CPSC 535: Advanced Algorithms

Instructor: Dr. Doina Bein

Text-search Algorithms

(slides taken from http://people.cs.aau.dk/~simas/aalg04/)

- Goals of the lecture:
  - *Naive text-search algorithm and its analysis;*
  - ***Horspool** algorithm and its analysis.*
  - ***Boyer-Moore** algorithm and its analysis*
  - ***Rabin-Karp** algorithm and its analysis;*
  - ***Knuth-Morris-Pratt** algorithm ideas;*
  - *Comparison of the **advantages and disadvantages** of the different text-search algorithms.*

Text Search (or String Matching) Problem

- **Input**: Alphabet Σ and two strings T[0..n-1] and P[0..m-1], containing symbols from alphabet Σ
- **Output**: a set S containing all “shifts” $0 \leq s \leq n-m$ such that T[s..s+m-1] = P
- Example:
  Input: Σ = {a, b, …, z}, T[0..17]=“to be or not to be”, and P[0..1]=“be”
  Output: S = [3, 16]
- Chapter 32 in the textbook

(taken from Levitin, page 105)

- String P is called the *pattern:*

$$\begin{tikzcd}
t_0 & \cdots & t_i \arrow[d, leftrightarrow] & \cdots & t_{i+j} \arrow[d, leftrightarrow] & \cdots & t_{i+m-1} \arrow[d, leftrightarrow] & \cdots & t_{n-1} & \text{text } T \\
 & & p_0 & \cdots & p_j & \cdots & p_{m-1} & \text{pattern } P & &
\end{tikzcd}$$

- A brute-force algorithm:
  - Until string is exhausted do the following:
    - Align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right In the latter case
    - If a match, add index to the list of indices S
    - If not, shift the pattern one position to the right and resume the character comparisons, starting again with the first character of the pattern and its counterpart in the text.
  - Note that the last position in the text that can still be a beginning of a matching substring is $n - m$ (provided the text positions are indexed from 0 to $n - 1$).
  - Beyond that position, there are not enough characters to match the entire pattern; hence, the algorithm need not make any comparisons there.

Simple Algorithm

- Uses brute force

```
ALGORITHM BruteForceStringMatch(T [0..n − 1], P [0..m − 1] )
//Implements brute-force string matching
//Input: An array T [0..n − 1] of n characters representing a text and
// an array P [0..m − 1] of m characters representing a pattern
//Output: The indices of the characters in the text that start a
// matching substring or [] (empty set) if the search is unsuccessful
S = [] // empty list
for i ← 0 to n − m do
  match ← True
  for j ← 0 to m-1 do
    if P [j ] ≠ T [i + j ] then
      match ← False
      break // exit the for loop
    endif
  endfor
  if j == m then
    S = S ○ i
  endif
endfor
Return S
```

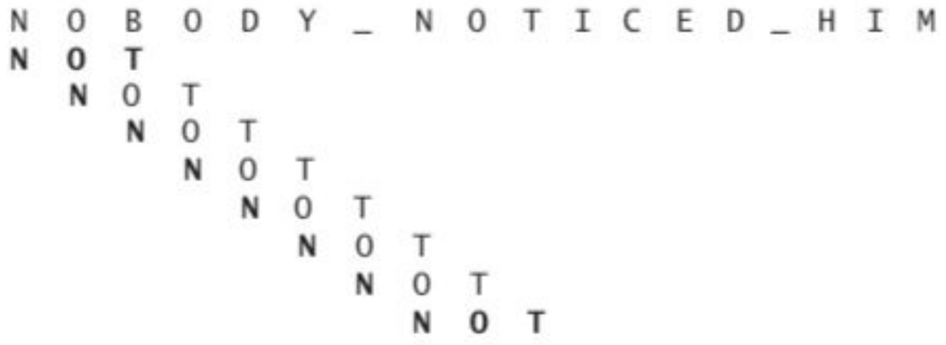


FIGURE 3.3 Example of brute-force string matching. The pattern's characters that are compared with their text counterparts are in bold type.

Time complexity

- Worst case: O(nm)
- Average case (random text): O(n)
- Is it possible to obtain O(n) for any input?
  - Knutt-Morris-Pratt (1977): deterministic
  - Karp-Rabin (1981): randomized

Reverse naïve algorithm

- Why not search from the end of *P*?
  - Boyer and Moore

```
Reverse-Naive-Search(T,P)
for s ← 0 to n – m
    j ← m – 1   // start from the end
    // check if T[s..s+m–1] = P[0..m–1]
    while T[s+j] = P[j] do
        j ← j - 1
        if j < 0 return s
return -1
```

- Running time is exactly the same as of the naïve algorithm…

Occurrence heuristic

- Boyer and Moore added two heuristics to reverse naïve, to get an $O(n+m)$ algorithm, but it's complex
- Horspool suggested just to use the modified *occurrence* heuristic:
  - *After a mismatch, align* $T[s+m-1]$ *with the rightmost occurrence of that letter in the pattern* $P[0..m-2]$
  - Examples:
    - *T*= "**`detective date`**" and *P*= "**`date`**"
    - *T*= "**`tea kettle`**" and *P*= "**`kettle`**"

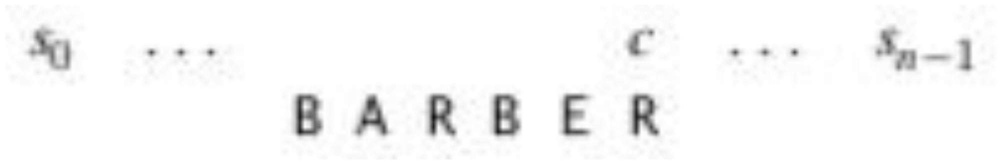
Input-enhancement Algorithms

- Input enhancement: preprocess the pattern to get some information about it, store this information in a table, and use that information during an actual search for the pattern in a given text
- The best-known algorithms:
  - Knutt-Morris-Pratt algorithm: compare characters of a pattern with their counterparts in a text from left to right
  - Boyer-Moore algorithm: compare characters of a pattern with their counterparts in a text from right to left
- Before we proceed with Boyer-Moore, we start with a simplified version of Horspool

Horspool’s Algorithm

(taken from Levitin, page 259)

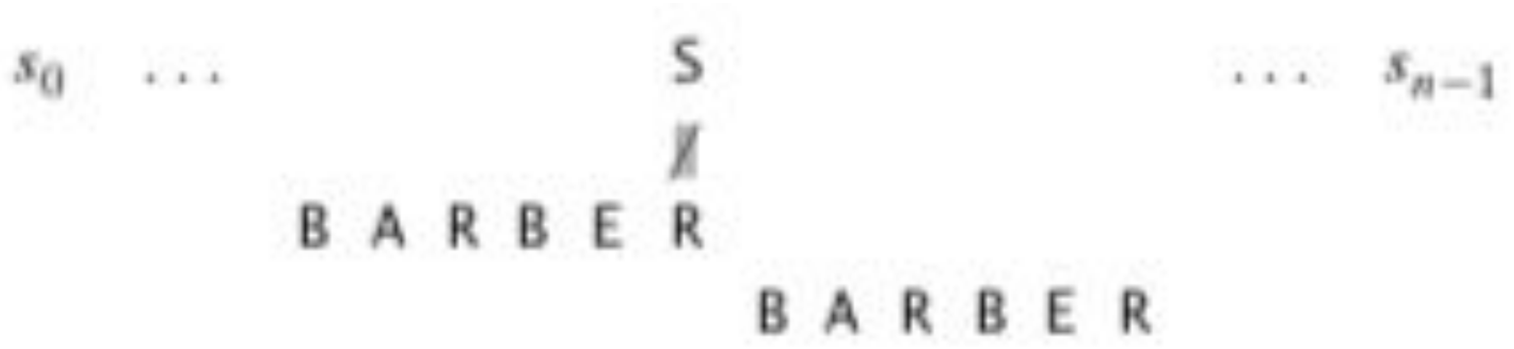
- When comparing a pattern against a position in the text, if a mismatch occurs, we need to shift the pattern to the right
- The shift would be as large as possible without risking the possibility of missing a matching substring in the text
- Idea: determine the size of the shift by looking at the character *c* of the text that is aligned against the last character of the pattern



  - Four cases:

Case 1

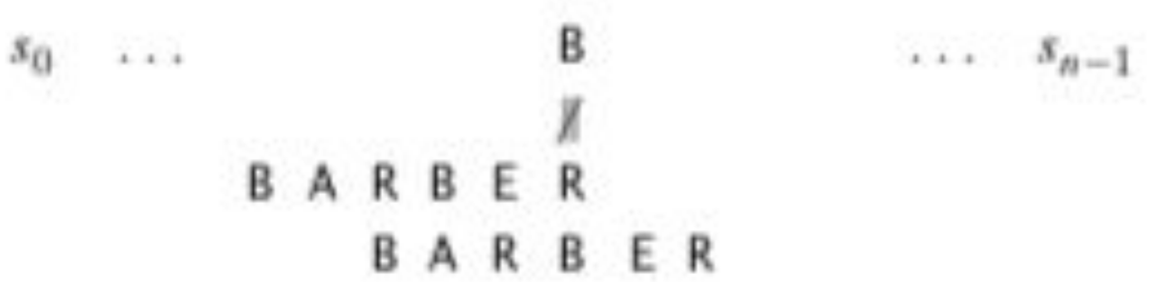
- Case 1: There are no *c*’s in the pattern;
  for example if *c* is S



  – Then we can safely shift the pattern by its entire length

Case 2

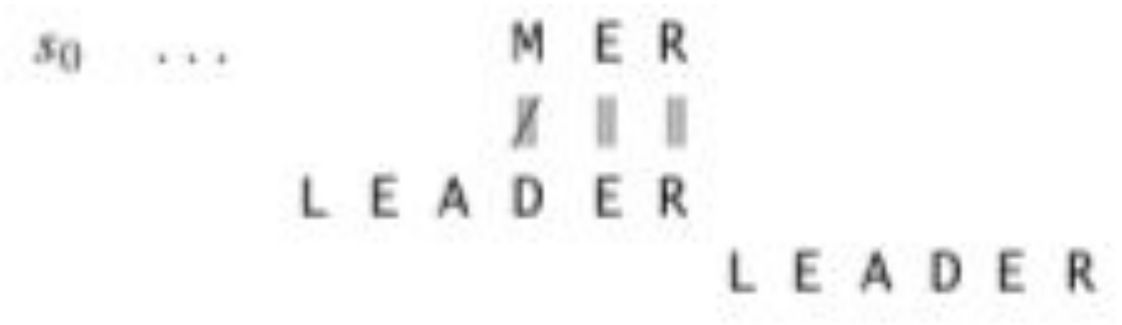
- Case 2: If there are occurrences of character *c* in the pattern but it is not the last one there;
  for example if *c* is letter B



  - Then the shift should align the rightmost occurrence of *c* in the pattern with the *c* in the text

Case 3

- Case 3: If *c* happens to be the last character in the pattern but there are no *c*'s among its other m − 1 characters; for example if *c* is letter R

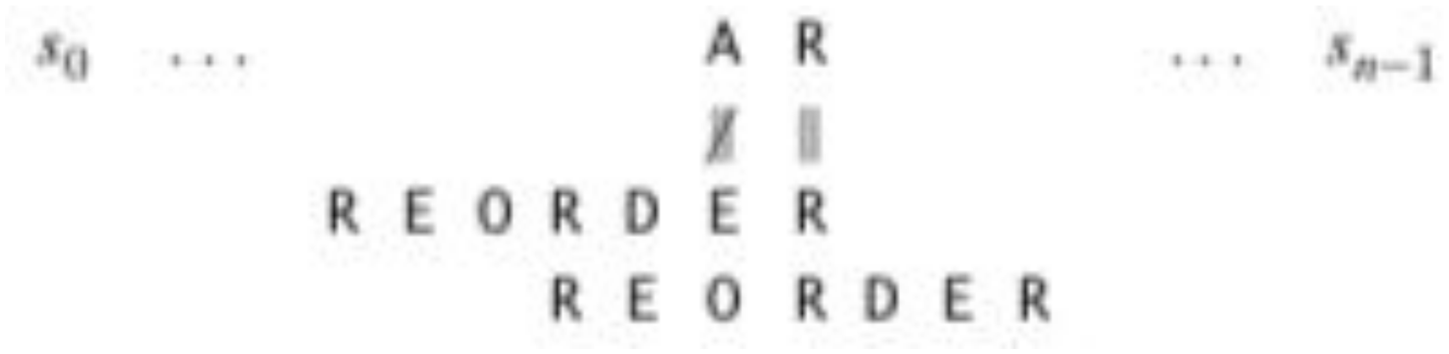


s_0 . . . M E R . . . s_{n-1}
$\not\parallel$ $\parallel$ $\parallel$
L E A D E R
L E A D E R

  - Similar to Case 1: the pattern should be shifted by the entire pattern's length *m*

Case 4

- Case 4: If *c* happens to be the last character in the pattern and there are other *c*'s among its first m − 1 characters; for example if *c* is letter R



  – Similar to Case 2: the rightmost occurrence of *c* among the first m − 1 characters in the pattern should be aligned with the text's *c*

What we need to do

- We can precompute shift sizes and store them in a table called *shift table*
- The table will be indexed by all possible characters that can be encountered in a text, including, for natural language texts, the space, punctuation symbols, and other special characters.
- The table's entries will indicate the shift sizes computed by the formula

$$t(c) = \begin{cases} \text{the pattern's length } m & \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern} \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters of the pattern to its last character} & \text{otherwise} \end{cases}$$

Example of shift table

- Consider searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores).
- The shift table, as we mentioned, is filled as follows:

Character *c*	A	B	C	D	E	F	…	R	…	Z	_
Shift *t(c)*	4	2	6	6	1	6	6	3	6	6	6

Algorithm to construct the shift table

- Initialize all the entries to the pattern's length m and scan the pattern left to right repeating the following step m − 1 times: for the j th character of the pattern ($0 \le j \le m - 2$), overwrite its entry in the table with $m - 1 - j$, which is the character's distance to the last character of the pattern.
- Note that since the algorithm scans the pattern from left to right, the last overwrite will happen for the character's rightmost occurrence - exactly as we would like it to be.

```
ALGORITHM ShiftTable(P [0..m − 1] )
//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern P [0..m − 1] and an alphabet of possible characters
//Output: Table[0..size − 1] indexed by the alphabet's characters and
// filled with shift sizes computed by formula (7.1)
1: for i ← 0 to size − 1 do
2:   Table[i] ← m
3: for j ← 0 to m − 2 do
4:   Table[P [j ]] ← m − 1 − j
5: return Table
```

Example

- For the pattern BARBER, all the table's entries will be equal to 6, except for the entries for E, B, R, and A, which will be 1, 2, 3, and 4, respectively.

Horspool’s Algorithm: Steps

Step 1 For a given pattern of length m and the alphabet used in both the pattern and text, construct the shift table as described above.

Step 2 Align the pattern against the beginning of the text.

Step 3 Repeat the following until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and text until either all m characters are matched (then stop) or a mismatching pair is encountered. In the latter case, retrieve the entry t (c) from the c’s column of the shift table where c is the text’s character currently aligned against the last character of the pattern, and shift the pattern by t(c) characters to the right along the text.

Here is pseudocode of Horspool’s algorithm.

Horspool’s Algorithm: pseudocode

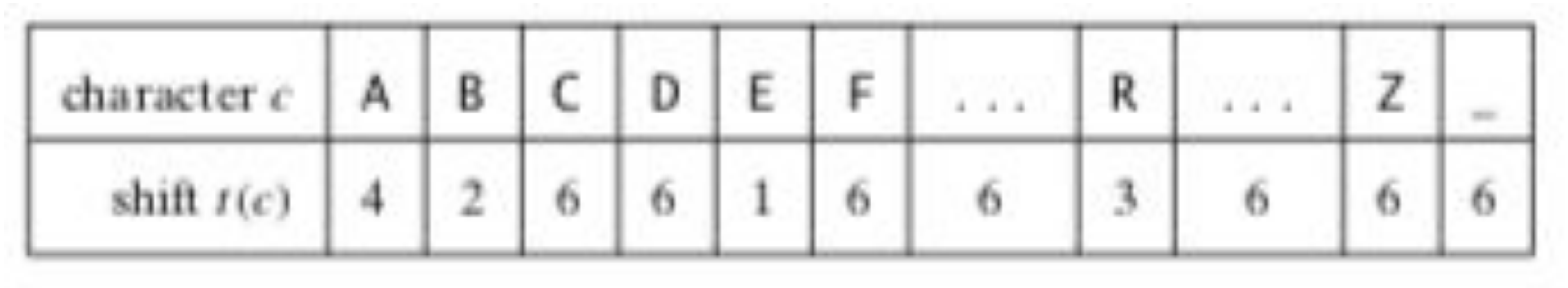
```
HorspoolMatching(P [0..m − 1], T [0..n − 1] )
//Implements Horspool’s algorithm for string matching
//Input: Pattern P [0..m − 1] and text T [0..n − 1]
//Output: The index of the left end of the first matching substring
// or −1 if there are no matches
ShiftTable(P [0..m − 1] )
1:   i = m-1
2:   while i <= n-1 do
3:       k = 0
4:       while k <= m-1 and P[m-1-k] == T[i-k] do
5:           k = k+1
6:           if k == m
7:               return i-m+1
8:       i = i + Table[T[i]]
9:   return -1
```

Time Complexity of Horspool’s algorithm

- The worst-case efficiency is in O(nm)
- For random texts, it is in $\Theta(n)$
- Although in the same efficiency class, Horspool’s algorithm is obviously faster on average than the brute-force algorithm.

Example

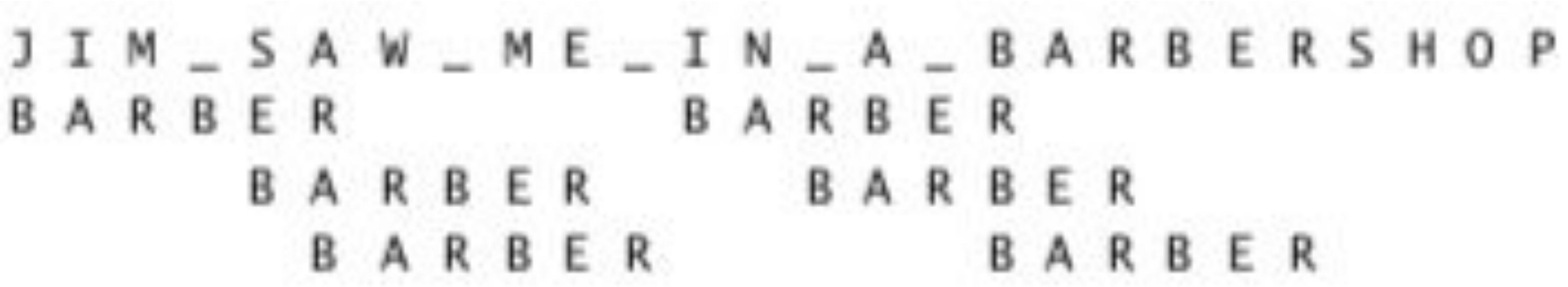
- Consider searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores).
- The shift table, as we mentioned, is filled as follows:



character c	A	B	C	D	E	F	...	R	...	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

Contd.

- The actual search in a particular text proceeds as follows:

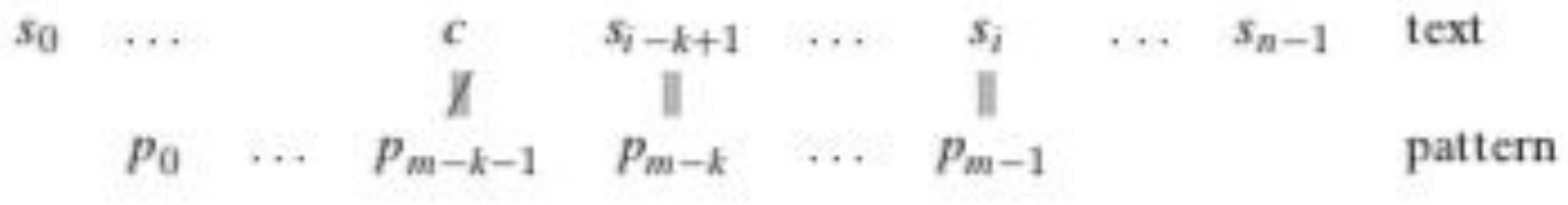


```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R           B A R B E R
        B A R B E R       B A R B E R
          B A R B E R           B A R B E R
```

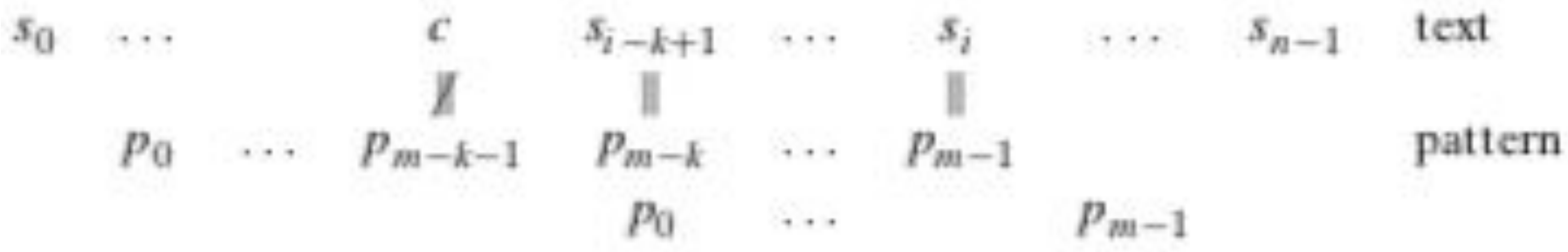
Boyer-Moore Algorithm

- If the first comparison of the rightmost character in the pattern with the corresponding character c in the text fails, the algorithm does exactly the same thing as Horspool’s algorithm, namely, it shifts the pattern to the right by the number of characters retrieved from the shift table
- The two algorithms act differently, however, after some positive number k (0 < k < m) of the pattern’s characters are matched successfully before a mismatch is encountered:

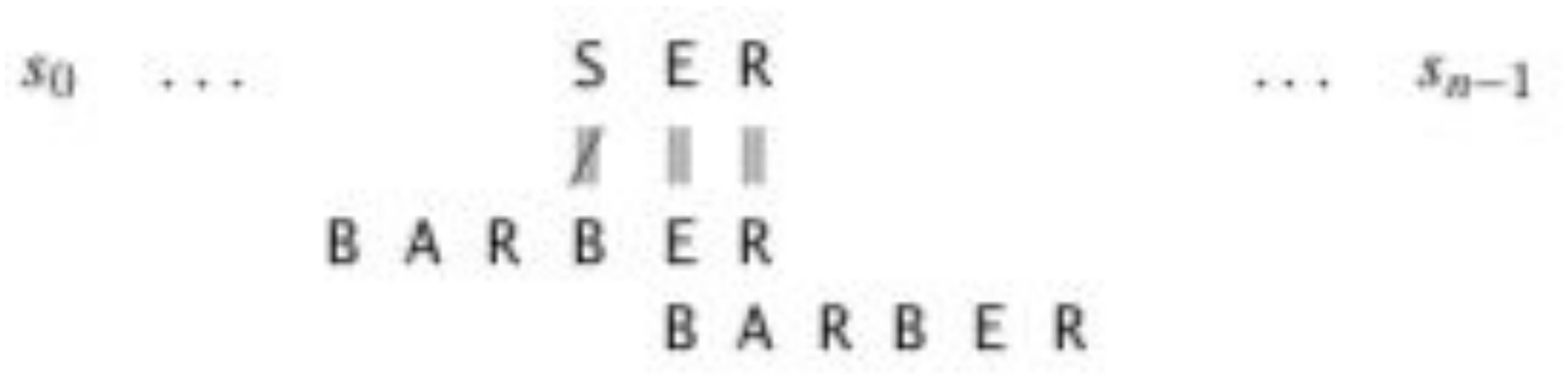
$$\begin{array}{cccccccccc} s_0 & \dots & & c & s_{i-k+1} & \dots & s_i & \dots & s_{n-1} & \text{text} \\ & & & \nparallel & \parallel & & \parallel & & & \\ & p_0 & \dots & p_{m-k-1} & p_{m-k} & \dots & p_{m-1} & & & \text{pattern} \end{array}$$



- Determines the shift size by considering two quantities
- Bad-symbol shift: If *c* is not in the pattern, we shift the pattern to just pass this *c* in the text. Conveniently, the size of this shift can be computed by the formula $t_1(c) - k$ where $t_1(c)$ is the entry in the shift table



- For example, if we search for the pattern BARBER in some text and match the last two characters before failing on letter S in the text, we can shift the pattern by $t_1(S) - 2 = 6 - 2 = 4$ positions:



- Good-suffix shift: a successful match of the last $k > 0$ characters of the pattern; we refer to the ending portion of the pattern as its *suffix* of size k and denote it suff (k). We now apply the reasoning that guided us in filling the bad-symbol shift table, which was based on a single alphabet character c, to the pattern's suffixes of sizes $1, . . . , m - 1$ to fill in the good-suffix shift table.
  – If there is another occurrence of suff(k) not preceded by the same character as in its rightmost occurrence, we can shift the pattern by the distance d_2 between such a second rightmost occurrence (not preceded by the same character as in the rightmost occurrence) of suff (k) and its rightmost occurrence.

- If there is no other occurrence of suff (k) not preceded by the same character as in its rightmost occurrence, we need to find the longest prefix of size $l < k$ that matches the suffix of the same size l. If such a prefix exists, the shift size d_2 is computed as the distance between this prefix and the corresponding suffix; otherwise, d_2 is set to the pattern’s length m.

Boyer-Moore Algorithm: Steps

Step 1 For a given pattern and the alphabet used in both the pattern and the text, construct the bad-symbol shift table as described earlier.

Step 2 Using the pattern, construct the good-suffix shift table as described earlier.

Step 3 Align the pattern against the beginning of the text.

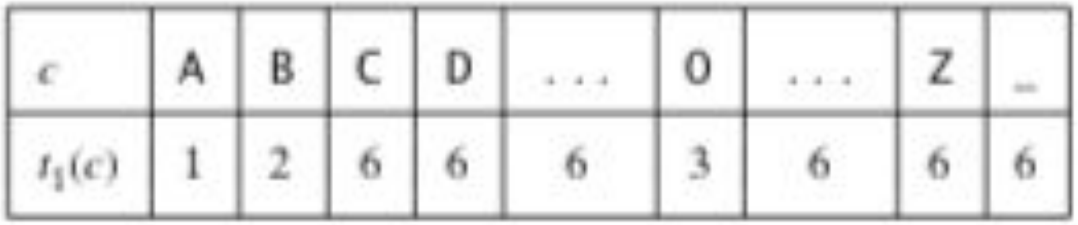
Step 4 Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and the text until either all m character pairs are matched (then stop) or a mismatching pair is encountered after $k \geq 0$ character pairs are matched successfully. In the latter case, retrieve the entry $t_1(c)$ from the c's column of the bad-symbol table where c is the text's mismatched character. If $k > 0$, also retrieve the corresponding d_2 entry from the good-suffix table. Shift the pattern to the right by the number of positions computed by the formula

$$d = \begin{cases} d_1 & \text{if } k = 0, \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases}$$

where $d_1 = \max\{t_1(c) - k, 1\}$.

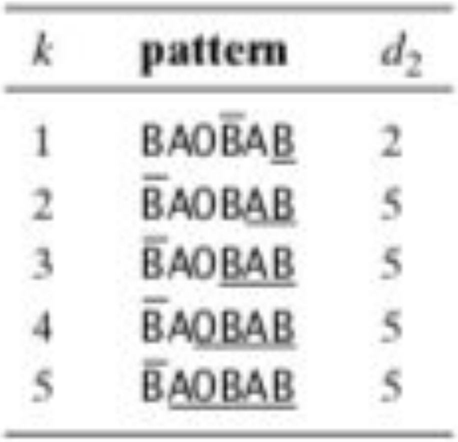
Example

- Searching for the pattern BAOBAB in a text made of English letters and spaces.
- The bad-symbol table looks as follows:



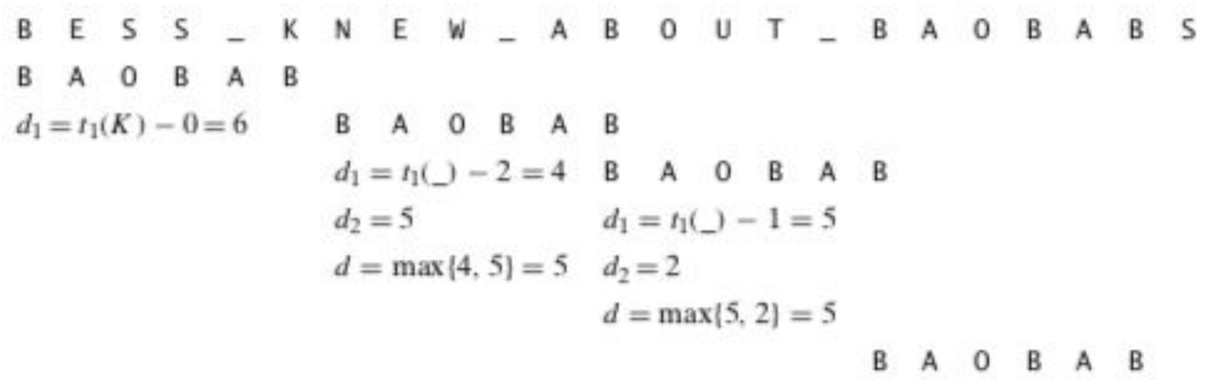
c	A	B	C	D	...	O	...	Z	_
$t_1(c)$	1	2	6	6	6	3	6	6	6

- The good-suffix table:



k	**pattern**	d_2
1	$\text{BAO}\overline{\text{B}}\text{A}\underline{\text{B}}$	2
2	$\overline{\text{B}}\text{AOB}\underline{\text{AB}}$	5
3	$\overline{\text{B}}\text{AO}\underline{\text{BAB}}$	5
4	$\overline{\text{B}}\text{A}\underline{\text{OBAB}}$	5
5	$\overline{\text{B}}\underline{\text{AOBAB}}$	5

- The actual search for this pattern in the text

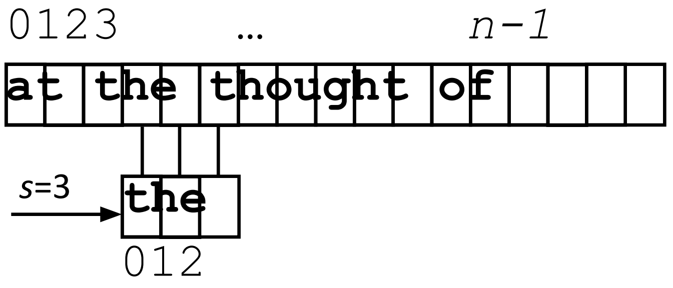


```
B E S S _ K N E W _ A B O U T _ B A O B A B S
B A O B A B
d1 = t1(K) − 0 = 6   B A O B A B
                     d1 = t1(_) − 2 = 4   B A O B A B
                     d2 = 5               d1 = t1(_) − 1 = 5
                     d = max{4, 5} = 5    d2 = 2
                                          d = max{5, 2} = 5
                                                            B A O B A B
```

- After the last B of the pattern fails to match its counterpart K in the text, the algorithm retrieves t_1(K) = 6 from the bad-symbol table and shifts the pattern by d_1 = max{t1(K) − 0, 1} = 6 positions to the right.
- The new try successfully matches two pairs of characters.
- After the failure of the third comparison on the space character in the text, the algorithm retrieves t1( ) = 6 from the bad-symbol table and d2 = 5 from the good-suffix table to shift the pattern by max{d1, d2} = max{6 − 2, 5} = 5.
- Note that on this iteration it is the good-suffix rule that leads to a farther shift of the pattern.
- The next try successfully matches just one pair of B’s.
- After the failure of the next comparison on the space character in the text, the algorithm retrieves t_1( ) = 6 from the bad-symbol table and d_2 = 2 from the good-suffix table to shift the pattern by max{d1,d2} = max{6 − 1, 2} = 5.
- Note that on this iteration it is the bad-symbol rule that leads to a farther shift of the pattern. The next try finds a matching substring in the text after successfully matching all six characters of the pattern with their counterparts in the text.

Text-Search Problem

- Input:
  - *Text T* = “**`at the thought of`**”
    - *n* = length(*T*) = 17
  - *Pattern P* = “**`the`**”
    - *m* = length(*P*) = 3
- Output:
  - *Shift s* – the smallest integer (0 ≤ *s* ≤ *n* – *m*) such that *T*[*s* .. *s*+*m*–1] = *P*[0 .. *m*–1]. Returns –1, if no such *s* exists



Naïve Text Search

- Idea: Brute force
  - Check all values of s from 0 to *n* – *m*

```
Naive-Search(T,P)
for s ← 0 to n – m
    j ← 0
    // check if T[s..s+m–1] = P[0..m–1]
    while T[s+j] = P[j] do
        j ← j + 1
        if j = m return s
return –1
```

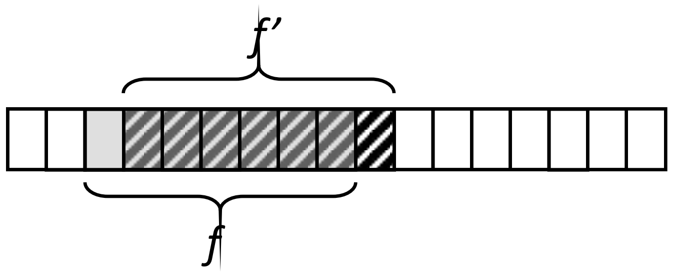
- Let *T* = "**`at the thought of`**" and *P* = "**`though`**"
  - What is the number of character comparisons?

Analysis of Naïve Text Search

- Worst-case:
  - Outer loop: $n - m$
  - Inner loop: m
  - Total $(n-m)m = O(nm)$
  - What is the input the gives this worst-case behaviuor?
- Best-case: n-m
  - When?
- Completely random text and pattern:
  - $O(n-m)$

Fingerprint idea

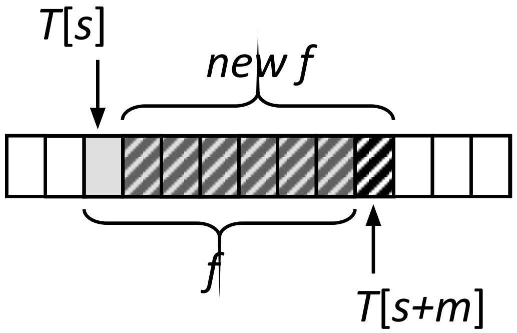
- Assume:
  - We can compute a ***fingerprint*** $f(P)$ of P in O(m) time.
  - If $f(P) \neq f(T[s\ ..\ s+m-1])$, then $P \neq T[s\ ..\ s+m-1]$
  - We can compare fingerprints in O(1)
  - We can compute $f' = f(T[s+1 ..\ s+m])$ from $f(T[s\ ..\ s+m-1])$, in O(1)



Algorithm with Fingerprints

- Let the alphabet Σ={**0,1,2,3,4,5,6,7,8,9**}
- Let fingerprint to be just a decimal number, i.e., *f*("**1045**") = $1*10^3 + 0*10^2 + 4*10^1 + 5 = 1045$

```
Fingerprint-Search(T,P)
01 fp ← compute f(P)
02 f ← compute f(T[0..m-1])
03 for s ← 0 to n – m do
04     if fp = f return s
05     f ← (f – T[s]*10^(m-1))*10 + T[s+m]
06 return -1
```



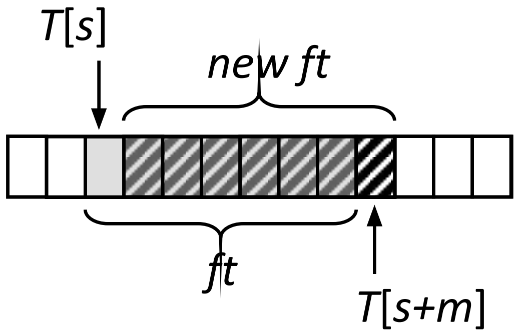
- Running time $2O(m) + O(n-m) = O(n)$!
- Where is the catch?

Using a Hash Function

- Problem:
  - we can not assume we can do arithmetics with *m*-digits-long numbers in O(1) time
- Solution: Use a hash function $h = f \bmod q$
  - For example, if $q = 7$, h(“**52**”) = 52 mod 7 = 3
  - $h(S_1) \neq h(S_2) \Rightarrow S_1 \neq S_2$
  - But $h(S_1) = h(S_2)$ *does not* imply $S_1 = S_2$!
    - For example, if $q = 7$, h(“**73**”) = 3, but “**73**” ≠ “**52**”
- Basic “mod *q*” arithmetics:
  - $(a+b) \bmod q = (a \bmod q + b \bmod q) \bmod q$
  - $(a*b) \bmod q = (a \bmod q)*(b \bmod q) \bmod q$

Preprocessing and Stepping

- Preprocessing:
  - *fp* = *P*[*m*-1] + 10*(*P*[*m*-2] + 10*(*P*[*m*-3]+ … … + 10*(*P*[1] + 10**P*[0])…)) mod *q*
  - In the same way compute *ft* from *T*[0..*m*-1]
  - Example: *P* = "**`2531`**", *q* = 7, what is *fp*?
- Stepping:
  - *ft* = (*ft* – *T*[*s*]*10^{m-1} mod *q*)*10 + *T*[*s*+*m*]) mod *q*
  - 10^{m-1} mod *q* can be computed once in the preprocessing
  - Example: Let *T*[…] = "**`5319`**", *q* = 7, what is the corresponding *ft*?



Rabin-Karp algorithm

(https://users.cs.duke.edu/~reif/courses/alglectures/indyk.lectures/lecture_22.pdf)

- Uses hashing:
  - Hash all substrings T[0..m-1], T[1..m], …T[n-m..n-1]
  - Hash the pattern P[0..m-1]
  - Report the strings that hash to the same value as P
- Challenge: how to hash n-m substrings, each of length m, in O(n) time?

Converting strings to decimal values

(textbook, pg 990-991)

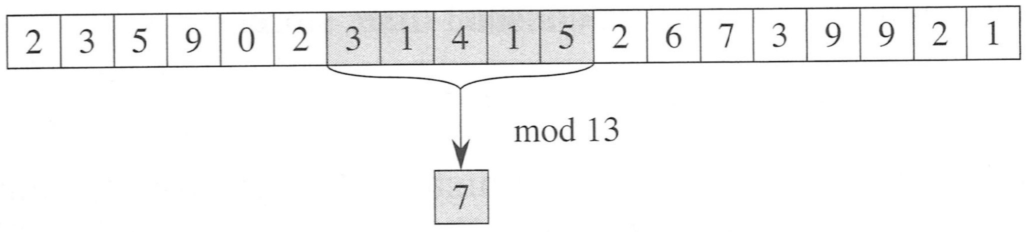
- For the pattern P[0..m-1] we need to produce a decimal value
- Consider Σ ={0,1,…9} and consider each character to be a digital digit; e.g. “31415” will have the decimal value of 31,415
  - Otherwise we use polynomial coding with a =$|\Sigma|$ to convert form any string to a decimal value
- Let p be its decimal value of the pattern P[0..m-1] and let t_s be the decimal value of the substrings T[s..(m-1)+s]
- The decimal value p can be computed in O(m) time and all $t_0, t_1, \ldots t_{n-m}$ can be computed in O(n-m+1) time; why?
  - Because t_{s+1} can be computed from t_s in constant time, since $t_{s+1}=10(t_s-10^{m-1}T[s])+T[s+m]$
    and computing 10^{m-1} can be done in O(log m)

- We might get large decimal values, so we apply hashing by taking modulo some q, where q is a prime such that 10q fits within one computer word
  - In general, we choose q to be a prime such that $q \cdot |\Sigma|$ fits in a computer word
- We need to compare p mod q with each $t_0, t_1, \ldots t_{n-m}$ where $t_{s+1} = (10(t_s - (10^{m-1} \bmod q)T[s]) + T[s+m]) \bmod q$
- Therefore we can find all occurrences of the pattern P[0..m-1] in the text T[0..n-1] with O(m) preprocessing time and O(n-m+1) matching time

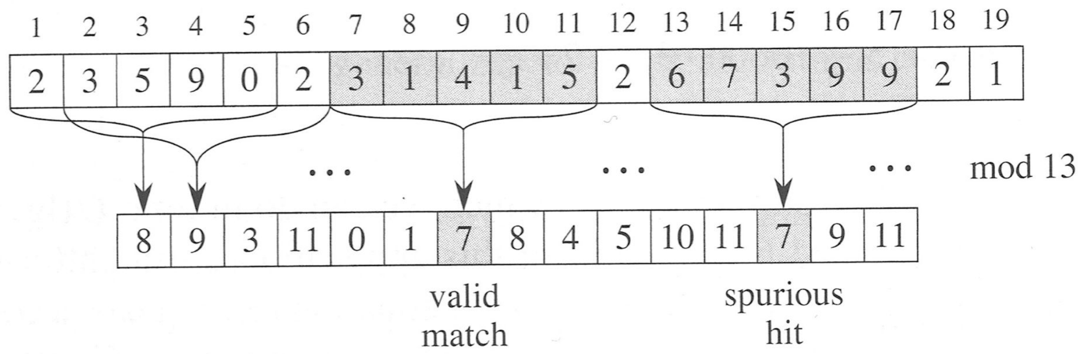
Rabin-Karp Algorithm

```
Rabin-Karp-Searcher(T,P)
q ← a prime larger than m
h ← $10^{m-1}$ mod q  // run a loop multiplying by 10 mod q
fp ← 0; ft ← 0
for i ← 0 to m-1  // preprocessing
   fp ← (10*fp + P[i]) mod q
   ft ← (10*ft + T[i]) mod q
for s ← 0 to n – m  // matching
   if fp = ft then   // run a loop to compare strings
      if P[0..m-1] = T[s..s+m-1] return s
   ft ← ( (ft – T[s]*h)*10 + T[s+m]) mod q
return -1
```

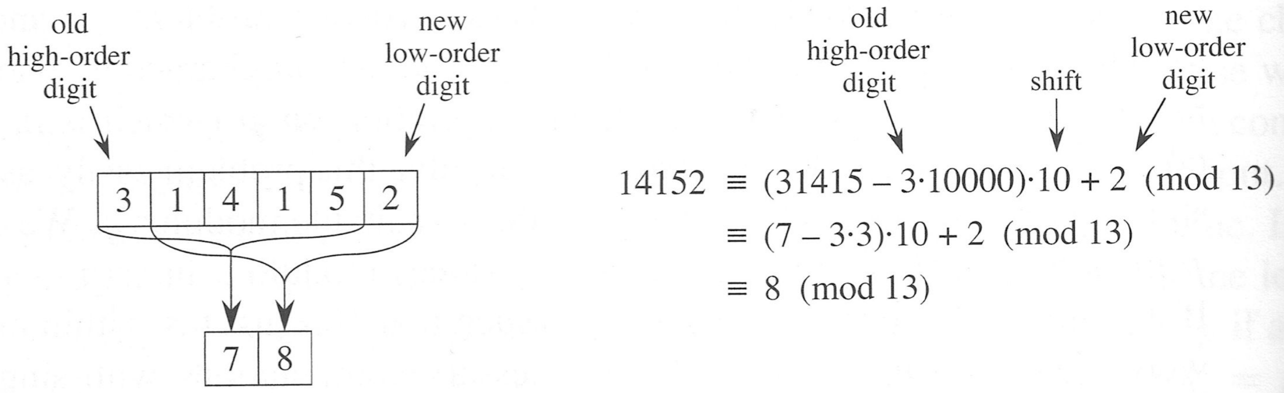
- How many character comparisons are done if *T* = "**2359023141526739921**" and *P* = "31415"? (see Fig. 32.5 on page 992 of the textbook)



(a)



(b)



$$
\begin{aligned}
14152 &\equiv (31415 - 3\cdot 10000)\cdot 10 + 2 \pmod{13}\\
&\equiv (7 - 3\cdot 3)\cdot 10 + 2 \pmod{13}\\
&\equiv 8 \pmod{13}
\end{aligned}
$$

(c)

Analysis

- If *q* is a prime, the hash function distributes *m*-digit strings evenly among the *q* values
  - Thus, only every *q*-th value of shift *s* will result in matching fingerprints (which will require comparing strings with *O*(*m*) comparisons)
- Expected running time (if *q* > *m*):
  - Preprocessing: *O*(*m*)
  - Outer loop: *O*(*n*-*m*)
  - All inner loops: $\frac{n-m}{q}m = O(n-m)$
  - Total time: *O*(*n*-*m*)
- Worst-case running time: *O*(*nm*)

Rabin-Karp in Practice

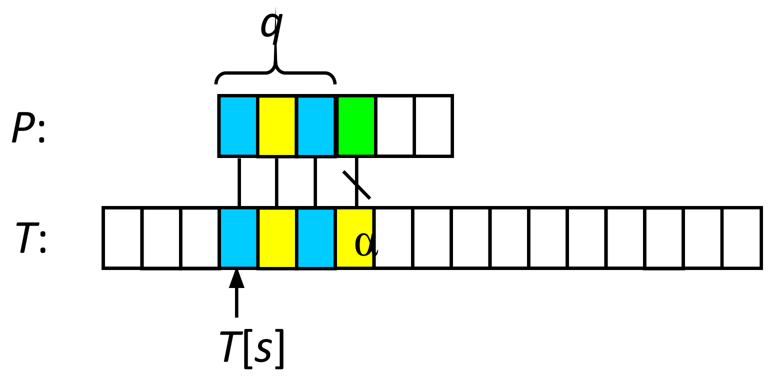
- If the alphabet has *d* characters, interpret characters as radix-*d* digits (replace 10 with *d* in the algorithm).
- Choosing prime $q > m$ can be done with randomized algorithms in O(*m*), or *q* can be fixed to be the largest prime so that 10**q* fits in a computer word.
- Rabin-Karp is simple and can be easily extended to two-dimensional pattern matching.

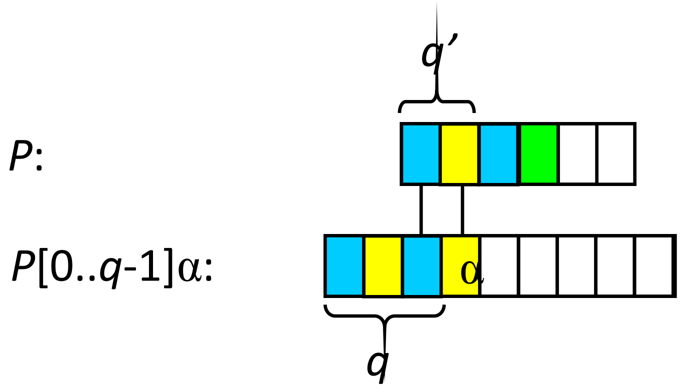
Searching in *n* comparisons

- The goal: each character of the text is compared only once!
- Problem with the naïve algorithm:
  - Forgets what was learned from a partial match!
  - Examples:
    - *T* = "**`Tweedledee and Tweedledum`**" and *P* = "**`Tweedledum`**"
    - *T* = "**`pappar`**" and *P* = "**`pappappappar`**"

General situation

- State of the algorithm:
  - Checking shift *s*,
  - *q* characters of *P* are matched,
  - we see a non-matching character α in *T*.
- Need to find:
  - Largest prefix "*P*-" such that it is a suffix of *P*[0..*q*-1]α:
    - New $q' = \max\{k \leq q \mid P[0..k-1] = P[q-k+1..q-1]\alpha\}$



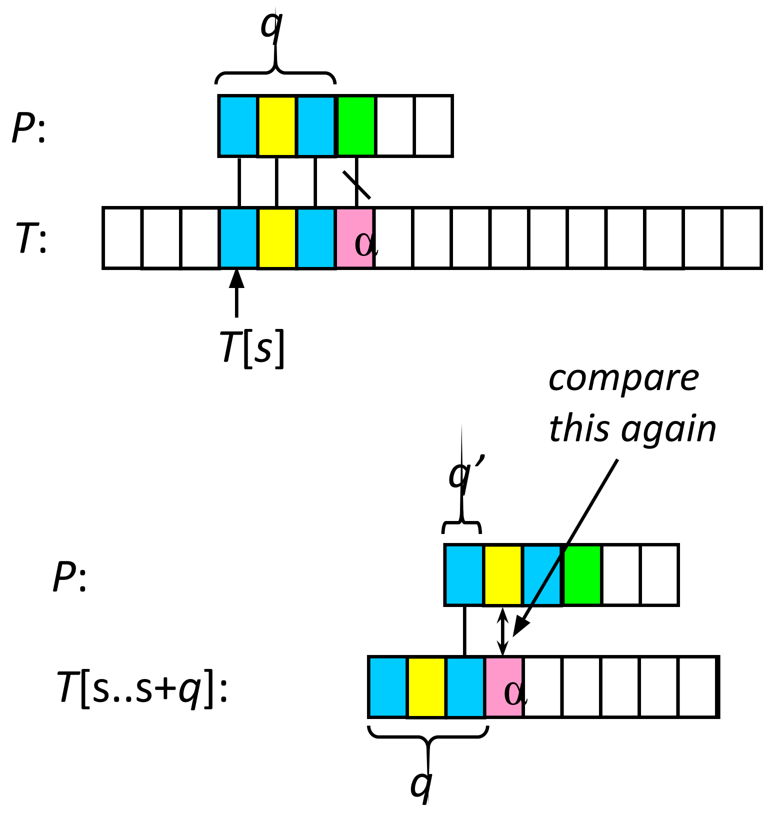


Finite automaton search

- Algorithm:
  - Preprocess:
    - For each *q* ($0 \le q \le m-1$) and each $\alpha \in \Sigma$ pre-compute a new value of *q*. Let's call it $\sigma(q,\alpha)$
    - Fills a table of a size $m|\Sigma|$
  - Run through the text
    - Whenever a mismatch is found ($P[q] \neq T[s+q]$):
    - Set $s = s + q - \sigma(q,\alpha) + 1$ and $q = \sigma(q,\alpha)$
- Analysis:
  - ☺️ Matching phase in $O(n)$
  - ☹️ Too much memory: $O(m|\Sigma|)$, too much preprocessing: at best $O(m|\Sigma|)$.

Prefix function

- Idea: forget unmatched character (α)!
- State of the algorithm:
  - Checking shift *s,*
  - *q* characters of *P* are matched*,*
  - we see a non-matching character α in *T.*
- Need to find:
  - Largest prefix "*P*-" such that it is a suffix of *P*[0..*q*-1]:



- New $q' = \pi[q] = \max\{k < q \mid P[0..k-1] = P[q-k..q-1]\}$

Prefix table

- We can pre-compute a *prefix table* of size m to store values of $\pi[q]$ $(0 \le q < m)$

P		**`p`**	**`a`**	**`p`**	**`p`**	**`a`**	**`r`**
q	0	1	2	3	4	5	6
$\pi[q]$	0	0	0	1	1	2	0

- Compute a prefix table for: P = "**`dadadu`**"

Knuth-Morris-Pratt Algorithm

```
KMP-Search(T,P)
$\pi \leftarrow$ Compute-Prefix-Table(P)
$q \leftarrow 0$      // number of characters matched
for $i \leftarrow 0$ to $n-1$  // scan the text from left to right
   while $q > 0$ and $P[q] \neq T[i]$ do
      $q \leftarrow \pi[q]$
   if $P[q] = T[i]$ then $q \leftarrow q + 1$
   if $q = m$ then return $i - m + 1$
return $-1$
```

- **Compute-Prefix-Table** is the essentially the same KMP search algorithm performed on P.

Analysis of KMP

- Worst-case running time: $O(n+m)$
  - Main algorithm: $O(n)$
  - **`Compute-Prefix-Table`**: $O(m)$
- Space usage: $O(m)$