9);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(9, 3);

        cout << "Following is Depth First Traversal \n";
        g.DFS();

        return 0;
}
```

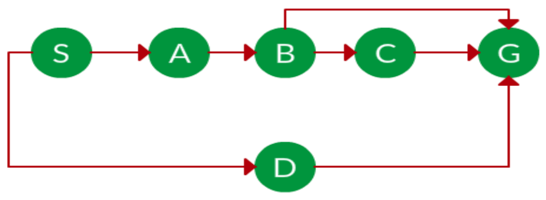
/*Output:

```
Following is Depth First Traversal
0 1 2 3 9
*/
```

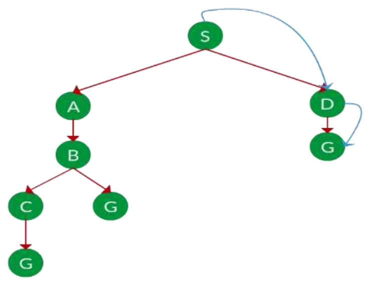
Breadth-first search

BFS is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'), and explores all of the neighbour nodes at the present depth prior to moving on to the nodes at the next depth level. It is implemented using a queue.

Example:
Question. Which solution would BFS find to move from node S to node G if run on the graph below?



Solution. The equivalent search tree for the above graph is as follows. As BFS traverses the tree "shallowest node first", it would always pick the shallower branch until it reaches the solution (or it runs out of nodes, and goes to the next branch). The traversal is shown in blue arrows.



Path: S -> D -> G

= the depth of the shallowest solution.

= number of nodes in level .

Time complexity: *Equivalent to the number of nodes traversed in BFS until the shallowest solution.*

Space complexity: *Equivalent to how large can the fringe get.*

Completeness: *BFS is complete, meaning for a given search tree, BFS will come up with a solution if it exists.*

Optimality: *BFS is optimal as long as the costs of all edges are equal.*

A C++ Program that makes traversal method for Breadth first search from a given source vertex.

```
//BFS(int s) traverses vertices reachable from s.
#include<iostream>
#include <list>

using namespace std;

// This class represents a directed graph using adjacency list representation
class Graph
{
        int V; // No. of vertices

        // Pointer to an array containing adjacency lists
        list<int> *adj;
public:
        Graph(int V); // Constructor

        // function to add an edge to graph
        void addEdge(int v, int w);

        // prints BFS traversal from a given source s
        void BFS(int s);
};

Graph::Graph(int V)
{
        this->V = V;
        adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
        adj[v].push_back(w); // Add w to v's list.
}
```

```
void Graph::BFS(int s)
{
        // Mark all the vertices as not visited
        bool *visited = new bool[V];
        for(int i = 0; i < V; i++)
                visited[i] = false;

        // Create a queue for BFS
        list<int> queue;

        // Mark the current node as visited and enqueue it
        visited[s] = true;
        queue.push_back(s);

        // 'i' will be used to get all adjacent vertices of a vertex
        list<int>::iterator i;

        while(!queue.empty())
        {
                // Dequeue a vertex from queue and print it
                s = queue.front();
                cout << s << " ";
                queue.pop_front();

                // Get all adjacent vertices of the dequeued vertex s.
                //If a adjacent has not been visited, then mark it visited and enqueue it
                for (i = adj[s].begin(); i != adj[s].end(); ++i)
                {
                        if (!visited[*i])
                        {
                                visited[*i] = true;
                                queue.push_back(*i);
                        }
                }
        }
}

int main()
{
        // Create a graph given in the above diagram

    Graph g(4);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);
        cout << "Following is Breadth First Traversal "
             << "(starting from vertex 2) \n";
        g.BFS(2);

        return 0;
}
/*Output
Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1
*/
```

String Matching Algorithms

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, 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

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.

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

1. The Boyer-Moore Algorithm
2. The Knuth-Morris-Pratt Algorithm

The Boyer-Moore Algorithm

Robert Boyer and J Strother Moore established it in 1977. The B-M String search algorithm is a particularly efficient algorithm and has served as a standard benchmark for string search algorithm ever since.

The B-M algorithm takes a 'backward' approach: the pattern string (P) is aligned with the start of the text string (T), and then compares the characters of a pattern from right to left, beginning with rightmost character.

If a character is compared that is not within the pattern, no match can be found by analysing any further aspects at this position so the pattern can be changed entirely past the mismatching character.

For deciding the possible shifts, B-M algorithm uses two pre-processing strategies simultaneously. Whenever a mismatch occurs, the algorithm calculates a variation using both approaches and selects the more significant shift thus, if make use of the most effective strategy for each case.

The two strategies are called heuristics of B - M as they are used to reduce the search. They are:

1. Bad Character Heuristics
2. Good Suffix Heuristics

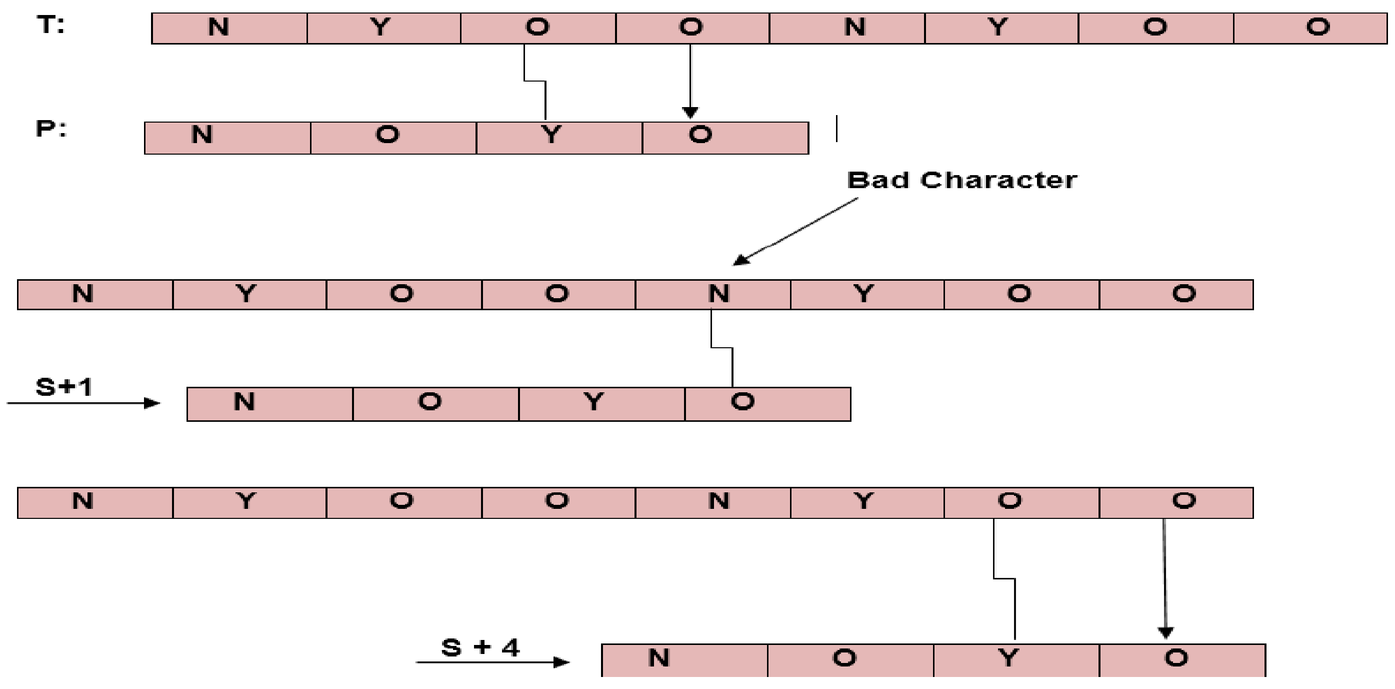
Bad Character Heuristics

This Heuristics has two implications:

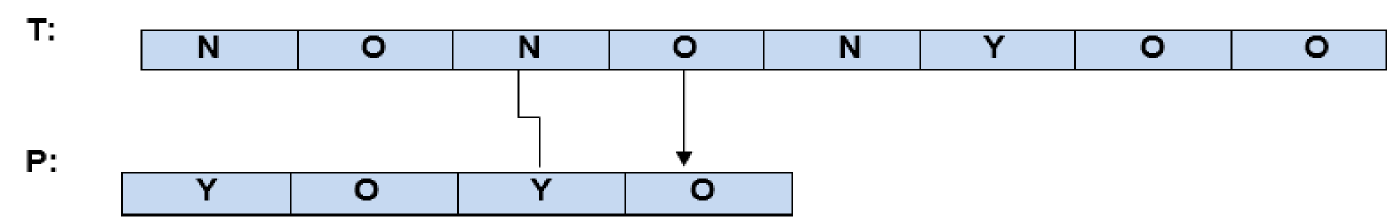
- Suppose there is a character in a text in which does not occur in a pattern at all. When a mismatch happens at this character (called as bad character), the whole pattern can be changed, begin matching form substring next to this 'bad character.'
-
- On the other hand, it might be that a bad character is present in the pattern, in this case, align the nature of the pattern with a bad character in the text.

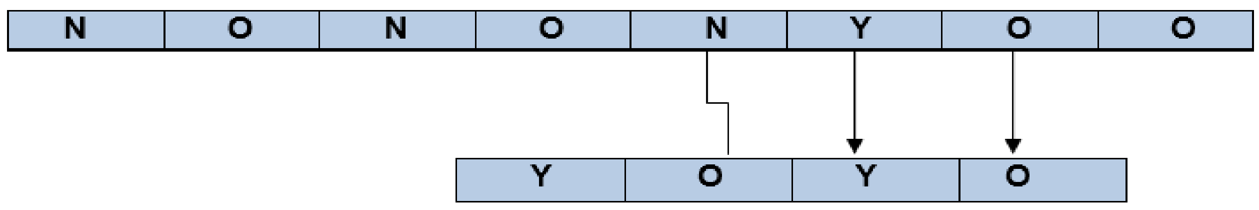
Thus, in any case shift may be higher than one.

Example1: Let Text T = <nyoo nyoo> and pattern P = <noyo>



Example2: If a bad character doesn't exist the pattern, then.

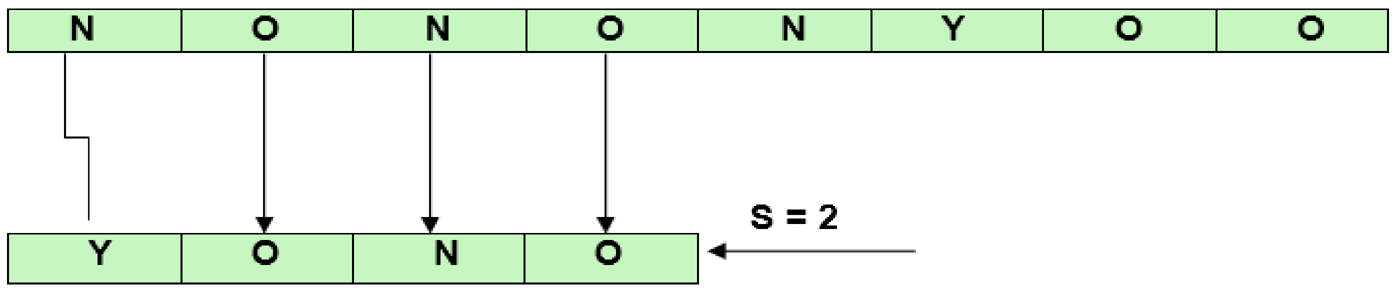




Problem in Bad-Character Heuristics

In some cases, Bad-Character Heuristics produces some negative shifts.

For Example



This means that we need some extra information to produce a shift on encountering a bad character. This information is about the last position of every aspect in the pattern and also the set of characters used in a pattern (often called the alphabet ∑of a pattern).

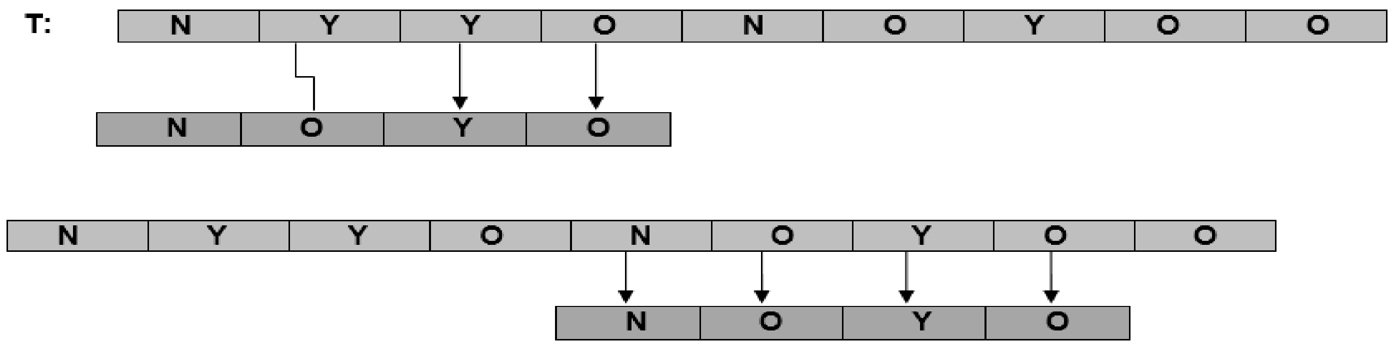
COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, ∑)

```
1. for each character a ∈ ∑
2. do λ [a] = 0
3. for j ← 1 to m
4. do λ [P [j]] ← j
5. Return λ
```

Good Suffix Heuristics

A good suffix is a suffix that has matched successfully. After a mismatch which has a negative shift in bad character heuristics, look if a substring of pattern matched till bad character has a good suffix in it, if it is so then we have an onward jump equal to the length of suffix found.

Example



COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)

```
1. Π ← COMPUTE-PREFIX-FUNCTION (P)
2. P'← reverse (P)
3. Π'← COMPUTE-PREFIX-FUNCTION (P')
4. for j ← 0 to m
5. do γ [j] ← m - Π [m]
6. for l ← 1 to m
7. do j ← m - Π' [L]
8. If γ [j] > l - Π' [L]
9. then γ [j] ← 1 - Π'[L]
10. Return γ
```

BOYER-MOORE-MATCHER (T, P, ∑)

```
1. n ←length [T]
2. m ←length [P]
3. λ← COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, ∑ )
4. γ← COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)
5. s ←0
6. While s ≤ n - m
7. do j ← m
8. While j > 0 and P [j] = T [s + j]
9. do j ←j-1
10. If j = 0
11. then print "Pattern occurs at shift" s
12. s ← s + γ[0]
13. else s ← s + max (γ [j], j - λ[T[s+j]])
```

The Knuth-Morris-Pratt (KMP)Algorithm

Knuth-Morris and Pratt introduces a linear time algorithm for the string-matching problem.

Components of KMP Algorithm:

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.'

The KMP Matcher: With string 'S,' pattern 'p' and prefix function 'Π' as inputs, finds 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 computes the prefix function, Π:

COMPUTE- PREFIX- FUNCTION (P)

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

T:															
	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

P:							
	a	b	a	b	a	c	a

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:

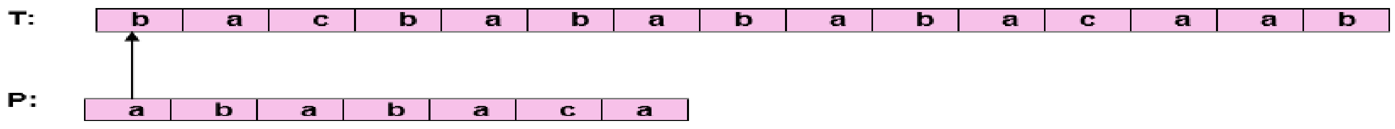
q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

Solution:

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

Step1: i=1, q=0

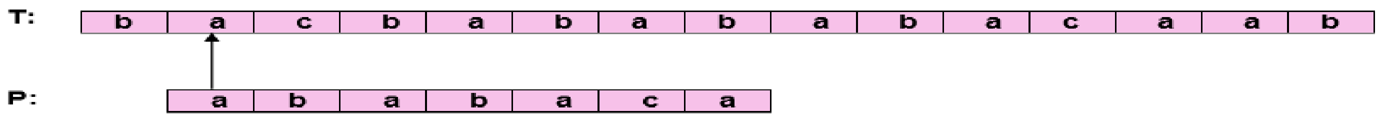
Comparing P [1] with T [1]



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

Step2: i = 2, q = 0

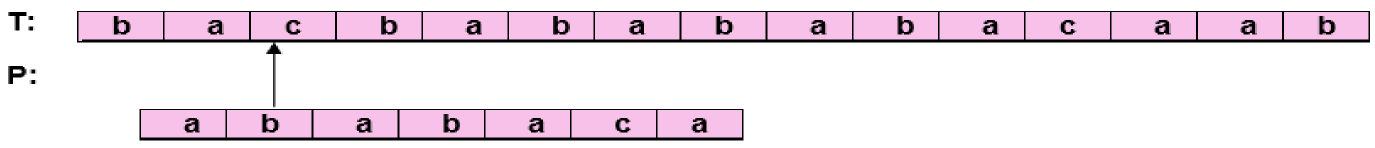
Comparing P [1] with T [2]



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

Step 3: i = 3, q = 1

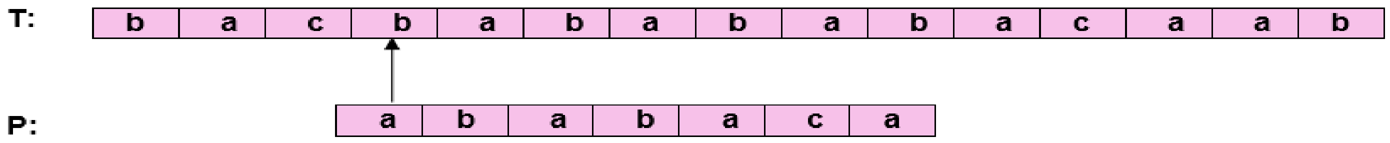
Comparing P [2] with T [3]   P [2] doesn't match with T [3]



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

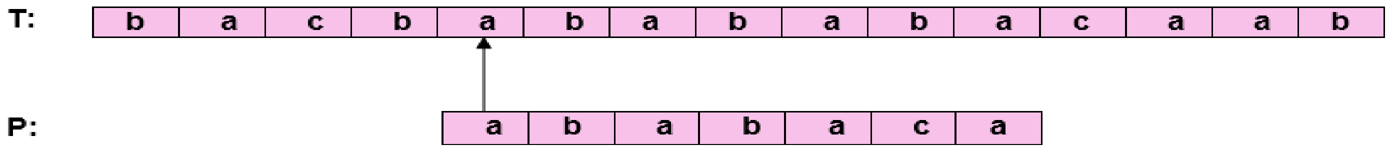
Step4: i = 4, q = 0

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



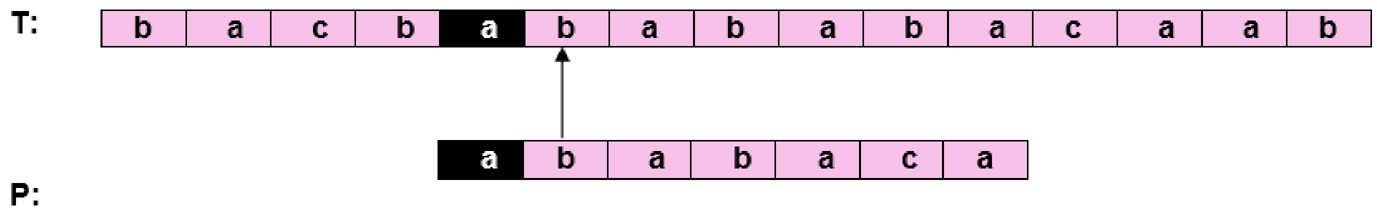
Step5: i = 5, q = 0

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



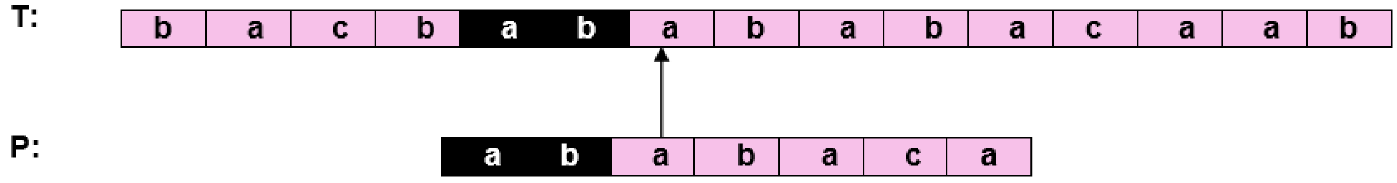
Step6: i = 6, q = 1

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



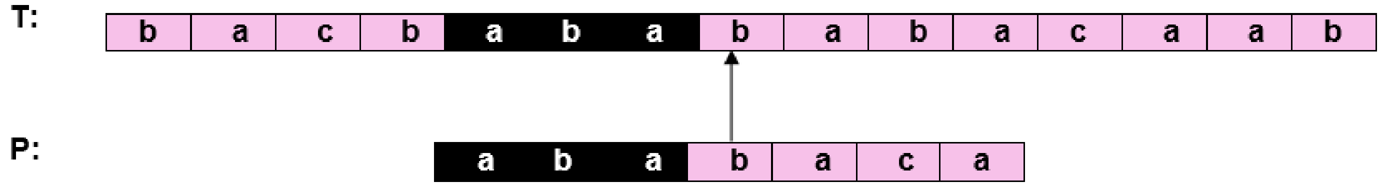
Step7: i = 7, q = 2

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



Step8: i = 8, q =3

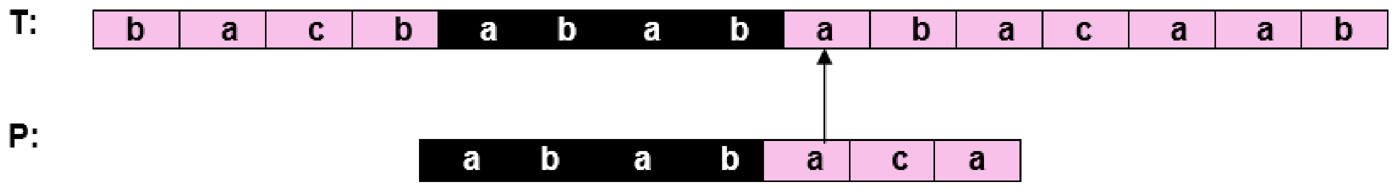
Comparing P [4] with T [8]   P [4] matches with T [8]



Step9: i = 9, q = 4

Comparing P [5] with T [9]

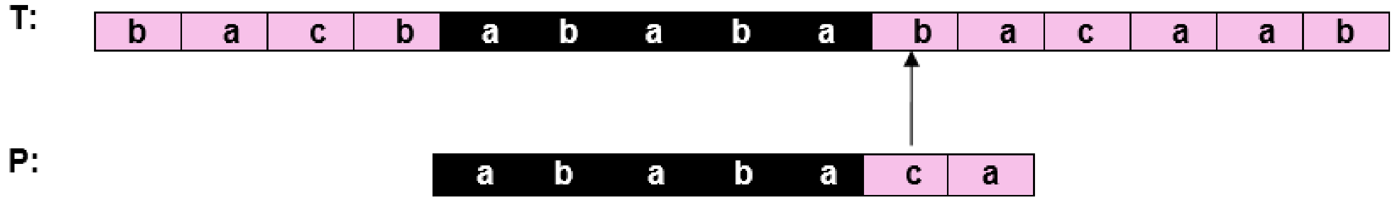
P [5] matches with T [9]



Step10: i = 10, q = 5

Comparing P [6] with T [10]

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

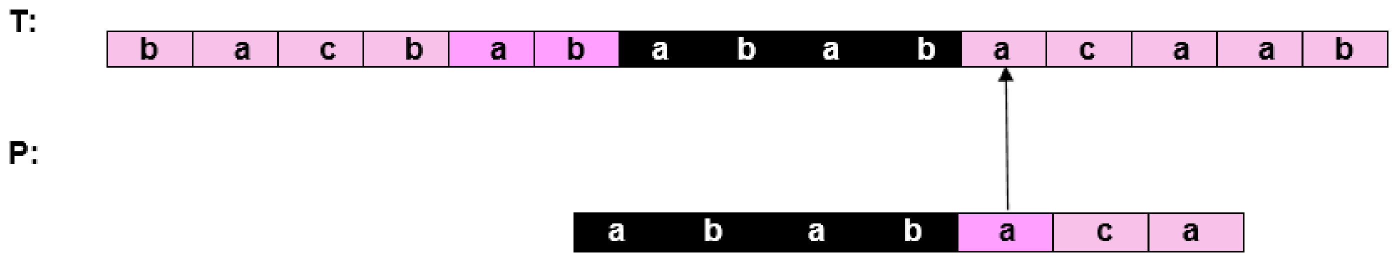


Backtracking on p, Comparing P [4] with T [10] because after mismatch q = π [5] = 3

Step11: i = 11, q =4

Comparing P [5] with T [11]

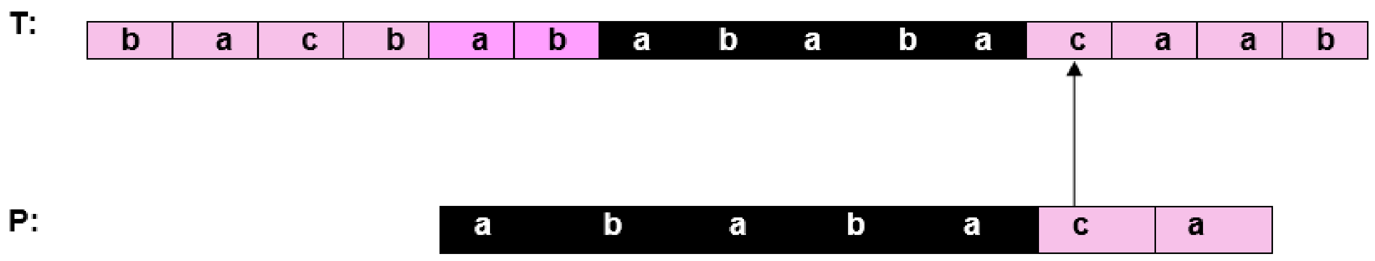
P [5] match with T [11]



Step12: i = 12, q = 5

Comparing P [6] with T [12]

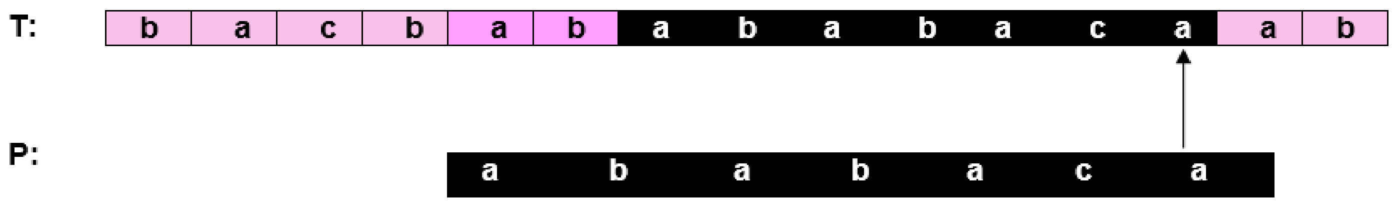
P [6] matches with T [12]



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.

Complexity Comparison of String-Matching Algorithm

Algorithm	Preprocessing Time	Matching Time
Knuth-Morris-Pratt	O(m)	O (n)
Boyer-Moore	O(|∑|)	(O ((n - m + 1) + |∑|))