COMP 6651 - Design and Analysis of Algorithms

Brigitte Jaumard

Fall 2019

String Matching

Search Text (Overview)

- The task of string matching
- The naive algorithm
  - How would you do it?
- The Rabin-Karp algorithm
  - Ingenious use of primes and number theory
- The Knuth-Morris-Pratt algorithm
  - Let a (finite) automaton do the job
  - This is optimal

Strings

- **String**: sequence of characters
- Examples of strings
  - Java program
  - HTML document
  - DNA sequence
  - Digitized sequence
- An alphabet Σ is the set of possible characters for a family of strings
- Examples of alphabets
  - ASCII
  - Unicode
  - {0,1}
  - {Q, D, N, P}
- Let P be a string of size m
  - A **substring** $P[i..j]$ is the subsequence of P consisting of the characters between i and j
  - A **prefix** of P is a substring of the type $P[0..i]$
  - A **suffix** of P is a substring of the type $P[i..m-1]$
- Given strings T (text) and P (pattern), the **string matching problem** consists of finding a / all substring(s) of T that is (are) equal to P
- Examples
  - Text editors
  - Search engines
  - Biological research

The task of string matching

- Given
  - A text *T* of length *n* over finite alphabet Σ

T[0] T[n-1]

m	a	n	a	m	a	n	a	p	a	t	i	p	i	t	i	p	i

  - A pattern *P* of length *m* over finite alphabet Σ

P[0] P[m-1]

p	a	t	i

- Output
  - All occurrences of *P* in *T*

$T[s+1..s+m] = P[1..m]$

m	a	n	a	m	a	n	a	p	a	t	i	p	i	t	i	p	i

Shift *s*

p	a	t	i

Three different algorithms

Algorithm	Preprocessing time	Matching time
Naïve	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Knuth-Morris-Prat	$\Theta(m)$	$\Theta(n)$

The Naïve Algorithm

- The naïve string matching algorithm compares the pattern *P* with the text *T* for each possible shift of *P* relative to *T*, until either
  - A match if found or
  - All placements of the pattern have been tried
- Complexity $O(mn)$
- Example of worst case
  - $T = aaaaaaaah$
  - $P = aaah$

The Naive Algorithm (2/2)

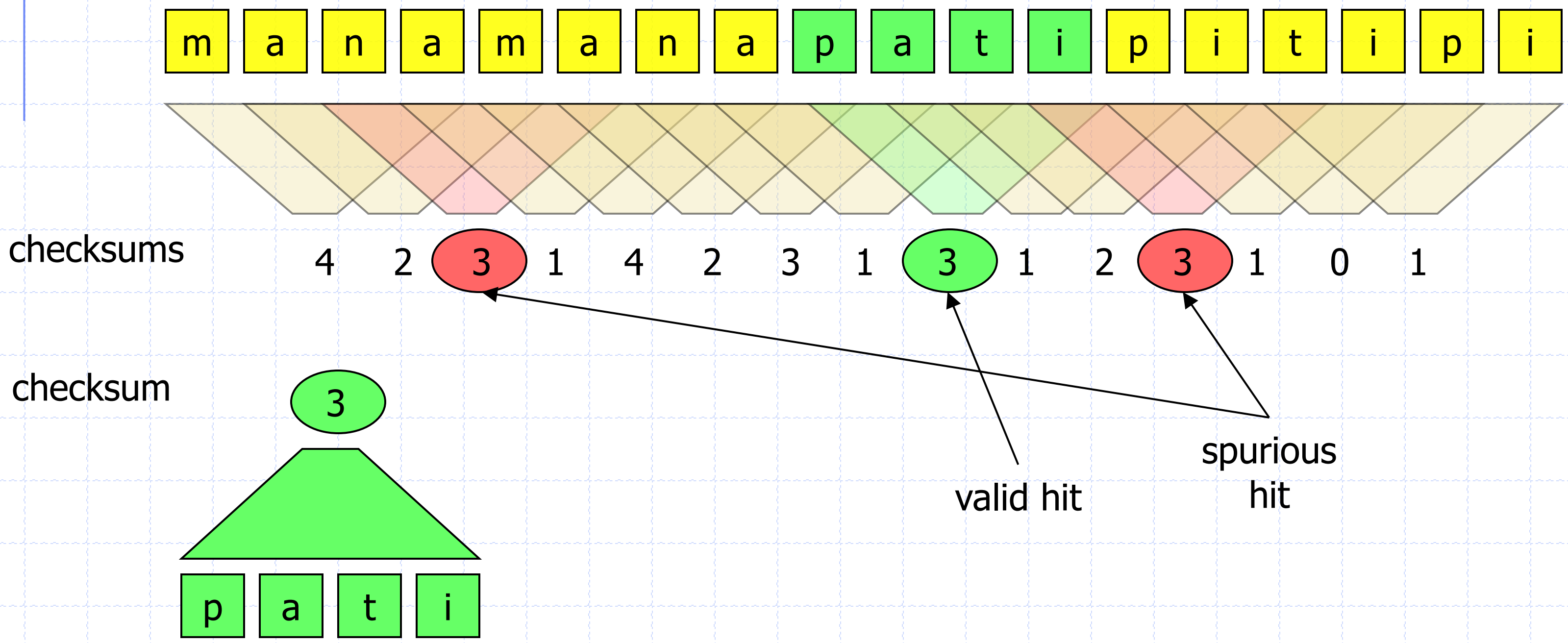
Naive-String-Matcher(*T,P*)

```
n ← length(T)
m ← length(P)
for s ← 0 to n-m do
        if P[1..m] = T[s+1 .. s+m] then
                return "Pattern occurs with shift s"
        endif
endfor
```

- Worst case running time $O((n-m+1)\ m)$ = **$O(mn)$**

The Rabin-Karp-Algorithm

- Idea: Compute
  - checksum for pattern *P* and
  - checksum for each sub-string of *T* of length *m*



Assume $\Sigma = \{0, 1, \ldots, 9\}$ so that each character is a digit

- A string of k consecutive characters → length-k decimal number
- String 31415 → decimal number 31,415

The Rabin-Karp Algorithm

- Pattern $P[1..m]$, let p denote its corresponding value.
- Text $T[1..n]$, let t_s denote the decimal value of the length-m substring $T[s+1..s+m]$, for $s=0, 1, .., n-m$.
- $t_s = p$ if and only if $T[s+1..s+m] = P[1..m]$, i.e., t_s is a valid shift if and only if $t_s = p$.
- Let us show that:
  - p can be computed in $\Theta(m)$
  - All the t_s values can be computed in $\Theta(n-m+1)$

Computing p in $\Theta(m)$

- $p = P[1..m]$
- Use Horner's rule

$$p = P[m] + 10\,(\,P[m-1] + 10\,(\,P[m-2] + \ldots + 10\,(\,P[2] + 10\,p[1]\,)\ldots\,)\,)$$

- Example: 1815
  - $1815 = 5 + 10 \times 1 + 10^2 \times 8 + 10^3 \times 1$
    $= 5 + 10 \times (\,1 + 10 \times (\,8 + 10 \times 1\,)\,)$

Computing all the t_s values in $\Theta(n-m+1)$

- $t_{s+1} = 10\,(\,t_s - 10^{m-1}\; T[s+1]) + T[s+m+1]$
- Example
  - $m = 5$ and $t_s = 31415 = T[s+1..s+m]$
  - Remove the high-order digit $T[s+1] = 3$ and bring the low-order digit $T[s+5+1] = 2$
  - $t_{s+1} = 10\,(\,31415 - 10000 \times 3) + 2 = 14152$
- Subtracting $10^{m-1}\; T[s+1]$ removes the high-order digit from t_s
  - Assuming 10^{m-1} is pre-computed, cost is only one multiplication
- Multiplying the result by 10 shifts the number left one position, and adding $T[s+m+1]$ brings in the appropriate low-order digit
- Overall complexity: O(1) for each t_s

One difficulty …

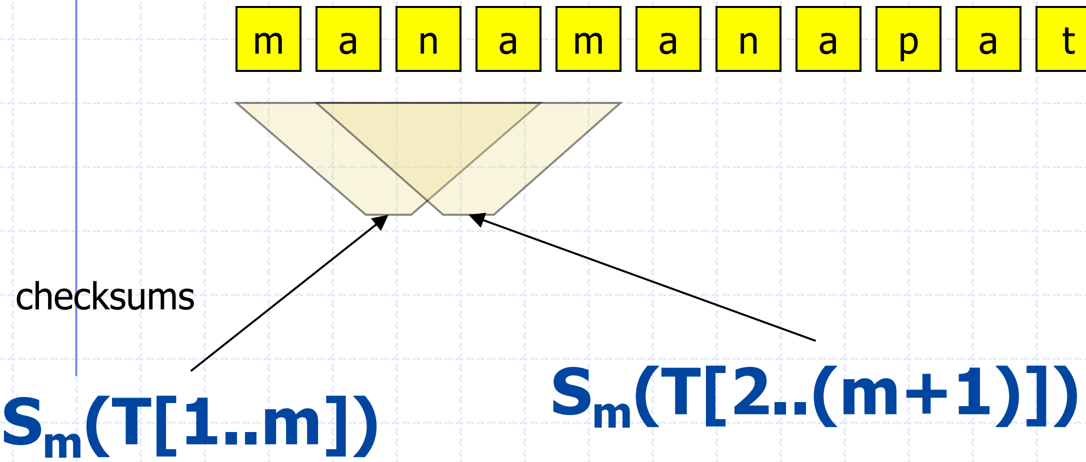
- p and t_s may be too large …
- **Cure:** compute p and t_s values modulo a suitable modulus q
- We can easily compute p modulo q in $\Theta(m)$ time and all the t_s modulo q in $\Theta(n-m+1)$
- How to choose q? As a prime such that $10q$ just fits within one computer word
- With a d-ary alphabet, $\{0, 1, \ldots, d-1\}$, choose q so that dq fits within a computer word

The solution of working modulo q is not perfect

- $t_s \neq p \pmod{q}$ implies that $t_s \neq p$: shift
- **BUT** $t_s \equiv p \pmod{q}$ does not imply that $t_s = p$
- **IDEA**
  - See the test $t_s \equiv p \pmod{q}$? as a fast heuristic to rule out invalid shift s.
  - Any shift s for which $t_s \equiv p \pmod{q}$ must be tested further to see if s is really valid or if we just have a *spurious hit*. If q is large enough, then we can hope that spurious hits occurs infrequently enough that the cost of the extra checking is low.

Computing the Checksums of the Text

- Start with $S_m(T[1..m])$



- $t_s = S_m(T[1..m])$; $t_{s+1} = S_m(T[2..(m+1)])$
- $S_m(T[2..(m+1)]) \equiv d\big(S_m(T[1..m]) - hT[1]\big)+T[m+1] \qquad (\text{mod } q)$
- $h = d^{m-1} \ (\text{mod } q)$

The Rabin-Karp Algorithm

Rabin-Karp-Matcher(T,P,d,q)

```
n ← length(T)
m ← length(P)
h ← d^(m-1) mod q
p ← 0
t_0 ← 0
for i ← 1 to m do
        p ← (d p + P[i]) mod q
        t_0 ← (d t_0 + T[i]) mod q
endfor
for s ← 0 to n-m do
        if p = t_s then
            if P[1..m] = T[s+1..s+m] then return "Pattern occurs with shift" s
        endif

        if s < n-m then t_(s+1) ← d(t_s-T[s+1]h) + T[s+m+1] mod q endif
endfor
```

Checksum of the pattern P

Checksum of T[1..m]

Checksums match Now test for false positive

Update checksum for T[s+1..s+m] using checksum T[s..s+m-1]

Performance of Rabin-Karp

- The worst-case running time of the Rabin-Karp algorithm is $O(m\,(n-m+1))$

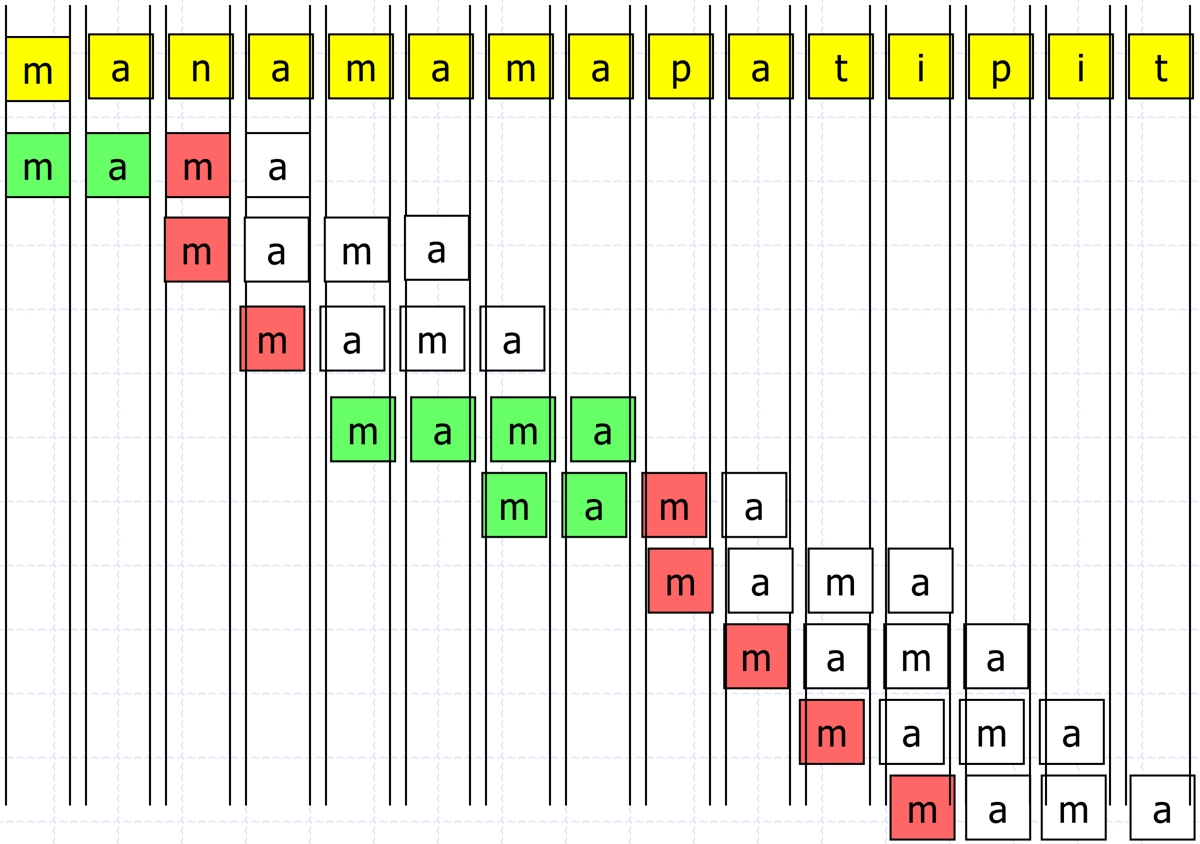
- Probabilistic analysis
  - The probability of a false positive hit for a random input is $1/q$
  - The expected number of false positive hits is $O(n/q)$
  - The expected run time of Rabin-Karp is
    $O(n + m\,(v+n/q))$
    if v is the number of valid shifts (hits)

- If we choose $q \geq m$ and have only a constant number of hits, then the expected run time of Rabin-Karp is $O(n + m)$.

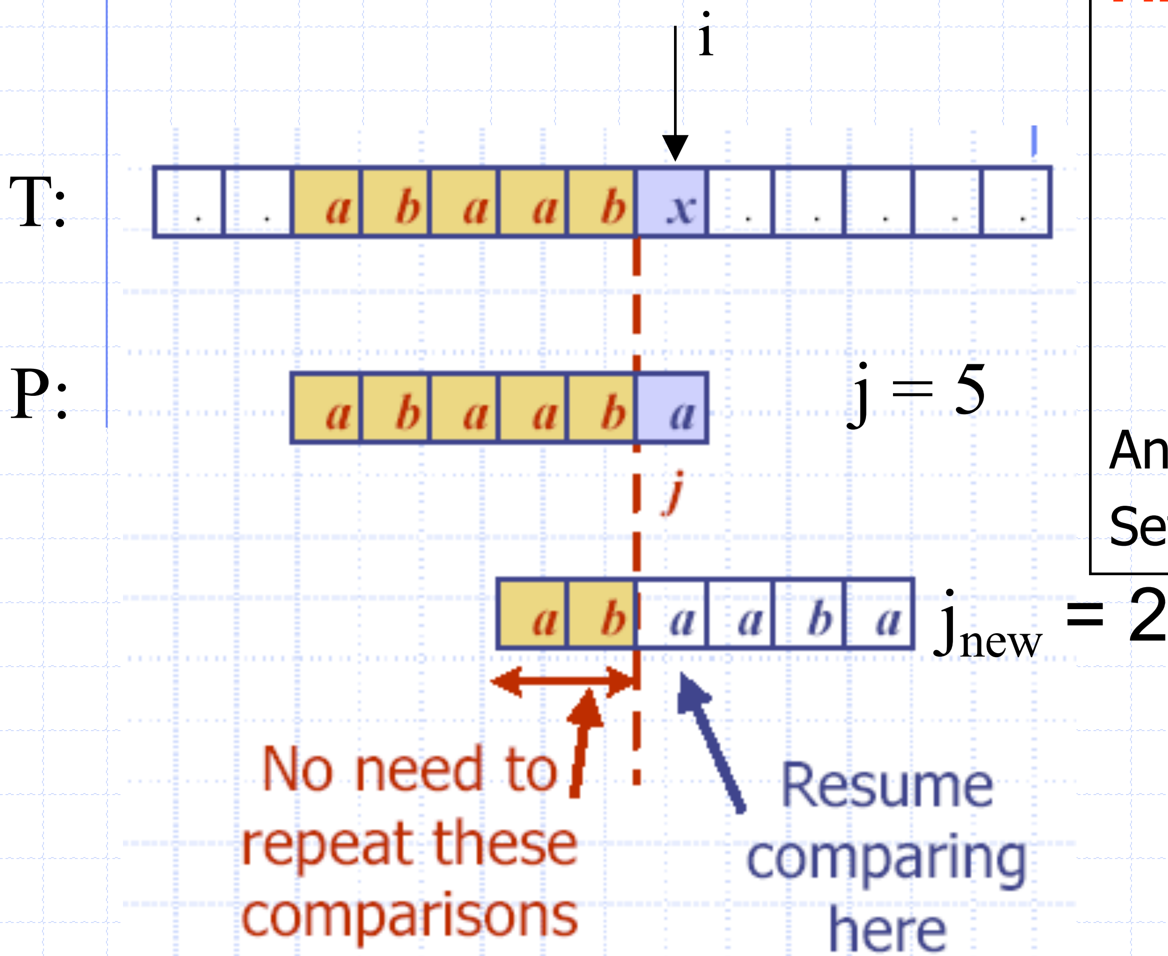
The Knuth-Morris-Pratt (KMP) Algorithm

- The KMP algorithm looks for the pattern in the text in a left-to-right order
- Shifts the pattern intelligently: If a mismatch occurs between the text and pattern *P* at *P[j]*, what is the most we can shift the pattern to avoid wasteful comparisons?
- Answer : the largest prefix of *P[0 .. j-1]* that is a suffix of *P[1 .. j-1]*

Knuth-Morris-Pratt: The Principle



The Knuth-Morris-Pratt (KMP) Algorithm Example: Shift



Find largest prefix (start) of:
"a b a a b" (P[0..j-1])

which is suffix (end) of:
"b a a b" (P[1 .. j-1])

Answer: "a b"

Set $j = 2$ // the new j value

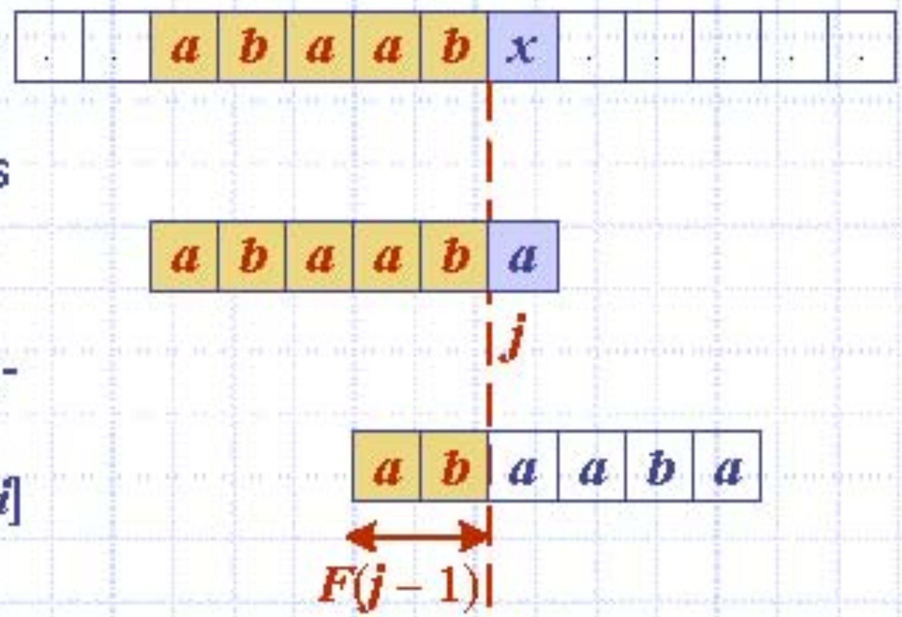
The Knuth-Morris-Pratt (KMP) Algorithm Components

- The Failure function
  - Encapsulates knowledge about how the pattern matches against shifts of itself
  - Used to avoid useless shifts of the pattern '*p*'. In other words, this enables avoiding backtracking on the string '*S*'.
  - Represented by an array, like the table.
- The KMP Matcher
  - Returns the number of shifts of '*p*' after which occurrence is found.

KMP Failure Function

- Knuth-Morris-Pratt's algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself
- The **failure function** $F(j)$ is defined as the size of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$
- Knuth-Morris-Pratt's algorithm modifies the brute-force algorithm so that if a mismatch occurs at $P[j] \neq T[i]$ we set $j \leftarrow F(j-1)$

j	0	1	2	3	4	5
$P[j]$	a	b	a	a	b	a
$F(j)$	0	0	1	1	2	3



Another Example

j	0	1	2	3	4	5
substring 0 to j	A	AB	ABA	ABAB	ABABA	ABABAC
longest prefix-suffix match	none	none	A	AB	ABA	none
next[j]	0	0	1	2	3	0
notes	no prefix and suffix that are different i.e. next[0]=0 for all patterns					

The failure function (2/3)

- The failure function can be represented by an array and can be computed in $O(m)$ time
- The construction is similar to the KMP algorithm itself
- At each iteration of the while loop, either
  - i increases by one
  - The shift amount $i - j$ increases by at least one (observe that $F[j-1] < j$)
- Hence, there are no more than $2m$ iterations of the while-loop

The failure function F (3/3)

- **Input:** String P (pattern) with m characters
- **Output:** The failure function F for P, which maps j to the length of the longest prefix of P that is a suffix of $P[1..j]$

$F[0] \leftarrow 0$; $i \leftarrow 1$; $j \leftarrow 0$
While $i < m$ **do**
  If $P[i] = P[j]$ **then**
    <* we have matched $j+1$ characters *>
    $F[i] \leftarrow j+1$; $i \leftarrow i+1$; $j \leftarrow j+1$
  Else If $j > 0$ **then**
    <* j indexes just after a prefix of P that must match *>
    $j \leftarrow F[j-1]$
    Else <* no match *>
    $F[i] \leftarrow 0$; $i \leftarrow i+1$
    Endif
  Endif
Endwhile

Example

i	0	1	2	3	4	5
P	a	b	a	a	b	a
F	0	0	1	1	2	3

F[0] = 0 *; i = 1 ; j = 0*

P[i] = P[j] ?   *P[i] = P[1] = b vs. P[j] = P[0] = a →* *No*

→ *F[i] =* ***F[1] = 0*** *; i = 2 (Prefix starting at P[j] and suffix starting at P[i] do not match)*

P[i] = P[j] ?   *P[i] = P[2] = a vs. P[j] = P[0] = a →* *Yes*

→ *F[i] = j + 1 →* ***F[2] = 1*** *; i = 3 ; j = 1*

P[i] = P[j] ?   *P[i] = P[3] = a vs. P[j] = P[1] = b →* *No*

→ *j = F [j] = F[0] = 0 (j* indexes just after a prefix of *P* that must match)

P[i] = P[j] ?   *P[i] = P[3] = a vs. P[j] = P[0] = a →* *Yes*

→ *F[i] = j + 1 →* ***F[3] = 1*** *; i = 4 ; j = 1*

P[i] = P[j] ?   *P[i] = P[4] = b vs. P[j] = P[1] = b →* *Yes*

→ *j = F [j] = F[1] = 0 (j* indexes just after a prefix of *P* that must match)

P[i] = P[j] ?   *P[i] = P[4] = b vs. P[j] = P[0] = a →* *Yes*

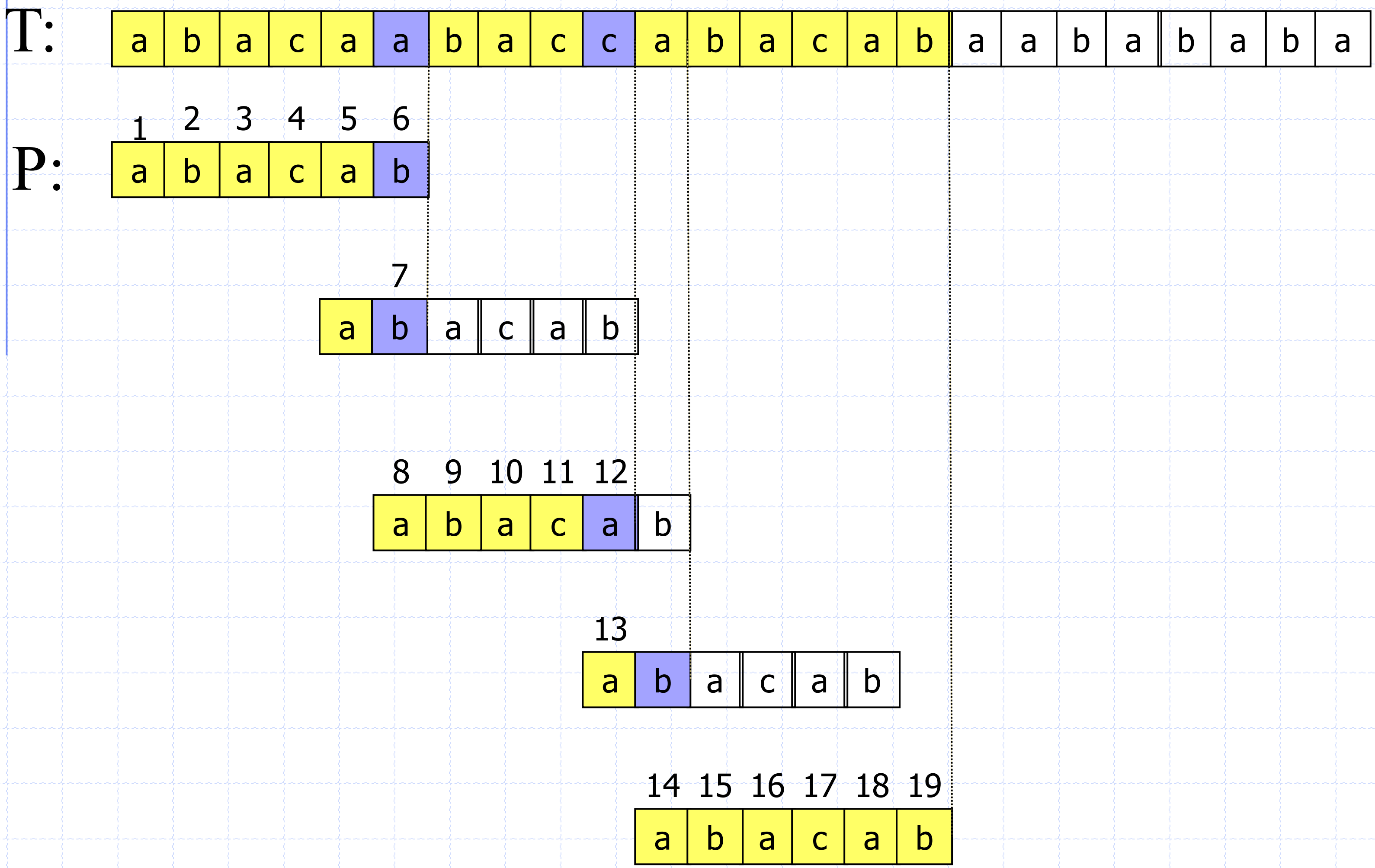
→ *j = F[j-1] = j+1 →* ***F[4] = 2*** *; i = 5 ; j = 1*

P[i] = P[j] ?   *P[i] = P[5] = a vs. P[j] = P[1] = b →* *Yes*

→ *j = F[j-1] = j+1 →* ***F[5] = 3*** *; i = 6 ; j = 2*

Another example

i	0	1	2	3	4	5
P	a	b	a	c	a	b
F	0	0	1	0	1	2



The KMP Algorithm

- The failure function can be represented by an array and can be computed in $O(m)$ time
- At each iteration of the while-loop, either
  - i increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j - 1) < j$)
- Hence, there are no more than $2n$ iterations of the while-loop
- Thus, KMP's algorithm runs in optimal time $O(m + n)$

```
Algorithm KMPMatch(T, P)
    F ← failureFunction(P)
    i ← 0
    j ← 0
    while i < n
        if T[i] = P[j]
            if j = m − 1
                return i − j { match }
            else
                i ← i + 1
                j ← j + 1
        else
            if j > 0
                j ← F[j − 1]
            else
                i ← i + 1
    return −1 { no match }
```

Complexity

- A character of T may be compared against many characters of P.
- If there is a mismatch, then the same character of T is compared against the character of P pointed by the next table.
- If there is another mismatch, then we continue comparing against the same character of T until there is either a match or we reach the beginning of P.

- **How many times can we backtrack for one character of *T*?**
- Let us assume that the first mismatch involved p_k.
- Since each backtrack leads us to a smaller index in P, we can backtrack only k times.
- However, to reach p_k we must have gone forward k times without any backtracking!
- If we assign the costs of backtracking to the forward moves, then we at most double the cost of the forward moves.
- But there are exactly n forward moves, so the number of comparisons is ***O(n).***