String Matching Algorithms:
Knuth–Morris–Pratt algorithm

Introduction

- The algorithm was conceived in 1974 by Donald Knuth and Vaughan Pratt, and independently by James H. Morris. The three published it jointly in 1977

Problem Defination

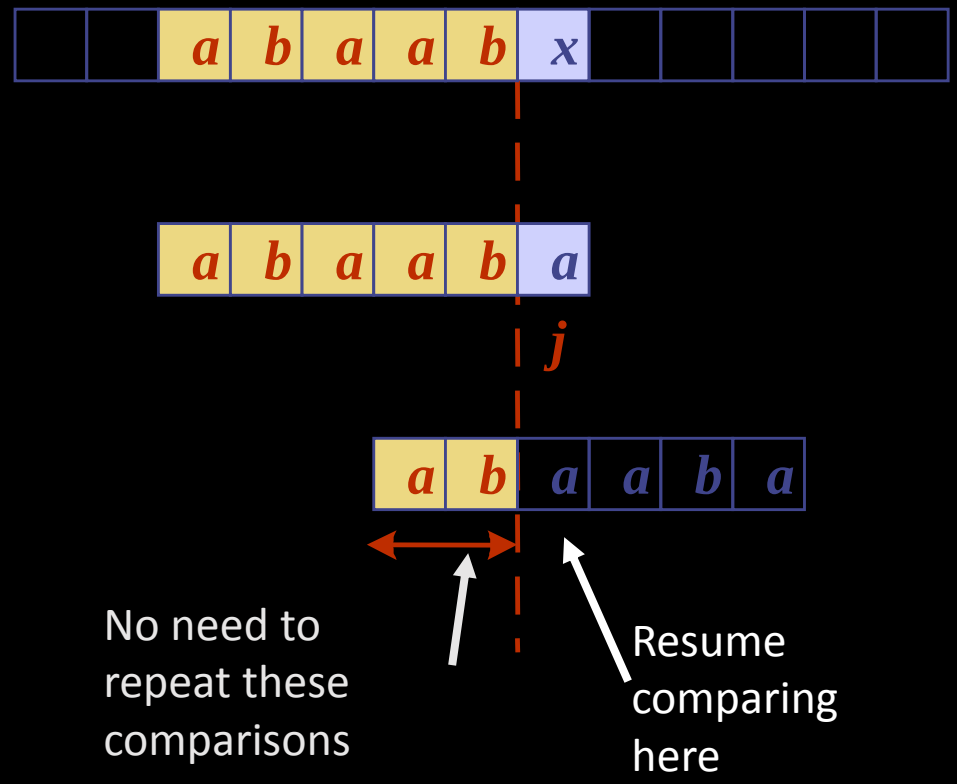
Given a string ‘S’, the problem of string matching deals with finding whether a pattern ‘p’ occurs in ‘S’ and if ‘p’ does occur then returning position in ‘S’ where ‘p’ occurs.

Drawbacks of the O(mn) Approach

- if ‘m’ is the length of pattern ‘p’ and ‘n’ the length of string ‘S’, the matching time is of the order O(mn). This is a certainly a very slow running algorithm.
- What makes this approach so slow is the fact that elements of ‘S’ with which comparisons had been performed earlier are involved again and again in comparisons in some future iterations. For example: when mismatch is detected for the first time in comparison of p[3] with S[3], pattern ‘p’ would be moved one position to the right and matching procedure would resume from here. Here the first comparison that would take place would be between p[0]=‘a’ and S[1]=‘b’. It should be noted here that S[1]=‘b’ had been previously involved in a comparison in step 2. this is a repetitive use of S[1] in another comparison.
- It is these repetitive comparisons that lead to the runtime of O(mn).

The Algorithm

- Knuth-Morris-Pratt's algorithm compares the pattern to the text in **left-to-right**, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?
- Answer: the largest prefix of $P[0..j]$ that is a suffix of $P[1..j]$



Components of KMP algorithm

- The prefix function, π

  The prefix function,π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. This information can be used to avoid useless shifts of the pattern ‘p’. In other words, this enables avoiding backtracking on 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 occurrence is found.

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:

p	a	b	a	b	a	c	a

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				

Step 3: 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			

Step 4: 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		

Step 5: 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	

Step 6: q = 7, k = 0

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 iterating 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

```
KMP_Matcher(S,p)
1 n ← length[S]
2 m ← length[p]
3 π ← Compute-Prefix-Function(p)
4 q ← 0                                  //number of characters matched
5 for i ← 1 to n                         //scan S from left to right
6     do while  q > 0 and p[q+1] != S[i]
7              do  q ← π[q]              //next character does not match
8          if p[q+1] = S[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
```

Note: KMP finds every occurrence of a ‘p’ in ‘S’. That is why KMP does not terminate in step 12, rather it searches remainder of ‘S’ for any more occurrences of ‘p’.

Illustration: given a String ‘S’ and pattern ‘p’ as follows:

S

b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

p

a	b	a	b	a	c	a

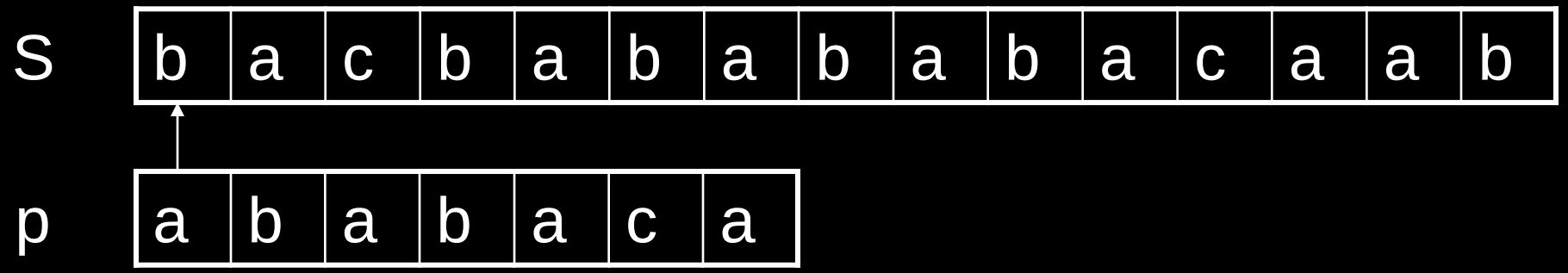
Let us execute the KMP algorithm to find whether ‘p’ occurs in ‘S’.

For ‘p’ the prefix function, π *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	1	1

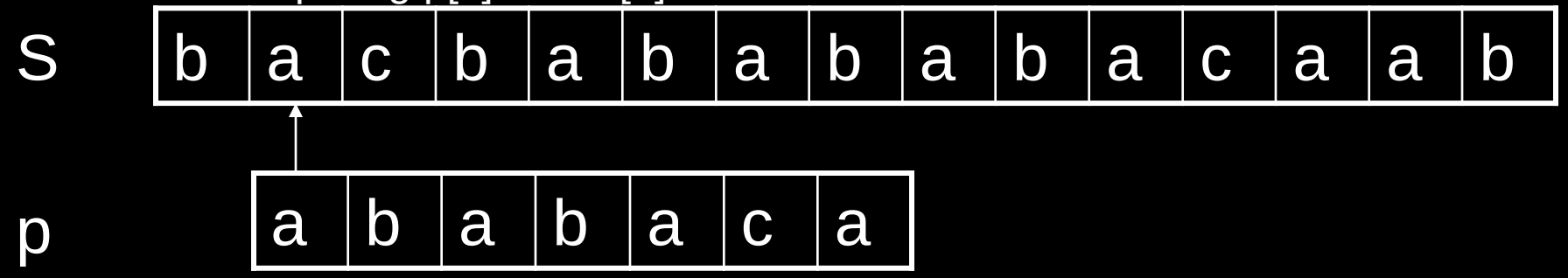
Initially: n = size of S = 15;
m = size of p = 7

Step 1: i = 1, q = 0
comparing p[1] with S[1]



P[1] does not match with S[1]. ‘p’ will be shifted one position to the right.

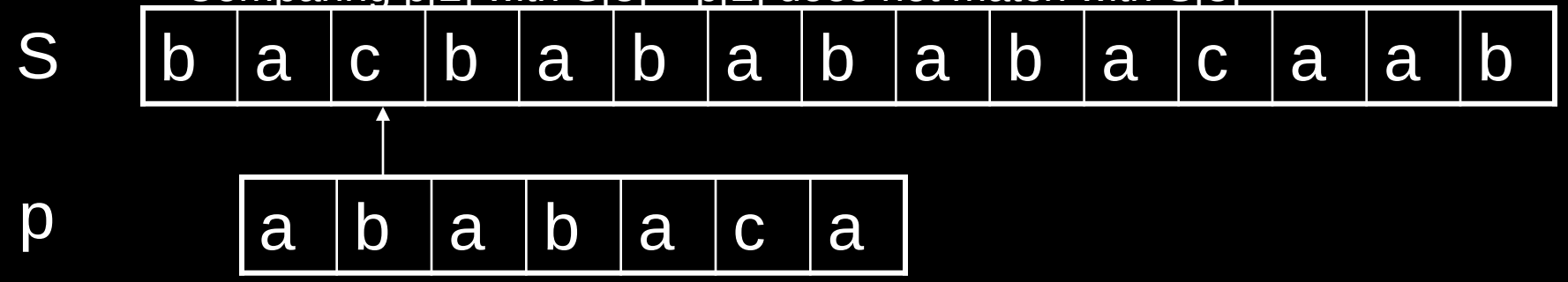
Step 2: i = 2, q = 0
comparing p[1] with S[2]



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

Step 3: i = 3, q = 1

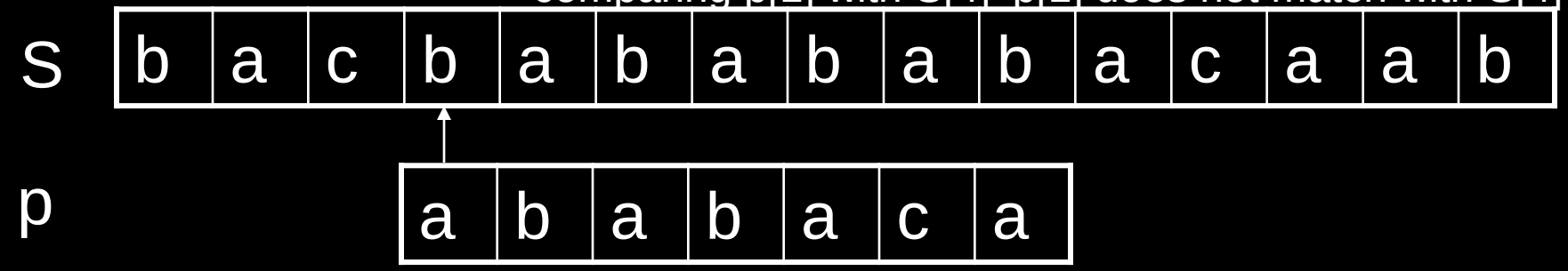
Comparing p[2] with S[3] p[2] does not match with S[3]



Backtracking on p, comparing p[1] and S[3]

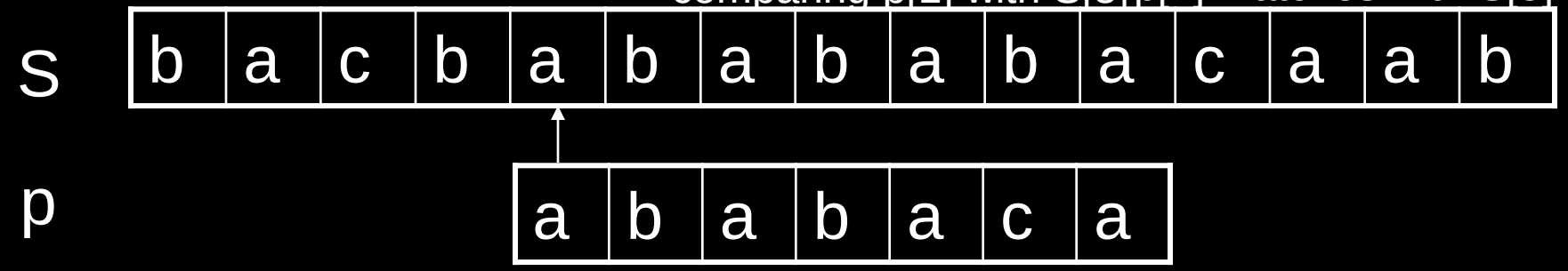
Step 4: i = 4, q = 0

comparing p[1] with S[4] p[1] does not match with S[4]



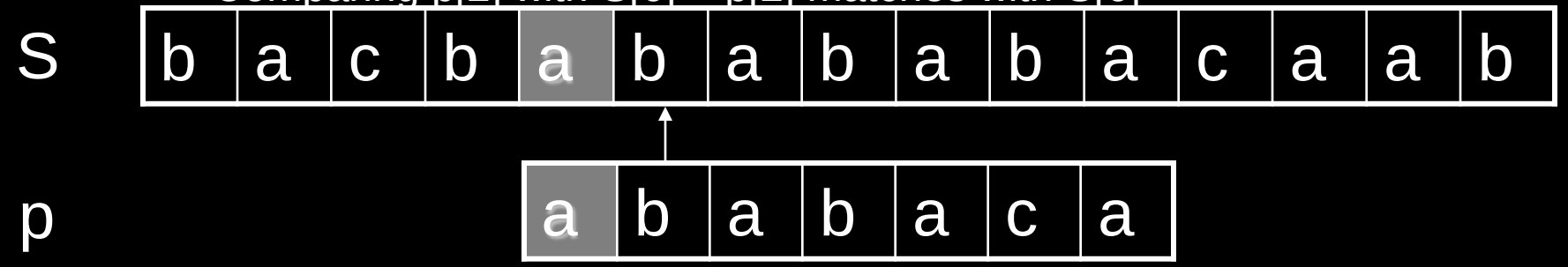
Step 5: i = 5, q = 0

comparing p[1] with S[5] p[1] matches with S[5]



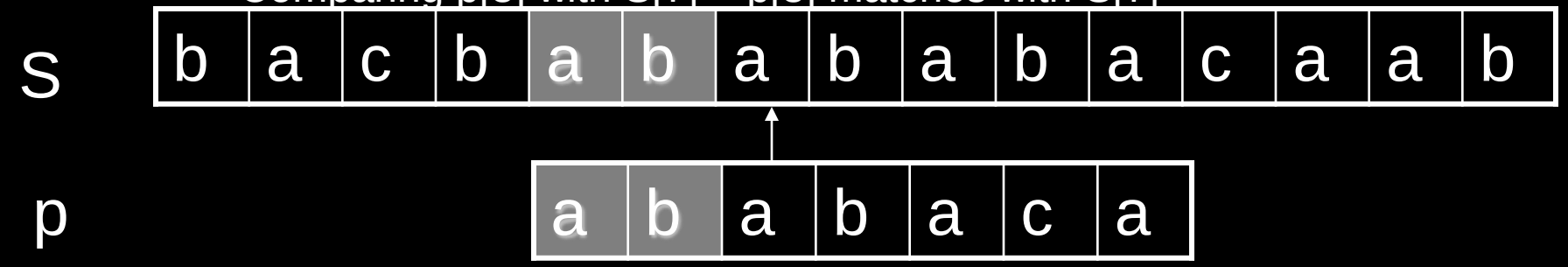
Step 6: i = 6, q = 1

Comparing p[2] with S[6] p[2] matches with S[6]



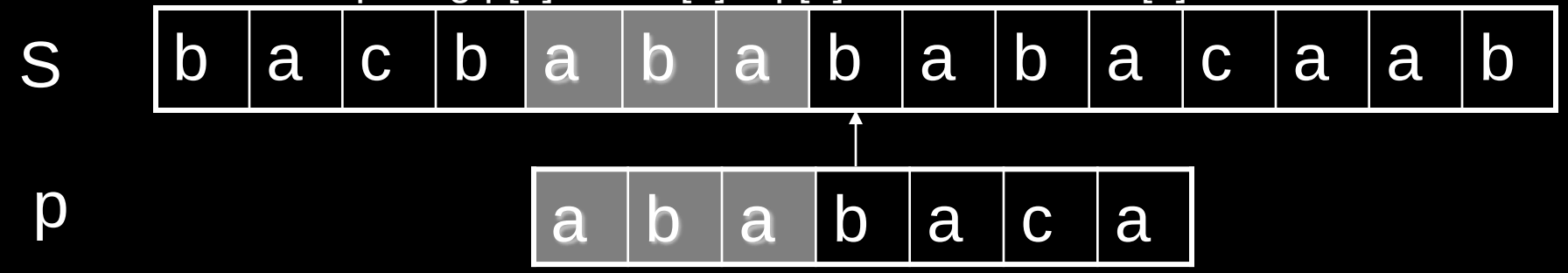
Step 7: i = 7, q = 2

Comparing p[3] with S[7] p[3] matches with S[7]



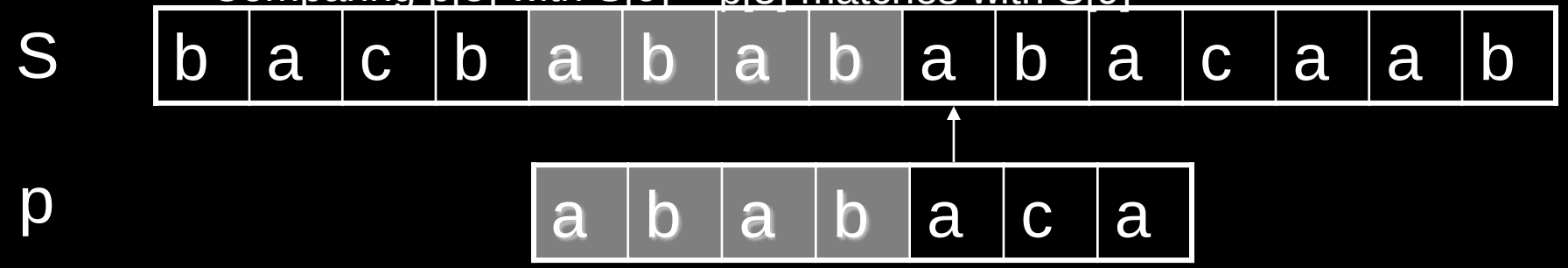
Step 8: i = 8, q = 3

Comparing p[4] with S[8] p[4] matches with S[8]



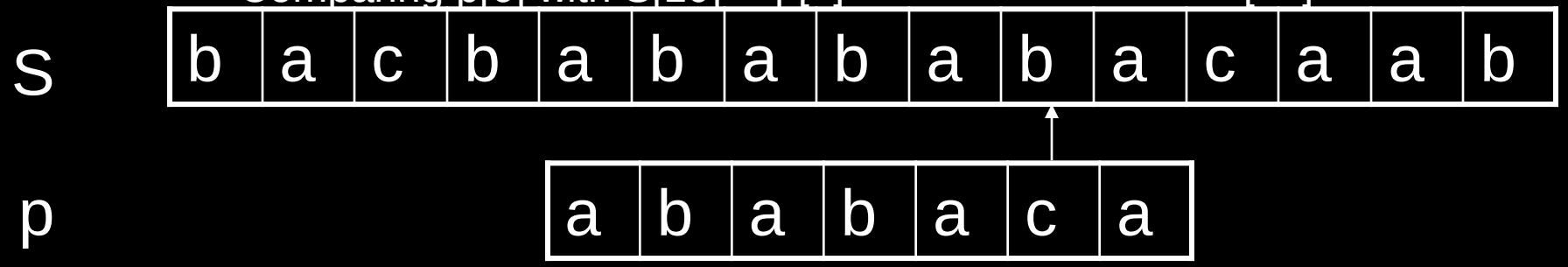
Step 9: i = 9, q = 4

Comparing p[5] with S[9] p[5] matches with S[9]



Step 10: i = 10, q = 5

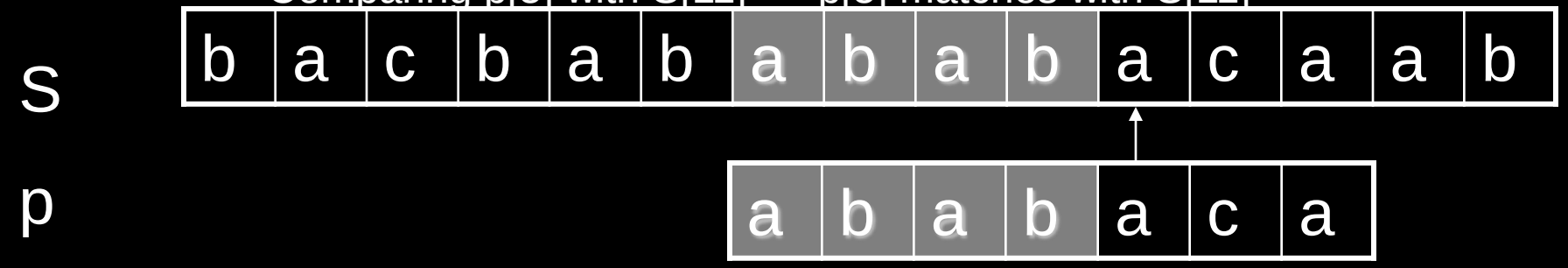
Comparing p[6] with S[10] p[6] doesn't match with S[10]



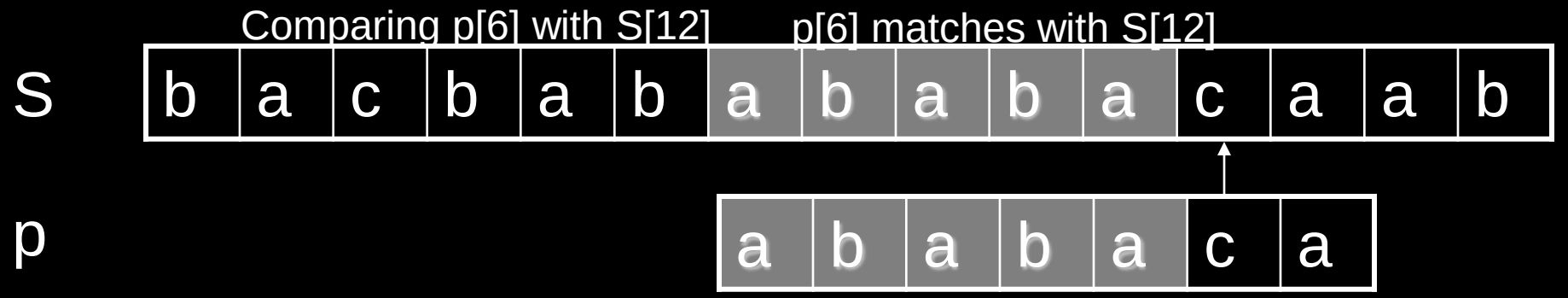
Backtracking on p, comparing p[4] with S[10] because after mismatch q = π[5] = 3

Step 11: i = 11, q = 4

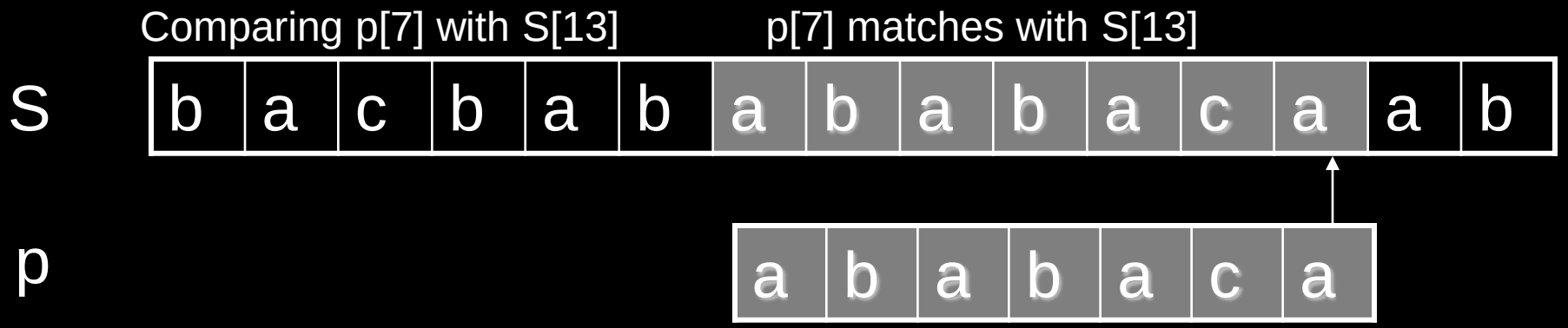
Comparing p[5] with S[11] p[5] matches with S[11]



Step 12: i = 12, q = 5



Step 13: i = 13, q = 6



Pattern ‘p’ has been found to completely occur in string ‘S’. The total number of shifts that took place for the match to be found are: i – m = 13 – 7 = 6 shifts.

Analysis

- Compute_Prefix_Function (π)

```
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
       π[q] ← k
10       return π
```

In the above pseudo code for computing the prefix function, the for loop from step 4 to step 10 runs 'm' times. Step 1 to step 3 take constant time. Hence the running time of compute prefix function is Θ(m).

- KMP_Matcher

```
1 n ← length[S]
2 m ← length[p]
3 π ← Compute-Prefix-Function(p)
4 q ← 0
5 for i ← 1 to n
6    do while  q > 0 and p[q+1] != S[i]
7          do  q ← π[q]
8    if p[q+1] = S[i]
9       then q ← q + 1
10   if q = m
11    then print "Pattern occurs with shift" i – m
12              q ← π[ q]
```

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 time, the running time is dominated by this for loop. Thus running time of matching function is Θ(n).

Time Complexity

- The running time of Knuth-Morris-Pratt algorithm is proportional to the time needed to read the characters in text and pattern. In other words, the worst-case running time of the algorithm is $\Theta(m + n)$ and it requires $O(m)$ extra space.

Applications

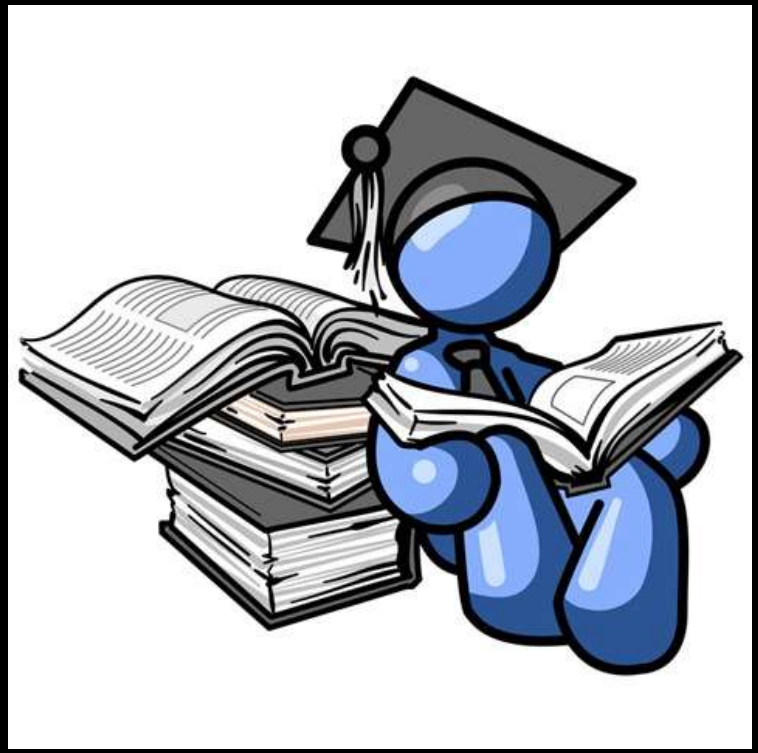
- finds its applications in many core Digital Systems and processes e.g.

  - ✓ Digital libraries
  - ✓ Screen scrapers
  - ✓ Word processors
  - ✓ Web search engines
  - ✓ Spam filters
  - ✓ Natural language processing

References

- Knuth, Donald; Morris, James H.; Pratt, Vaughan (1977). "Fast pattern matching in strings". *SIAM Journal on Computing* **6** (2): 323–350. doi:10.1137/0206024

- Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, PHI



THANK YOU
For
Being A Patient Audience