Knuth-Morris-Pratt Algorithm

- Knuth, Morris and Pratt discovered *first linear time string matching algorithm* by analysis of the naive algorithm
- It keeps the information that naive approach wasted gathered during the scan of the text. By avoiding this waste of information, it achieves a running time of $O(m + n)$.
- The implementation of Knuth-Morris-Pratt algorithm is efficient because it minimizes the total number of comparisons of the pattern against the input string.

Knuth-Morris-Pratt Algorithm

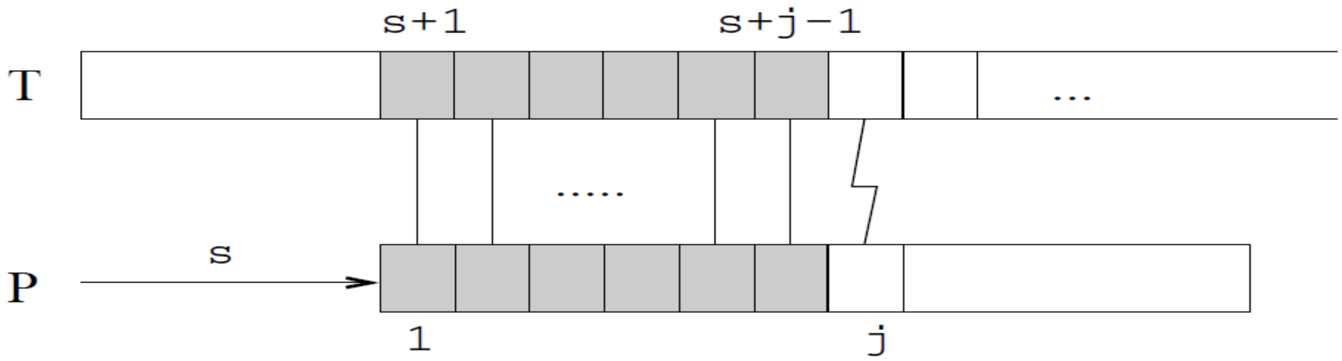
- The most expensive part of the string matching automaton method is to build the transition function δ,
  which takes $O(m^3|\Sigma|)$ time (or at least $O(m|\Sigma|)$ time).
- The KMP algorithm avoids to directly compute δ.
  Instead, it computes an auxiliary function π[1..m]
  pre-computed from pattern P in $O(m)$ time.
- The transition function δ can be obtained from array π in an efficient constant time when the algorithm runs on a text.(= array π allows δ to be computed efficiently "on the fly" as needed)

Knuth-Morris-Pratt Overview

- Achieve $\underline{O(m+n)}$ time by shortening automaton preprocessing time
- Approach:
  - don't precompute automaton's transition function
  - calculate enough transition data "on-the-fly"
  - obtain data via *alphabet-independent* pattern preprocessing
  - pattern preprocessing *compares pattern against shifts of itself*

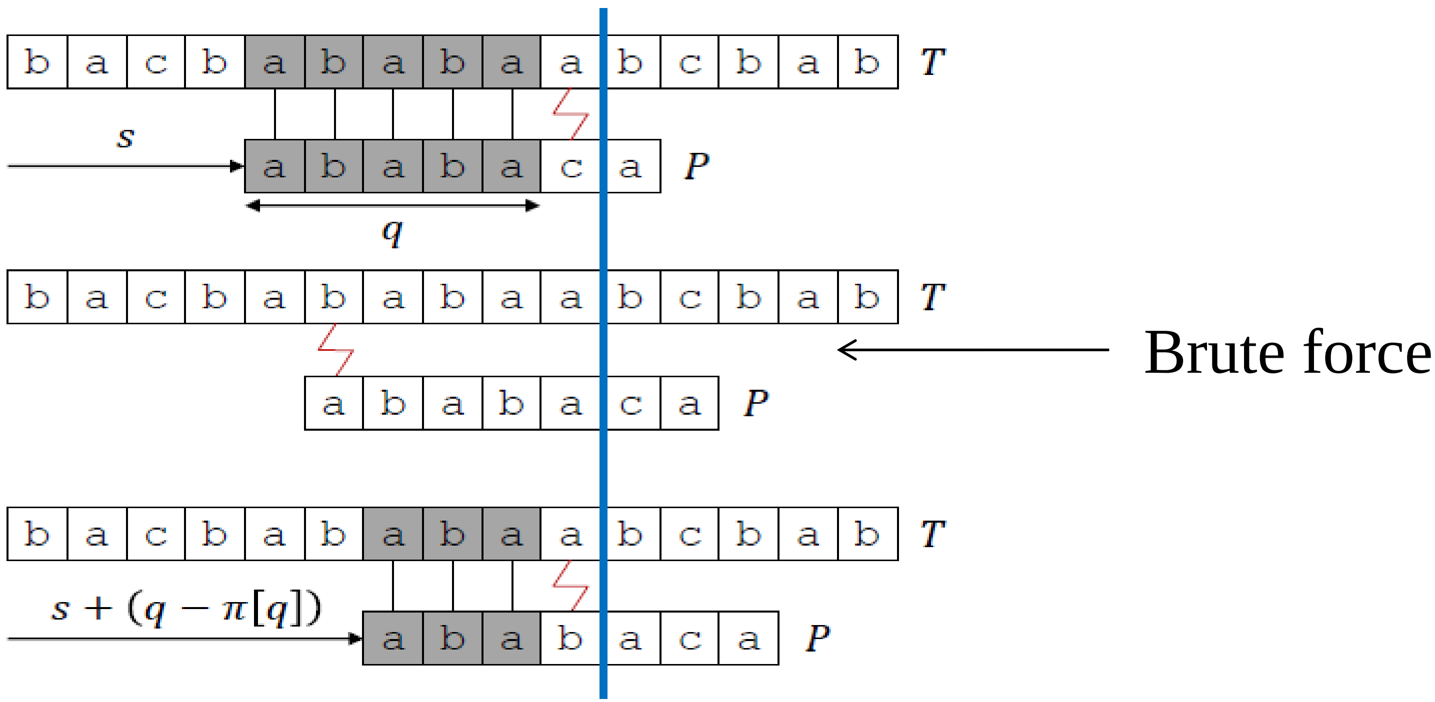
Problem with Brute force algorithm

- In the Brute-Force algorithm, if a mismatch occurs at P[j] (j>1), it only slides to right by 1 step.
- It wastes one piece of information that we've already known.
- *<u>What is that piece of information ?</u>*
  Let s be the current shift value. Since it is a mismatch at P[j], we know T[s+1..s+j-1]=P[1..j-1]



Example

- What's the next possible shift that should be tested?



$\pi[q]$ is the length of the longest prefix of P that is a **proper** suffix of P_q

The prefix function π for a pattern P:

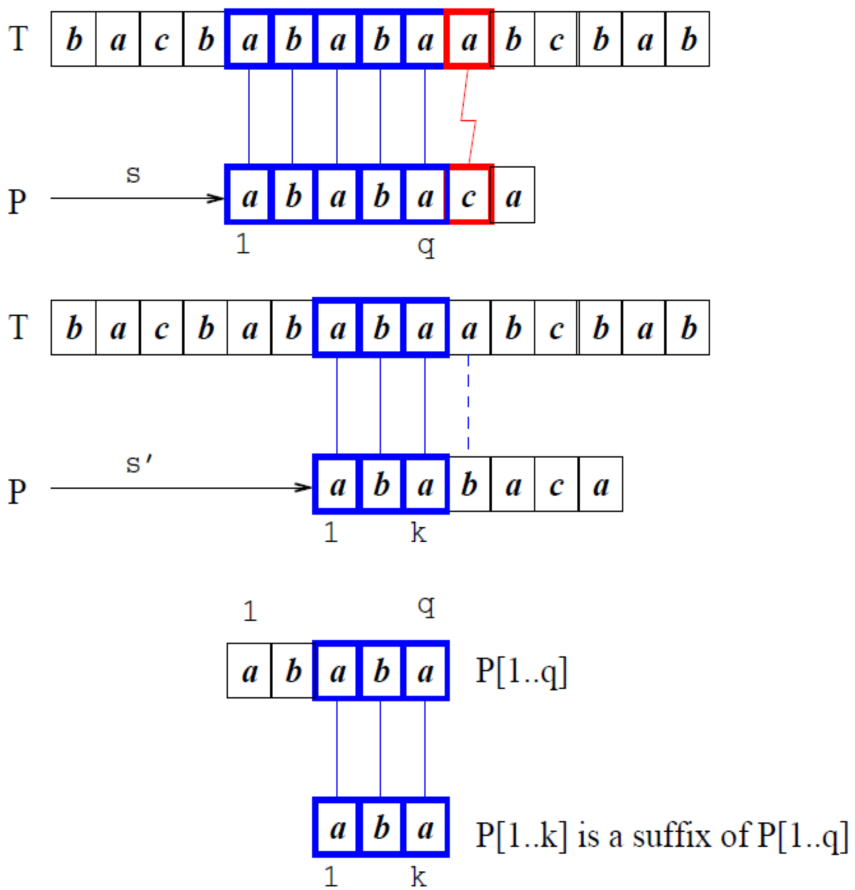
- it encapsulates the knowledge about *how the pattern P matches against shifts of itself.*

- Therefore, *the knowledge can be used to avoid the useless shifts in the naive method or to avoid to pre-compute δ in the automaton method.*

The prefix function π for a pattern P:

- Given that pattern characters P[1..q] match text characters T[s+1..s+q], what is the least shift s'>s such that P[1..k]=T[s'+1..s'+k], where s'+k=s+q?
- The above equation is equivalent to find the largest k<q such that $P_k \sqsupset P_q$. Then, s'=s+(q-k) is the potential next valid shift.
- Given a pattern P[1..m], the prefix function for the pattern P is the function $\pi : \{1,2,...,m\} \rightarrow \{0,1,...,m-1\}$ such that $\pi[q]=\max\{k:|k<q \;\&\; P_k \sqsupset P_q\}$.
- $\pi[q]$ is the length of the longest prefix of *P* that is a **proper** suffix of Pq

The prefix function π for a pattern P:

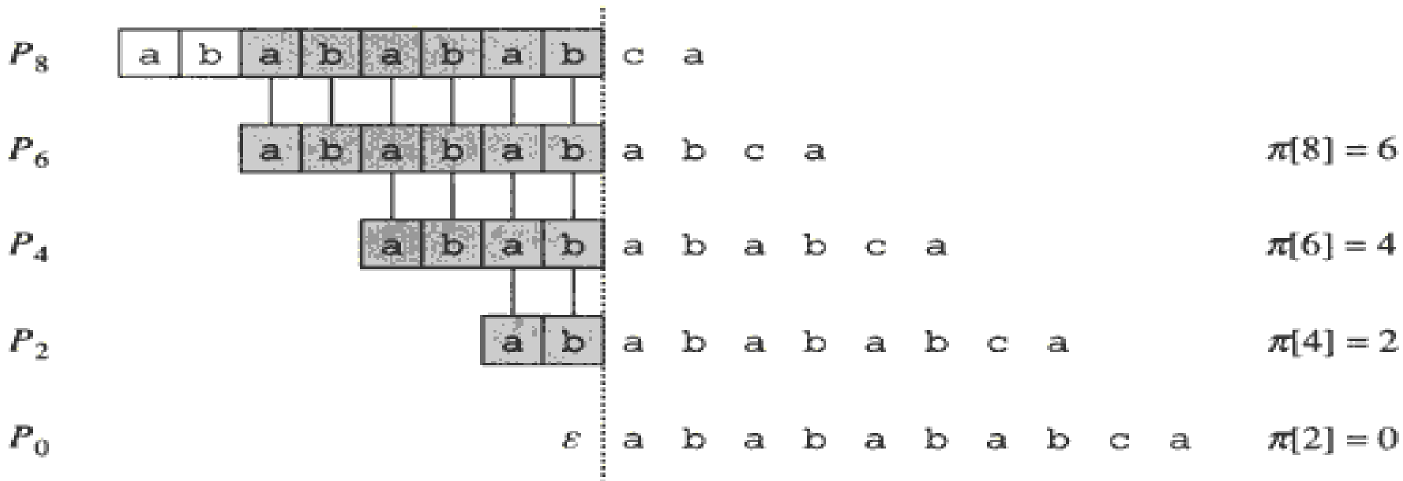


<u>s'=s+(q-k) is the potential next valid shift.</u>

q=5, k=3=π[5]
s'=s + 2

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

(a)



(b)

Figure 32.11 An illustration of Lemma 32.5 for the pattern P = ababababca and $q = 8$. **(a)** The π function for the given pattern. Since $\pi[8] = 6$, $\pi[6] = 4$, $\pi[4] = 2$, and $\pi[2] = 0$, by iterating π we obtain $\pi^*[8] = \{6, 4, 2, 0\}$. **(b)** We slide the template containing the pattern P to the right and note when some prefix P_k of P matches up with some proper suffix of P_8; this happens for $k = 6$, 4, 2, and 0. In the figure, the first row gives P, and the dotted vertical line is drawn just after P_8. Successive rows show all the shifts of P that cause some prefix P_k of P to match some suffix of P_8. Successfully matched characters are shown shaded. Vertical lines connect aligned matching characters. Thus, $\{k : k < q \text{ and } P_k \sqsupset P_q\} = \{6, 4, 2, 0\}$. The lemma claims that $\pi^*[q] = \{k : k < q \text{ and } P_k \sqsupset P_q\}$ for all q.

KMP-MATCHER(T,P)

```
KMP-MATCHER(T, P)
n ← length[T]
m ← length[P]
π ← COMPUTE-PREFIX-FUNCTION(P)
q ← 0                                   Number of characters matched
for i ← 1 to n                          Scan the text from left to right
     do while q > 0 and P[q + 1] ≠ T[i]
              do q ← π[q]               Next character does not match
        if P[q + 1] = T[i]
           then q ← q + 1               Next character matches
        if q = m                        Is all of P matched?
           then print "Pattern occurs with shift" i − m
                q ← π[q]                Look for the next match
```

COMPUTE-PREFIX-FUNCTION(P)

```
COMPUTE-PREFIX-FUNCTION(P)
m ← length[P]
π[1] ← 0
k ← 0
for q ← 2 to m
     do while k > 0 and P[k + 1] ≠ P[q]
            do k ← π[k]
        if P[k + 1] = P[q]
          then k ← k + 1
        π[q] ← k
return π
```

Analysis of KMP Algorithm

- Computing the prefix function takes time $\Theta(m)$
  - outer **for** loop takes time $\Theta(m)$
  - amortized cost of **for** loop body is $O(1)$
  - independent from the alphabet size
- Matching time on a text of length n is $\Theta(n)$

- Total time is $\underline{O(m + n)}$