FALL – SEMESTER **Name: Nidhi Singh**
Course Code: MCSE502L **Reg. No:22MAI0015**
Course-Title: – Design and Analysis of Algorithms
DIGITAL ASSIGNMENT - I

Slot-B1/TB1

Faculty : Dr. MOHAMMAD ARIF - SCOPE

Question –

1. Write the Brute force algorithm to string matching
2. Give the General Plan for Divide and Conquer Algorithms
3. Explain in detail about Knapsack Problem.
4. What is the convex hull problem? Explain the brute force approach to solve convex-hull with an example. Derive time complexity.
5. What does dynamic programming have in common with divide-and-Conquer
6. Define the Capacity Constraint in the context of the Maximum flow problem.
7. What is a residual network in the context of flow networks
8. Explain the KMP algorithm by taking one sample graph.
9. Explain the Rabin-Karp Algorithm by taking one sample graph.

1. Write the Brute force algorithm to string matching.

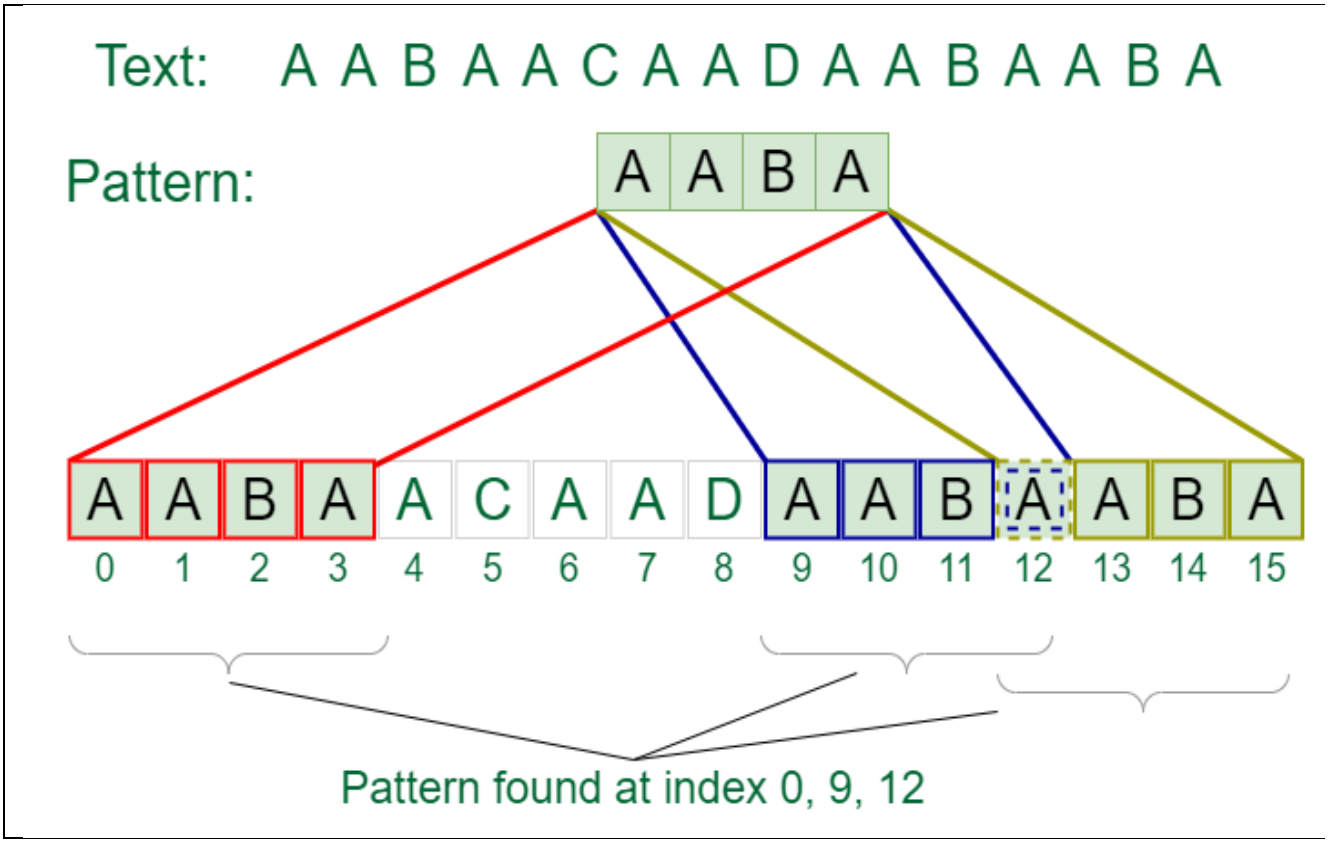
Given a text of length **N** txt[0..N-1] and a pattern of length M pat[0..M-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that N > M.

Examples:

Input: txt[] = "THIS IS A TEST TEXT", pat[] = "TEST"
Output: Pattern found at index 10
Input: txt[] = "AABAACAADAABAABA", pat[] = "AABA"
Output: Pattern found at index 0, Pattern found at index 9, Pattern found at index 12

Complexity :-

Time Complexity: $O(N^2)$
Auxiliary Space: O(1)



What is the best case of Naive algorithm for Pattern Searching?
The best case occurs when the first character of the pattern is not present in the text at all.

txt[] = "AABCCAADDEE";
pat[] = "FAA";
The number of comparisons in the best case is O(N).

What is the worst caseof Naive algorithm for Pattern Searching?

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

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

```
txt[] = "AAAAAAAAAAAAAAAAAA";
pat[] = "AAAAA";
```

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

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

Brute-Force String Matching

Searching for a pattern, P[0...m-1], in text, T[0...n-1]

Algorithm *BruteForceStringMatch*(*T*[0...*n*-1], *P*[0...*m*-1])

```
for i ← 0 to n-m do
        j ← 0
        while j < m and P[j] = T[i+j] do
                j++
        if j = m then return i
return -1
```

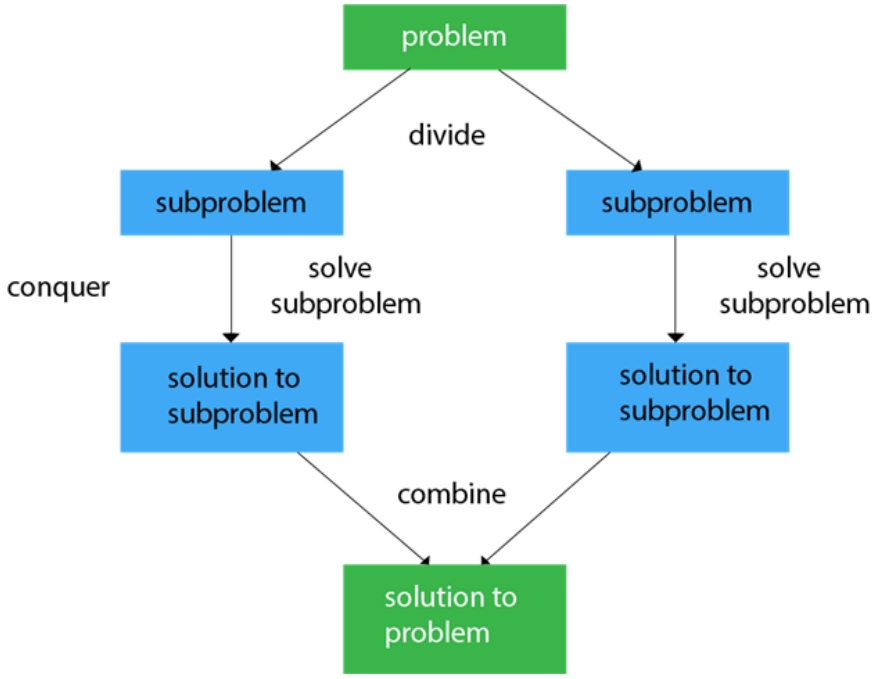
Worst case when a shift is not made until the *m*-th comparison, so $\Theta(nm)$
Typically shift is made early then Average case $\Theta(n)$ or for $m << n$

2. Give the General Plan for Divide and Conquer Algorithms.

Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

Divide and Conquer algorithm consists of a dispute using the following three steps.

1. **Divide** the original problem into a set of subproblems.
2. **Conquer:** Solve every subproblem individually, recursively.
3. **Combine:** Put together the solutions of the subproblems to get the solution to the whole problem.



Generally, we can follow the **divide-and-conquer** approach in a three-step process.

Examples: The specific computer algorithms are based on the Divide & Conquer approach:

1. Maximum and Minimum Problem
2. Binary Search
3. Sorting (merge sort, quick sort)
4. Tower of Hanoi.

Fundamental of Divide & Conquer Strategy:
There are two fundamental of Divide & Conquer Strategy:

1. Relational Formula
2. Stopping Condition

1. Relational Formula: It is the formula that we generate from the given technique. After generation of Formula we apply D&C Strategy, i.e. we break the problem recursively & solve the broken subproblems.

2. Stopping Condition: When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So the condition where the need to stop our recursion steps of D&C is called as Stopping Condition.

3. Explain in detail about Knapsack Problem.

The Knapsack Problem is an Optimization Problem in which we have to find an optimal answer among all the possible combinations. In this problem, we are given a set of items having different weights and values. We have to find the optimal solution considering all the given items.
There are three types of knapsack problems :
0-1 Knapsack, Fractional Knapsack and Unbounded Knapsack.

What is the 0/1 knapsack problem?

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by the dynamic programming.

Example of 0/1 knapsack problem.

Consider the problem having weights and profits are:
Weights: {3, 4, 6, 5}
Profits: {2, 3, 1, 4}
The weight of the knapsack is 8 kg
The number of items is 4
The above problem can be solved by using the following method:
x_i = {1, 0, 0, 1}
= {0, 0, 0, 1}
= {0, 1, 0, 1}

The above are the possible combinations. 1 denotes that the item is completely picked and 0 means that no item is picked. Since there are 4 items so possible combinations will be:

2^4 = 16; So. There are 16 possible combinations that can be made by using the above problem. Once all the combinations are made, we have to select the combination that provides the maximum profit.

What is the fractional knapsack problem?

The fractional knapsack problem means that we can divide the item. For example, we have an item of 3 kg then we can pick the item of 2 kg and leave the item of 1 kg. The fractional knapsack problem is solved by the Greedy approach.

4. What is the convex hull problem? Explain the brute force approach to solve convex-hull with an example. Derive time complexity.

Answer:-

The Convex Hull Problem :-

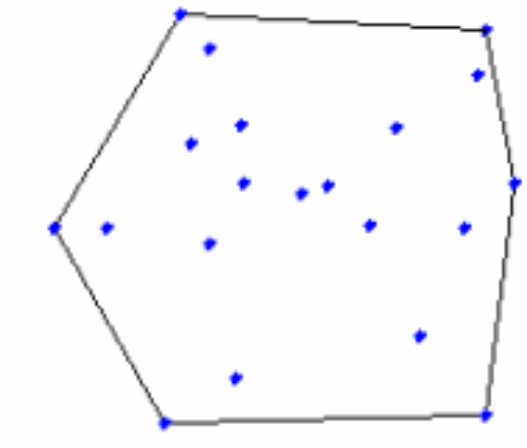
A polygon is convex if any line segment joining two points on the boundary stays within the polygon. Equivalently, if you walk around the boundary of the polygon in counter clockwise direction you always take left turns. The convex hull of a set of points in the plane is the smallest convex polygon for which each point is either on the boundary or in the interior of the polygon. One might think of the points as being nails sticking out of a wooden board: then the convex hull is the shape formed by a tight rubber band that surrounds all the nails. A vertex is a corner of a polygon. For example, the highest, lowest, leftmost and rightmost points are all vertices of the convex hull. Some other characterizations are given in the exercises.

In this problem, we want to compute the convex hull of a set of points? What does this mean?

• **Formally:** It is the smallest convex set containing the points. A convex set is one in which if we connect any two points in the set, the line segment connecting these points must also be in the set.

• **Informally:** It is a rubber band wrapped around the "outside" points.

It is an applet so you can play with it to see what a convex hull is if you like.



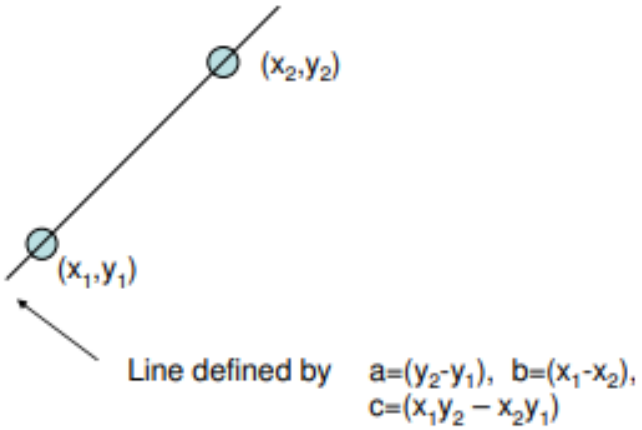
Theorem: The convex hull of any set S of n>2 points (not all collinear) is a convex polygon with the vertices at some of the points of S.

Brute-force algorithm to find the convex hull:-

In addition to the theorem, also note that a line segment connecting two points P1 and P2 is a part of the convex hull's boundary if and only if all the other points in the set lie on the same side of the line drawn through these points.

For all points above the line, ax + by > c, while for all points below the line, ax + by < c. Using these formulas, we can determine if two points are on the boundary to the convex hull.

With a little geometry:



Algorithm :-

High level pseudocode for the algorithm then becomes:

```
for each point Pi
    for each point Pj where Pj ≠ Pi
      Compute the line segment for Pi and Pj
          for every other point Pk where Pk ≠ Pi and Pk ≠ Pj
```

If each Pk is on one side of the line segment, label Pi and Pj in the convex hull

worst case cost of the algorithm : $O(n^3)$

5. What does dynamic programming have in common with divide-and-Conquer.

Answer :-

I would not treat them as something completely different. Because **they both work by recursively breaking down a problem into two or more sub-problems** of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

So why do we still have different paradigm names then and why I called dynamic programming an extension. It is because dynamic programming approach may be applied to the problem **only if the problem has certain restrictions or prerequisites**. And after that dynamic programming extends divide and conquer approach with **memoization** or **tabulation** technique.

Let's go step by step…

Dynamic Programming Prerequisites/Restrictions

As we've just discovered there are two key attributes that divide and conquer problem must have in order for dynamic programming to be applicable:

- **Optimal substructure**—optimal solution can be constructed from optimal solutions of its sub problems
- **Overlapping sub-problems**—problem can be broken down into sub problems which are reused several times or a recursive algorithm for the problem solves the same sub problem over and over rather than always generating new sub problems.

Once these two conditions are met we can say that this divide and conquer problem may be solved using dynamic programming approach.

Dynamic Programming Extension for Divide and Conquer

Dynamic programming approach extends divide and conquer approach with two techniques (**memoization** and **tabulation**) that both have a purpose of storing and re-using sub-problems solutions that may drastically improve performance. For example naive recursive implementation of Fibonacci function has time complexity of O(2^n) where DP solution doing the same with only O(n) time.

Memoization (top-down cache filling) refers to the technique of caching and reusing previously computed results. The memoized fib function would thus look like this:

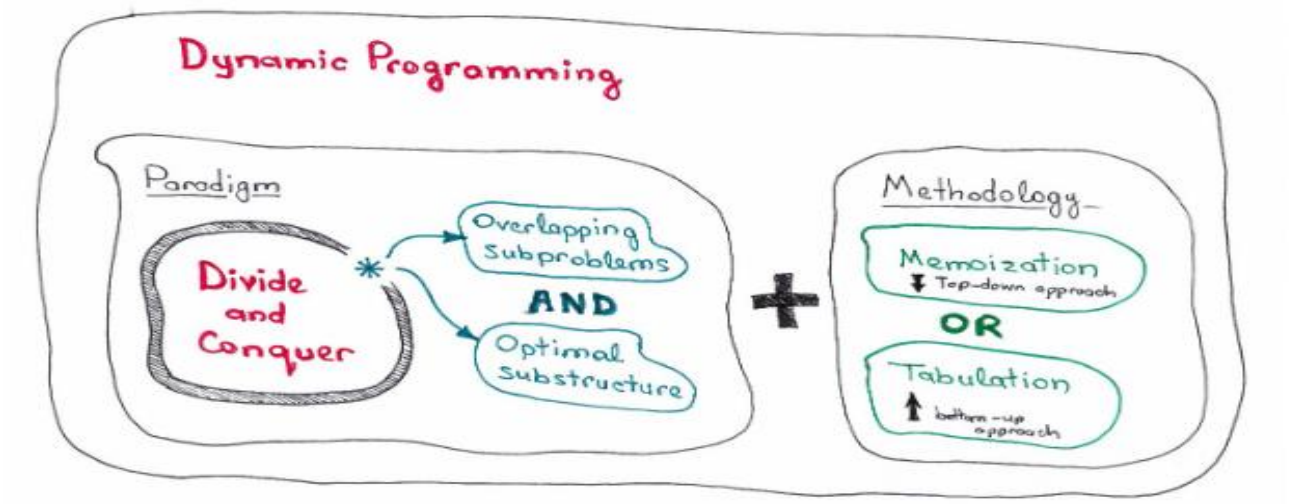
```
memFib(n) {
   if (mem[n] is undefined)
      if (n < 2) result = n
      else result = memFib(n-2) + memFib(n-1)
      mem[n] = result
   return mem[n]
}
```

Tabulation (bottom-up cache filling) is similar but focuses on filling the entries of the cache. Computing the values in the cache is easiest done iteratively. The tabulation version of fib would look like this:

```
tabFib(n) {
   mem[0] = 0
   mem[1] = 1
   for i = 2...n
      mem[i] = mem[i-2] + mem[i-1]
   return mem[n]
}
```

So What the Difference Between DP and DC After All

Since we're now familiar with DP prerequisites and its methodologies we're ready to put all that was mentioned above into one picture.



6. Define the Capacity Constraint in the context of the Maximum flow problem.

Answer :-

Definition: A Flow Network is a directed graph G = (V, E) such that

1. For each edge (u, v) ∈ E, we associate a nonnegative weight capacity c (u, v) ≥ 0.
2. If (u, v) ∉ E, we assume that c (u, v) = 0.
3. There are two distinguishing points, the source s, and the sink t;
4. For every vertex v ∈ V, there is a path from s to t containing v.

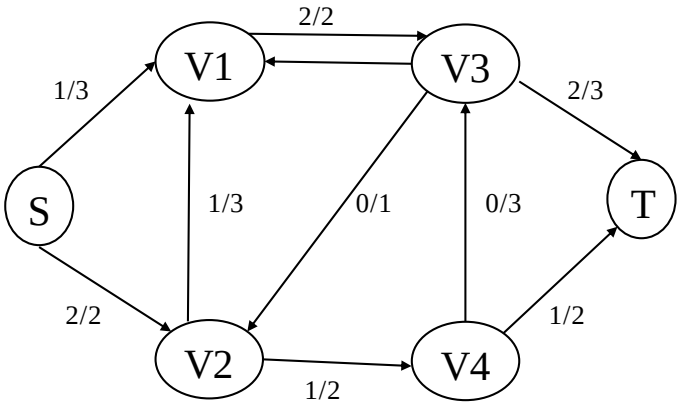
Let G = (V, E) be a flow network. Let s be the source of the network, and let t be the sink. A flow in G is a real-valued function f: V x V→R such that the following three properties hold:

1. Capacity Constraint makes sure that the flow through each edge is not greater than the capacity.
   - For all u, v ∈ V, we need f (u, v) ≤ c (u, v).

In the maximum-flow problem, we are given a flow network G with source s and sink t, and we wish to find a flow of maximum value from s to t.

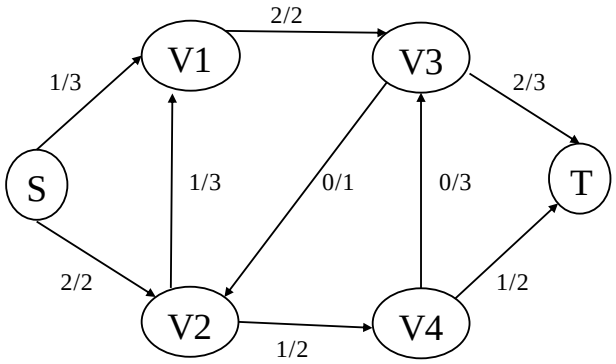
Maximum Flow :-

Maximum Flow



3 properties of Flow control hold:
1. Capacity Constraint: $F(u,v) \leq C(u,v)$
2. Skew Symmetry: $F(v,u) = - F(u,v)$
3. Flow Conservation: incoming flow = outgoing flow except S & T

Maximum Flow



Currently, the flow from the source is = 2+1 = 3
Is there any scope to increase the flow?
Is it possible to decrease the flow of anyone of the edge that can increase the overall flow.

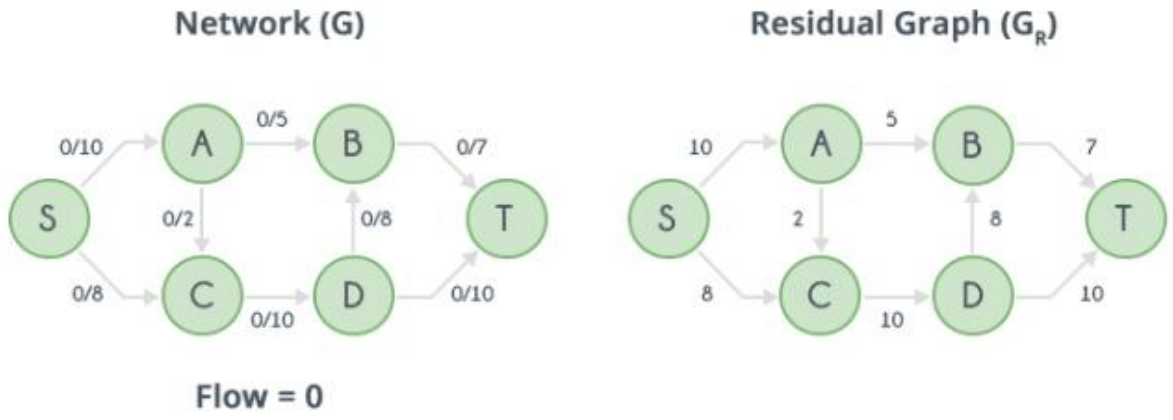
7. What is a residual network in the context of flow networks .

Answer :-

Residual Network :-

A residual network graph indicates how much more flow is allowed in each edge in the network graph. If there are no augmenting paths possible from S to T, then the flow is maximum. The result i.e. the maximum flow will be the total flow out of source node which is also equal to total flow in to the sink node.

A demonstration of working of Ford-Fulkerson algorithm is shown below with the help of diagrams.



Path 1: S - C - D - B - T ⟶ Flow = Flow + 7

Path 2: S - C - D - T ⟶ Flow = Flow + 1

Path 3: S - A - B - T ⟶ Flow = Flow + 5

Path 4: S - A - C - D - T ⟶ Flow = Flow + 2

No More Paths Left

Max Flow = 15

8. Explain the KMP algorithm by taking one sample graph.

KMP:-

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. 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

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

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

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:

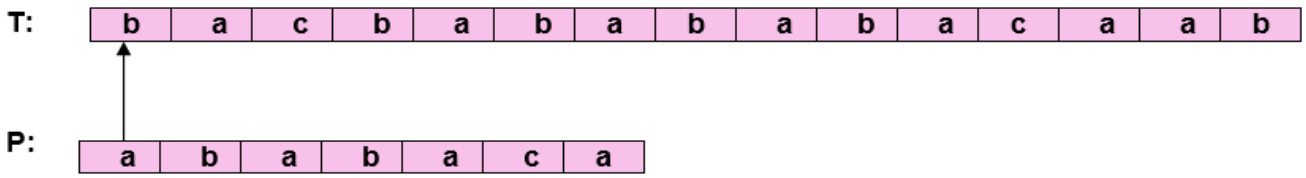
q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

Solution:

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

Step1: i=1, q=0

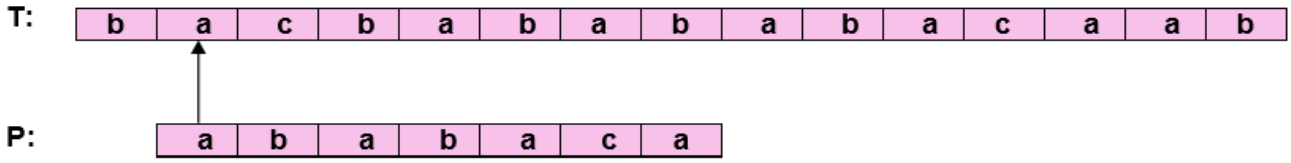
Comparing P [1] with T [1]



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

Step2: i = 2, q = 0

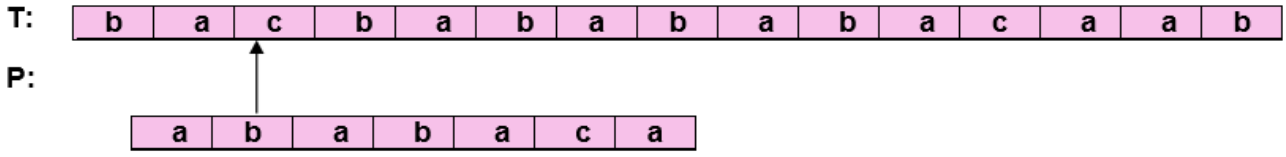
Comparing P [1] with T [2]



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

Step 3: i = 3, q = 1

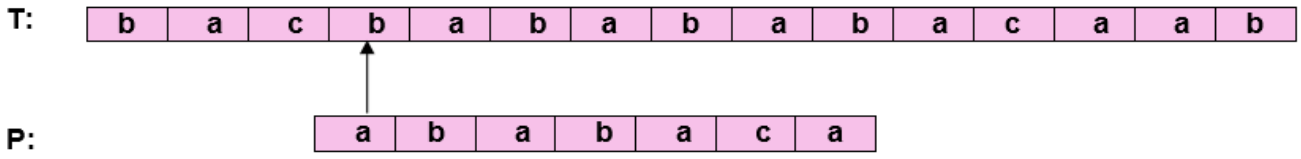
Comparing P [2] with T [3] P [2] doesn't match with T [3]



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

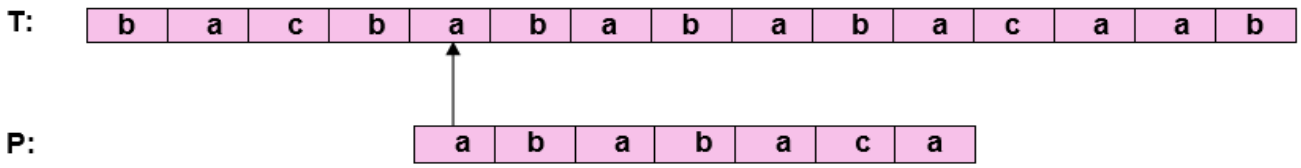
Step4: i = 4, q = 0

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



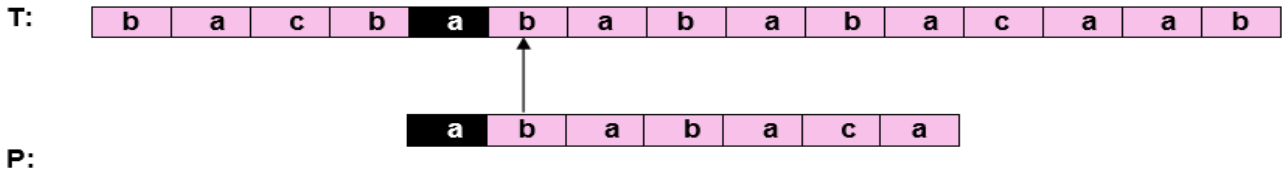
Step5: i = 5, q = 0

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



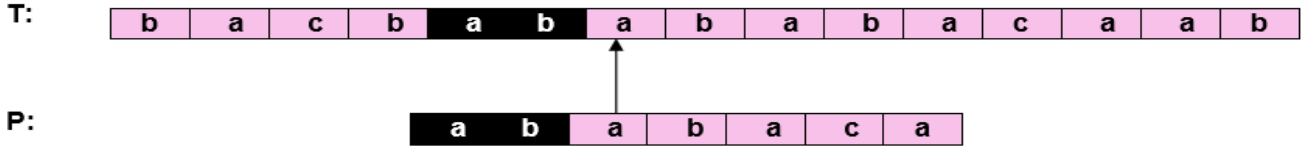
Step6: i = 6, q = 1

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



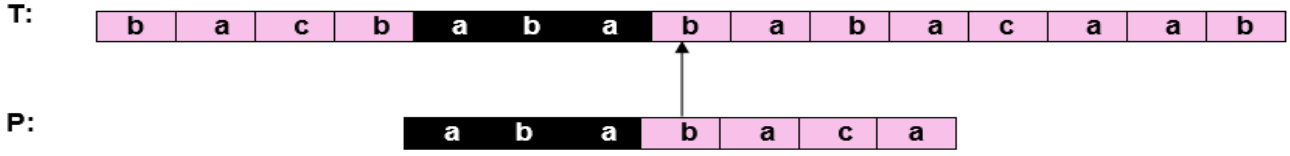
Step7: i = 7, q = 2

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



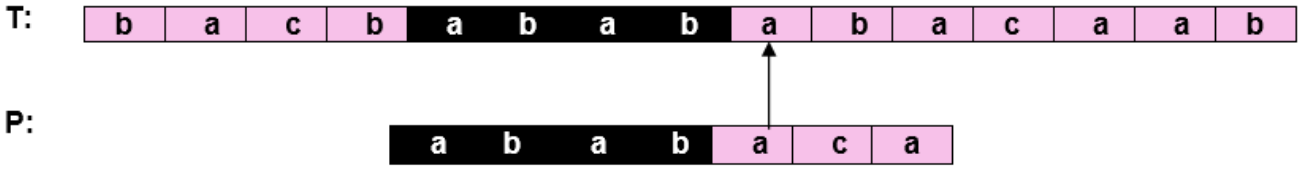
Step8: i = 8, q =3

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



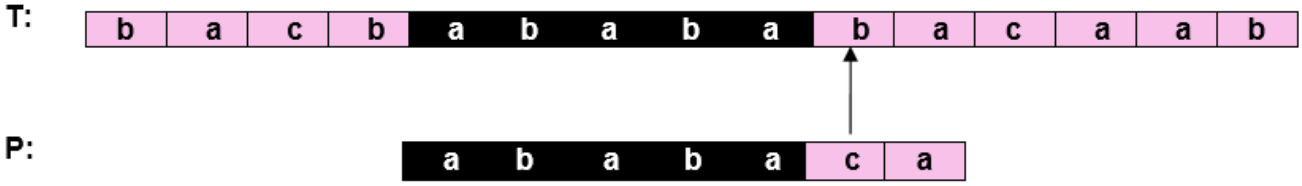
Step9: i = 9, q = 4

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



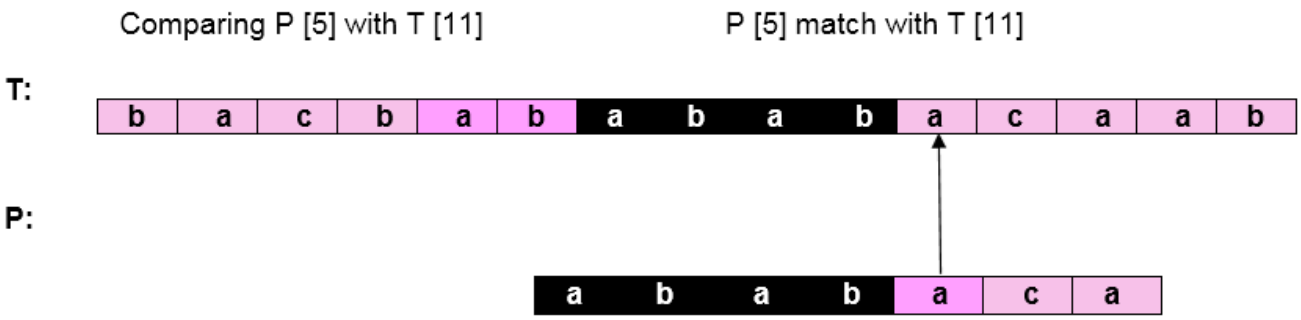
Step10: i = 10, q = 5

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

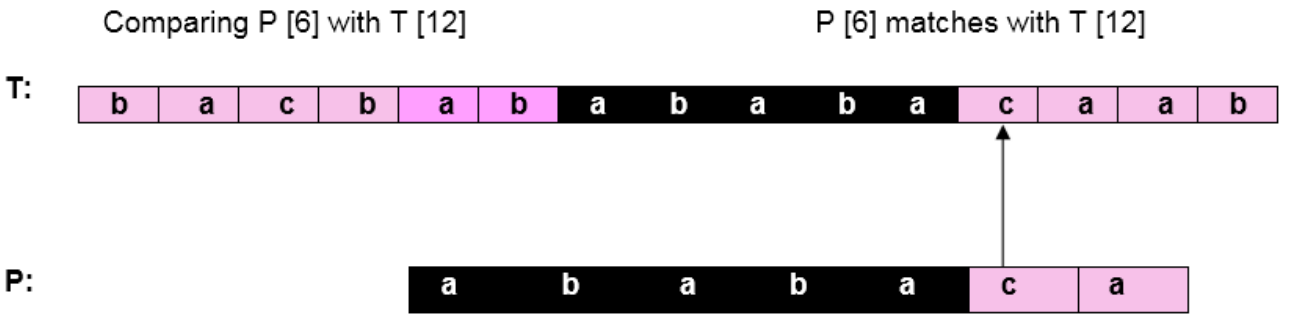


Backtracking on p, Comparing P [4] with T [10] because after mismatch q = π [5] = 3

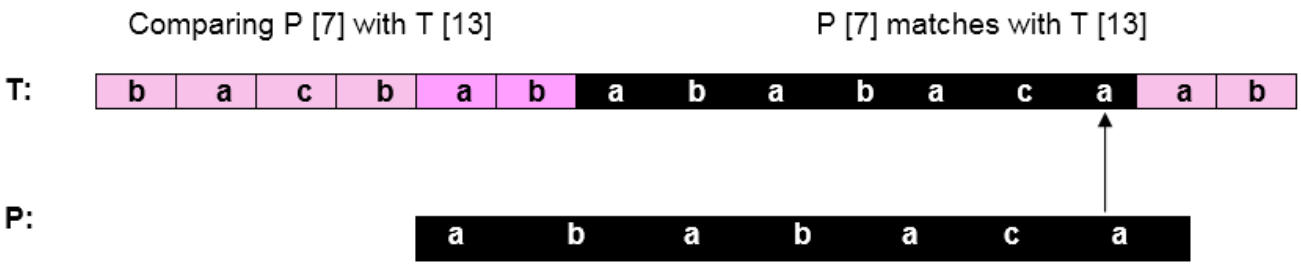
Step11: i = 11, q =4



Step12: i = 12, q = 5



Step13: i = 3, q = 6



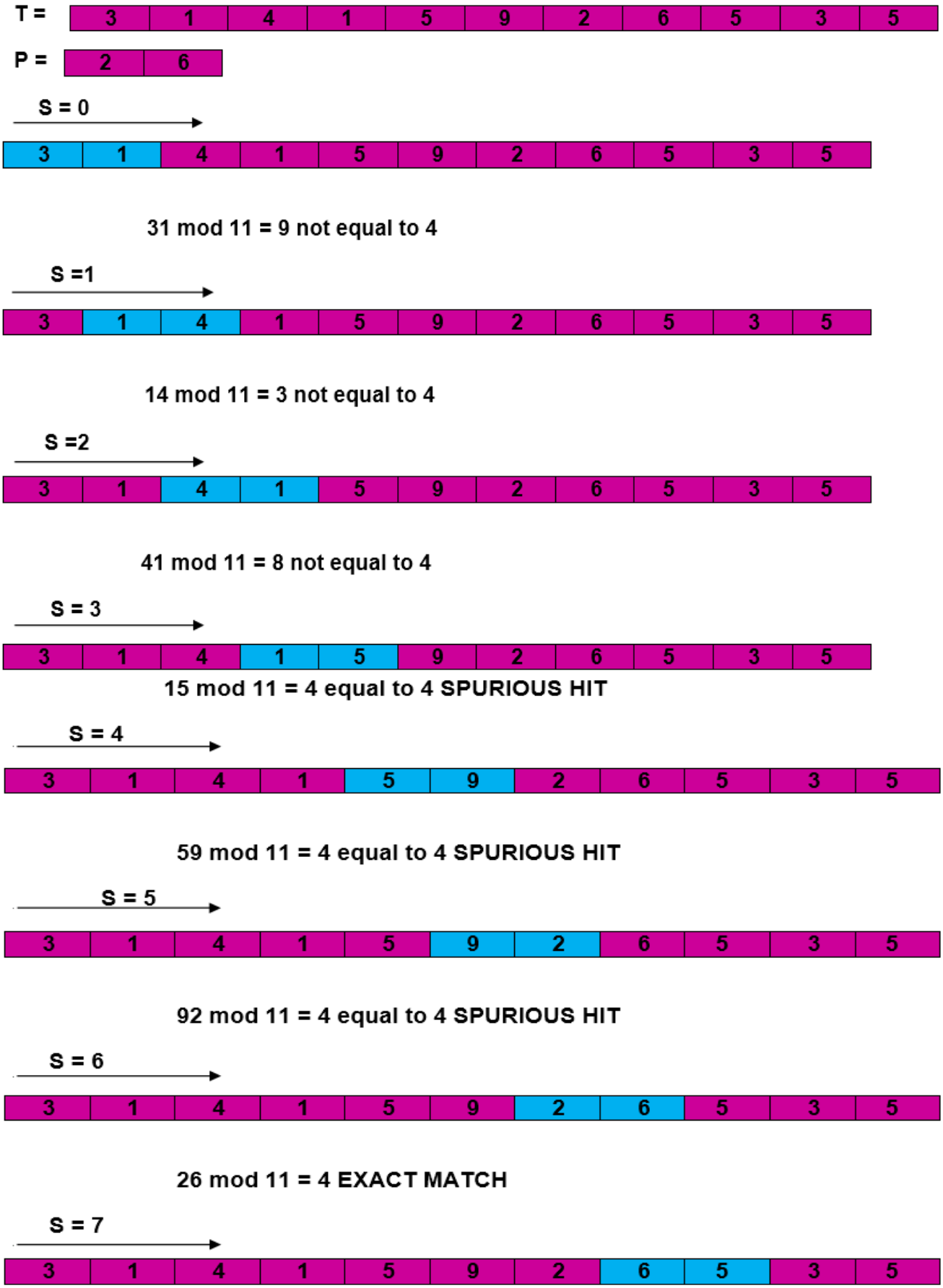
Pattern 'P' has been found to complexity occur in a string 'T.' The total number of shifts that took place for the match to be found is i-m = 13 - 7 = 6 shifts.

9. Explain the Rabin-Karp Algorithm by taking one sample graph.

Answer :

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

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


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

S = 7

65 mod 11 = 10 not equal to 4

S = 8

53 mod 11 = 9 not equal to 4

S = 9

35 mod 11 = 2 not equal to 4

The Pattern occurs with shift 6.