Special course in Computer Science: Advanced Text Algorithms

Lecture 3: Pattern Matching Algorithms

Eugen Czeizler
Department of IT, Abo Akademi
http://combio.abo.fi/teaching/textalg/

(slides originally by I. Petre, E. Czeizler, V. Rogojin)

Pattern Matching Problem

- The pattern matching problem: Find occurrences of a given pattern within a text.
  - Example: Find occurrences of the word “example” in the text: “Here is a simple example.”
- The input for the pattern matching algorithm:
  - the pattern pat of length m
  - the text text of length n
- The output of the algorithm:
  - A boolean value: 1 if pat occurs in the text or 0 otherwise.
  - A list of positions where pat occurs in the text.

Pattern matching algorithms

- The scheme for a brute force algorithm is the following:

$$\text{for } i := 0 \text{ to } n - m \text{ do}$$
$$\text{check if } pat = text[i+1 \,..\, i+m]$$

- The actual implementation of this algorithm depends on how we implement the function "check", e.g., scanning the text from the left, or from the right, or other method.

Pattern matching algorithms

In Lecture 2 we saw already two algorithms for pattern matching:

- A brute-force algorithm:
  - quadratic time complexity
  - unsuitable in practice, when we work with huge texts
- Knuth-Morris-Pratt algorithm
  - scans the pattern from left to right;
  - pre-processing phase in $O(m)$ time complexity;
  - searching phase in $O(n)$ time complexity (independent from the alphabet size);

Boyer-Moore (BM) algorithm

- The Boyer-Moore algorithm is a particularly efficient string searching algorithm, and it has been the standard benchmark for the practical string search literature.

- A simplified version of it or the entire algorithm is often implemented in text editors for the "search" and "substitute" commands.

- The reason is that it works the fastest when the alphabet is large and the pattern is relatively long.

KMP and BM algorithms

- The Boyer-Moore algorithm is extremely fast on large alphabets (relative to the length of the pattern).

- The payoff is that it is not as fast for binary strings or for very short patterns.

- For texts over a binary (or a small) alphabet Knuth-Morris-Pratt algorithm is recommended.

- Also, Knuth-Morris-Pratt algorithm generalizes more easily than Boyer-Moore algorithm to problems such as:
  - real-time matching and
  - matching against a set of patterns.

Boyer-Moore algorithm

- The execution time of the Boyer-Moore algorithm can be sub-linear: it does not need to check every character of the text, but rather skips over some of them.

- The BM algorithm has the peculiar property that, roughly speaking, it gets faster as the pattern becomes longer.

- This is due to the fact that with each unsuccessful attempt to find a match between pat and text, it uses the information gained from that attempt to rule out as many positions as possible of the text where the strings cannot match.

Boyer-Moore algorithm

- The speed of the BM-algorithm comes from shifting the pattern pat to the right in longer steps.

- The Boyer-Moore algorithm is based on three ideas:

  - Scanning the pattern pat from right to left: pat[m]; pat[m-1];...

  - The “bad character shift rule”: avoids repeating unsuccessful comparisons against a target character (from the text)

  - The “good suffix shift rule”: aligns only matching pattern characters against target characters already successfully matched

- Either rule alone works, but they are more effective together

Right to left scan

- For any alignment of pat with text BM algorithm checks for an occurrence of pat by scanning its characters from right to left.

- Example: Take the text="This picture shows a nice view of the park." and the pattern pat="future" and let us consider the following alignment:

```
text:   This picture shows a nice view of the park.
pat:          future
```

- To check if pat occurs in text at this position BM algorithm scans pat as follows:

Right to left scan

```
text:   This picture shows a nice view of the park.
pat:            future
```

1) pat[6]=text[12]=e, 2) pat[5]=text[11]=r
3) pat[4]=text[10]=u, 4) pat[3]=text[9]=t
5) pat[2]=u≠c=text[8]

- Since we found a mismatch, pat is shifted to the right (relative to the text) and the comparisons begin again from the right end of pat

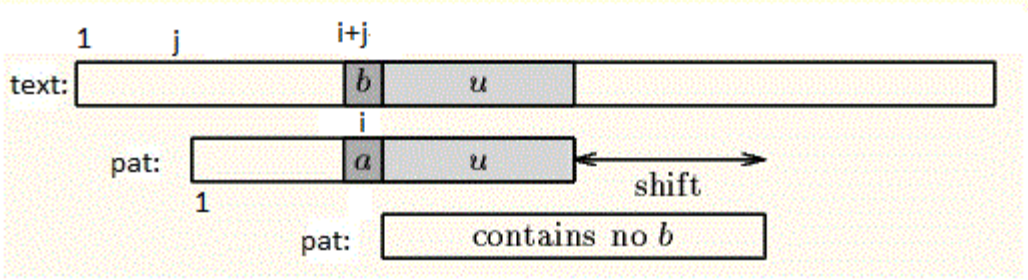
- The length of shift is given by the two rules:
  - the "bad character shift rule"
  - the "good suffix shift rule"

The "bad character shift rule"

- Assume that a mismatch occurs between the character *pat*[*i*]=*a* of the pattern and the character *text*[*i*+*j*]=*b* of the text during an attempt at position *j*.

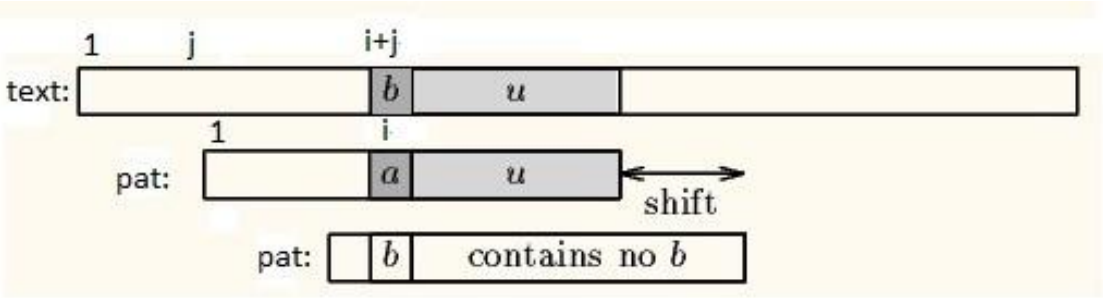
- Then, pat[*i*+*1* .. *m*]=*text*[*i*+*j*+*1* .. *j*+*m*]=u and *pat*[*i*] ≠text[*i*+*j*].

- If "b" is not contained anywhere in pat, then shift the pattern pat completely past text[i+j].



The “bad character shift rule”

- Otherwise, shift the pattern pat until the rightmost occurrence of the character “b” in pat gets aligned with text[i+j].



- A shorter shift will result in an immediate mismatch.
- This “long” shift is correct: it will not shift past any occurrence of pat in the text

Pre-processing for the "bad character shift rule"

- For all characters x in the alphabet we define the function

$$R(x) = \begin{cases} 0 & , \text{ if } x \text{ does not occur in pat} \\ \max\{i < m \mid pat[i] = x\} & , \text{ otherwise} \end{cases}$$

- It is easy to compute R(x) in O(m) time.

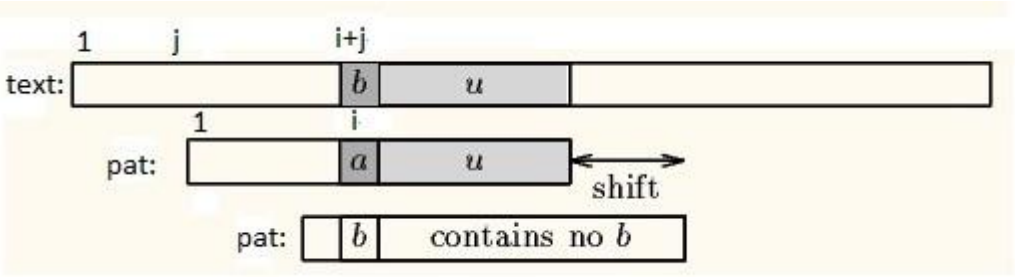
- This function is used to compute the bad character shift:

- When:
  - pat[$i+1$ .. m]=$text$[$i+j+1$ .. $j+m$]=u and
  - pat[i]≠text[i+j] and text[i+j]=b,

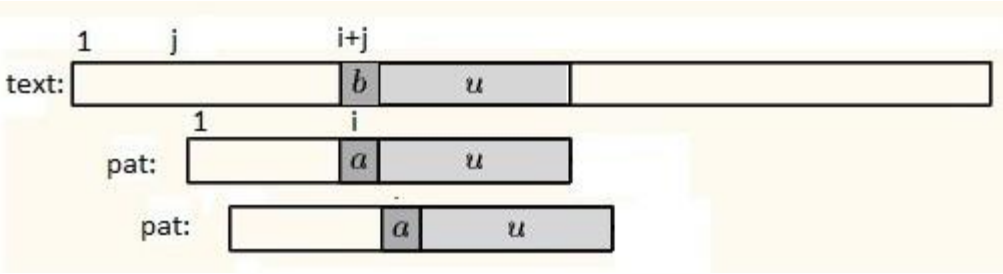
shift pat to the right by max{1, i-R(b)}.

Preprocessing for the "bad character shift rule"

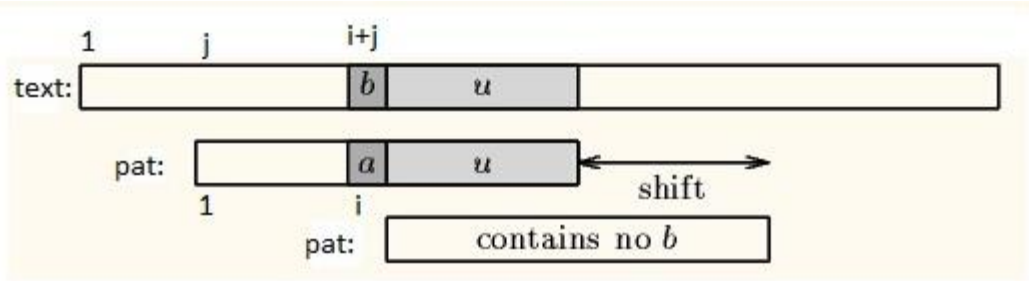
- if the right-most occurrence of b in pat[1..m-1] is at k < i, characters pat[k] and text[i+j] get aligned.



- if the right-most occurrence of b in pat[1..m-1] is at k > i, the pattern is shifted to the right by one.



- if b does not occur in pat[1..m-1], then shift = i, and the pattern is next aligned with text[i+j+1.. i+j+m].



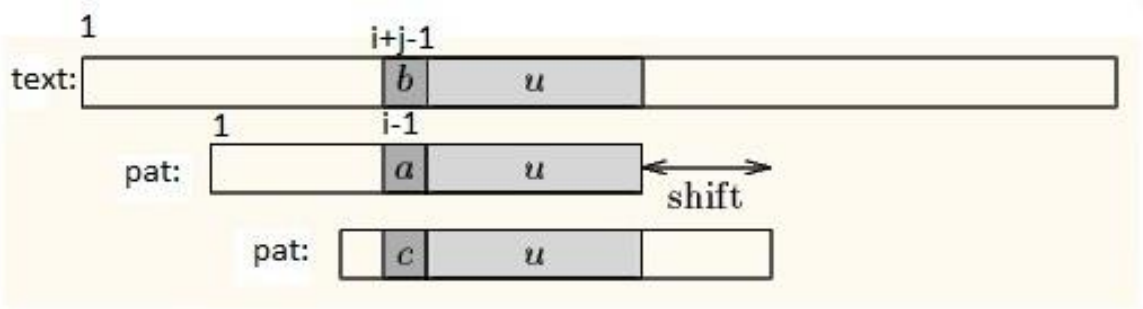
The "bad character shift rule"

- By itself, the "bad character shift rule":
  - is highly effective for large alphabets, e.g. searching in natural language texts (because mismatches are probable)
  - is less effective for small alphabets: occurrences of any character close to the end of the pattern are probable.
  - it does not lead to a linear worst-case running time.
- We consider then the so called strong good suffix rule, which is more powerful than the *(weak) suffix rule of the* original Boyer-Moore method

The "good suffix shift rule"

- Let us suppose we have pat[*i* .. *m*]=*text*[*i*+*j* .. *j*+*m*]=u and *pat*[*i*-*1*] ≠text[*i*+*j*-*1*].

- The good-suffix shift consists in aligning the segment u=*text*[*i*+*j* .. *j*+*m*] (=*pat*[*i* .. *m*]) with its rightmost occurrence in *pat (which is not a suffix)* that is preceded by a character different from *pat*[*i*-*1*]



- The original weaker version of this rule did not require that the character preceding the rightmost occurrence of u should not be pat[i-1].

The "good suffix shift rule"

- Example:

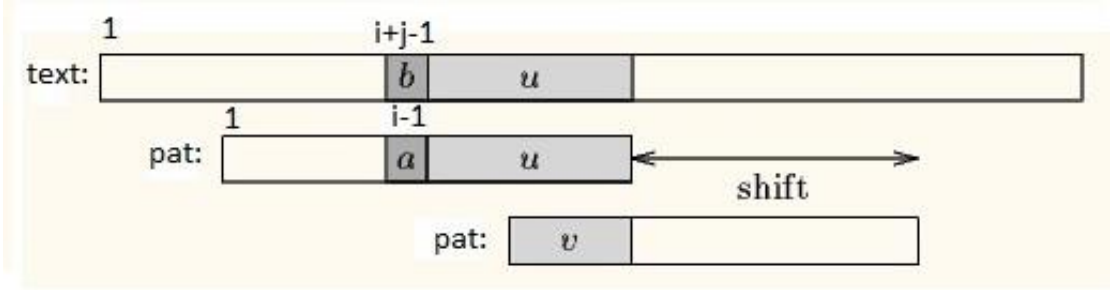
```
text: I visited Helsinki by bike and I pedaled a lot.
pat:    pedaled
pat:         pedaled
```

- We want to align text[7..9]="ted" with some factor of the pattern u="xed", with x≠l.

- In this example the factor is u=pat[1..3]="ped" so we shift pat such that pat[1..3] gets aligned with text[7..9].

The "good suffix shift rule"

- If there exists no such segment, the shift consists in aligning the longest suffix *v* of text[*i*+*j* .. *j*+*m*] with a matching prefix of *pat*



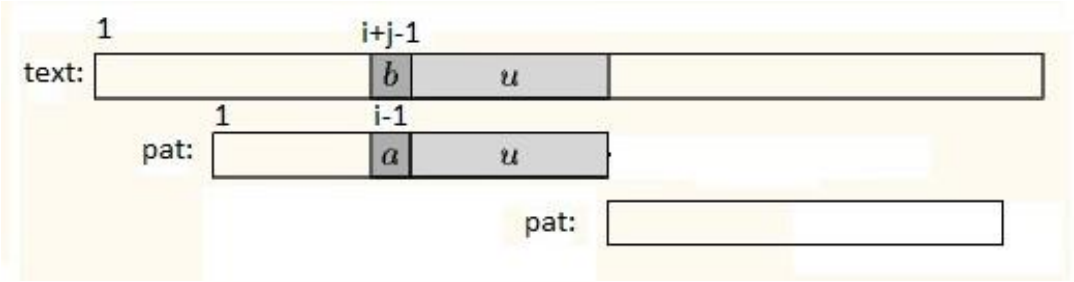

```
text: When the phone ranged he was disturbed.
pat:                  disturbed
```

- There is no other factor "ed" in pat but a potential occurrence of pat starts at text[21]=d

```
text: When the phone ranged he was disturbed.
pat:                        disturbed
```

The "good suffix shift rule"

- *If no such shift is possible, i.e.,* the longest suffix *v* of text[*i*+*j* .. *j*+*m*] which matches a prefix of *pat, is empty, then shift pat to the right by* |*pat*|=*m.*



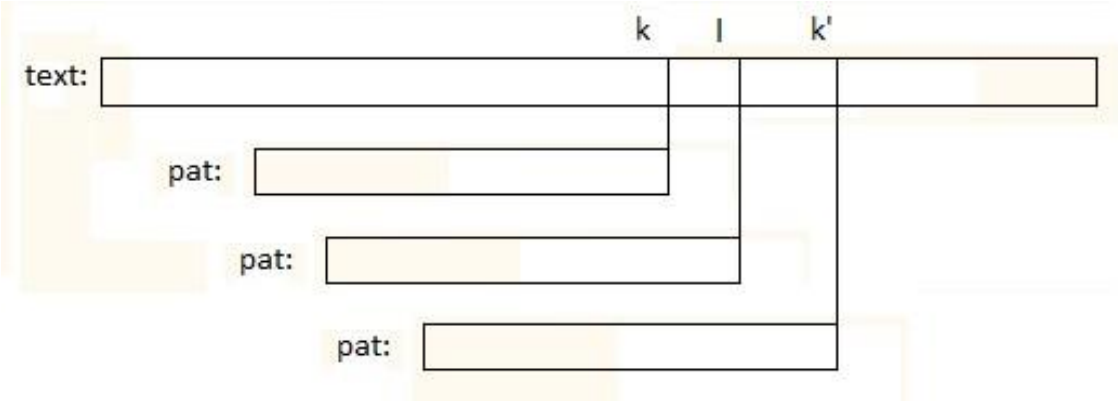

```
text: I travelled by bike, so I pedaled a lot.
pat:      pedaled
pat:            pedaled
```

The "good suffix shift rule"

- Theorem: The use of the good suffix shift rule never shifts the pattern past an occurrence in the text.

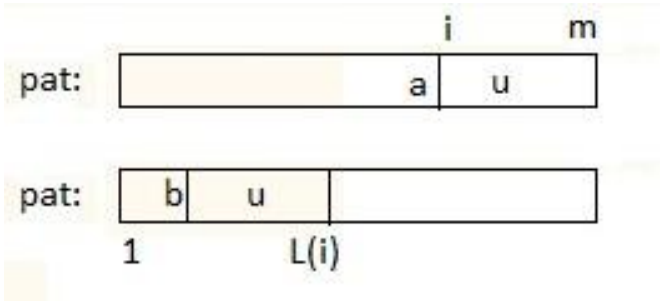
Proof:

- Suppose pat[m] is aligned with text[k] before a shift using the good suffix rule
- Suppose then that pat[m] gets aligned with text[k'] after this shift
- Suppose there is another occurrence of pat such that pat[m] is aligned with text[l] with k<l<k'
- This contradicts the selection of k':
- either the shift was not done for the rightmost occurrence of the suffix u
- or a longer prefix of pat matches a suffix of u



Pre-processing for the "good suffix shift rule"

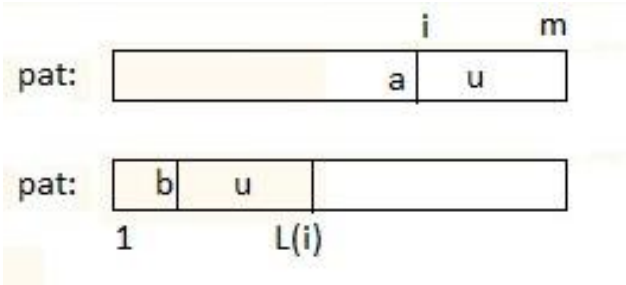
- For i = 2,.., m+1, define L(i) as follows:
  - L(i) is the largest position, less than m, such that pat[i..m] matches a suffix of pat[1..L(i)] and, moreover, this suffix is not preceded by character pat[i-1];
  - if no such copy of pat[i..m] exists in pat, let L(i)=0



  - Since pat[m+1..m]=ε, L(m+1) is the right-most position j s.t. pat[j]≠pat[m] (or 0 if all chars are equal).

Pre-processing for the "good suffix shift rule"

- L(i) gives the right end position of the rightmost copy of pat[i..m], which is not a suffix of pat, and is not preceded by pat[i-1]



- We will show that the values L(i) can be computed in O(m) time.

Computing the values L(i)

For a given i, the value L(i)<m is computed as follows:

- Take the suffix pat[i..m]=u
- Search for the rightmost occurrence of u in pat such that the preceding letter is not pat[i-1]
- If there is such an occurrence then L(i)= the right-end position of this occurrence of u
- Otherwise, L(i)=0

- Example: for the pattern pat=antecedence with |pat|=11, we have

L(12)=10, pat[12..11]=ε, antecedence

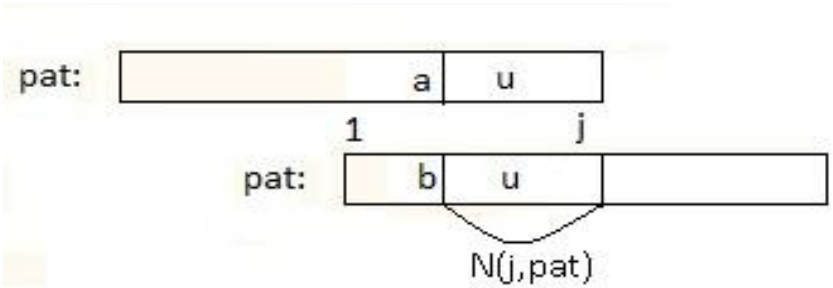
L(11)=8, pat[11]=e, antecedence,

L(10)=6, pat[10..11]=ce, antecedence

L(9)=..=L(2)=0, pat[9..11]=nce,…

Computing the values L(i)

- Define N(j,pat) to be the length of the longest common suffix of pat[1.. j] and pat itself (0≤N(j,pat)≤j).



- Example: for the pattern pat=antecedence we have

N(1,pat)=N(2,pat) =N(3,pat)=N(5,pat)= N(7,pat)= N(9,pat)= N(10,pat)= 0

N(4,pat)=1, (antecedence)

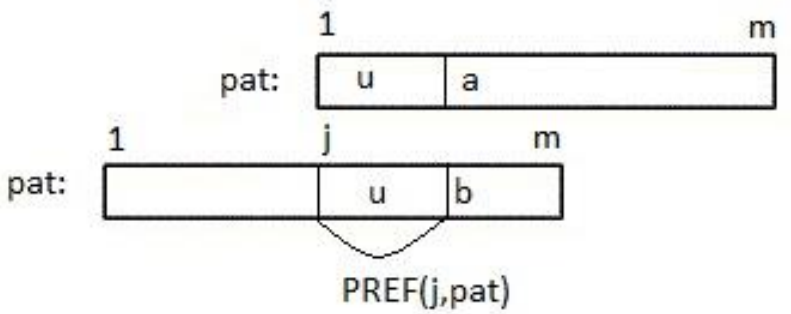
N(6,pat)=2, (antecedence)

N(8,pat)=1, (antecedence)

N(11,pat)=11 (antecedence)

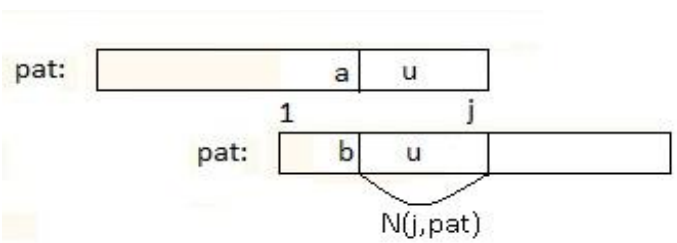
Computing the values L(i)

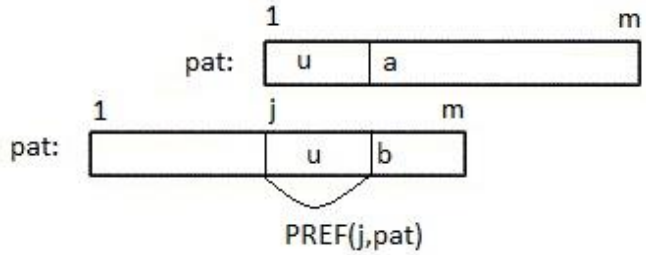
- Define PREF(j,pat) to be the length of the longest common prefix of pat[j.. m] and pat itself.



Computing the values L(i)

- Then the N(j,pat) (longest common suffix) values and the PREF(j,pat) (longest common prefix) values are reverses of each other, i.e., N(j,pat)=PREF(m-j+1,pat^r), where pat^r is the reverse of pat





- Example:

j: 12345678 k: 12345678

pat: aamunamu pat^r: umanumaa

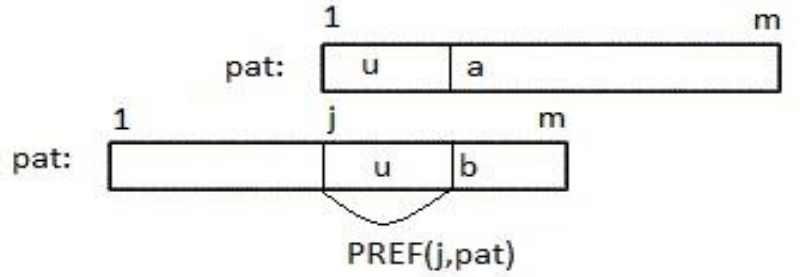
N(4,pat)=PREF(5,pat^r)

Computing the values L(i)

- The values N(j,pat) can be computed from PREF(j,pat^r).

- If there is no confusion for which pattern we compute them, we can use only PREF(j) instead of PREF(j,pat).

- PREF(j)=max{i | pat[j..j+i-1] is a prefix of pat}

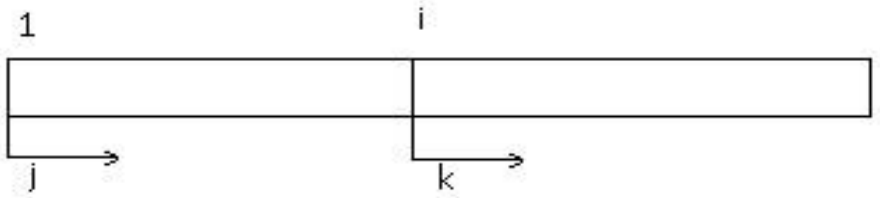


- The table PREF is called also the table of prefixes.

- There is an immediate "naive" approach for computing the table of prefixes.

Computing the table of prefixes

```
procedure Naive_Prefix_Table
    for i = 1 to m do begin
        j := 1; k := i; length := 0;
        while (k ≤ m) and (pat[j] = pat[k]) do begin
                j := j + 1;
                k := k + 1;
                length := length + 1;
        end;
        PREF(i) := length;
    end;
```



- This procedure has quadratic time complexity.
- There is a linear algorithm (O(m) time complexity, where m=|pat|) for computing PREF(i).
- So, the values N(j,pat) can be computed in O(m) time.

BONUS TOPIC : Computing the table of prefixes in O(m)

- We use the auxiliary function
  - Naive_Scan(p,q)=max{k≥1 | pat[p..p+k-1]=pat[q..q+k-1]}
  - If there is no such k, then Naive_Scan(p,q)=0.

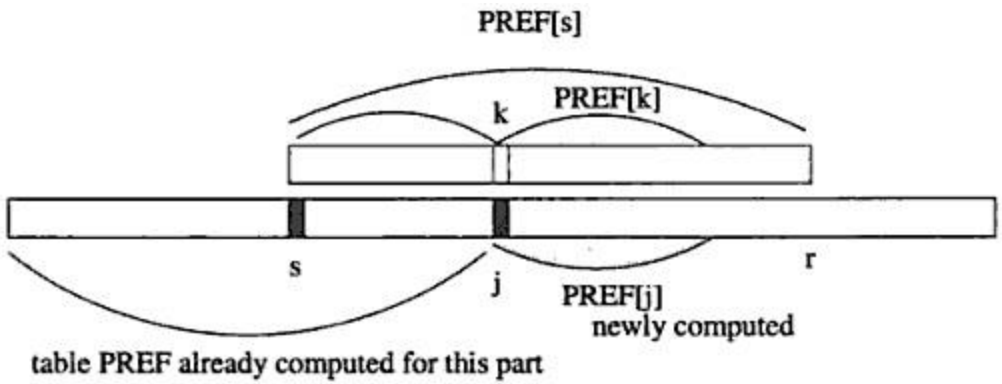
- There is a very simple algorithm for this function, with time complexity (i.e., the number of comparisons) at most k+1:

```
function Naive_Scan(p,q)
    result := 0;
    while (p ≤ m) and (q ≤ m) do begin
        if (pat[p] ≠ pat[q]) then break;
        p := p+1; q := q+1; result := result+1;
    end;
    return result;
```

BONUS TOPIC : Computing the table of prefixes in O(m)

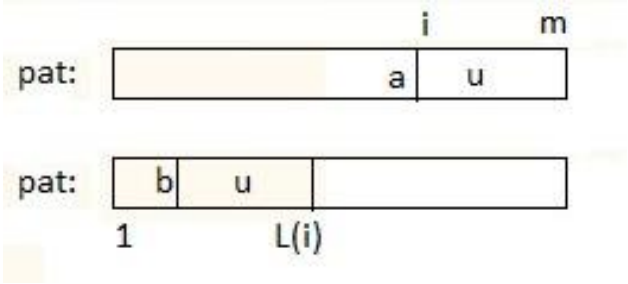
```
procedure Compute_Prefixes
    PREF(1) = 0;  s := 1;
    for j := 2 to m do begin
        k := j − s + 1;  r := s + PREF(s) − 1;
        if r < j then begin
            PREF(j) := Naive_Scan(j, 1);
            if PREF(j) > 0 then s := j;
        end else if PREF(k) + k < PREF(s) then
            PREF(j) := PREF(k)
        else begin
            x := Naive_Scan(r + 1, r − j + 2)
            PREF(j) := r − j + 1 + x;  s := j;
        end
    end;
    PREF(1) := m;
```

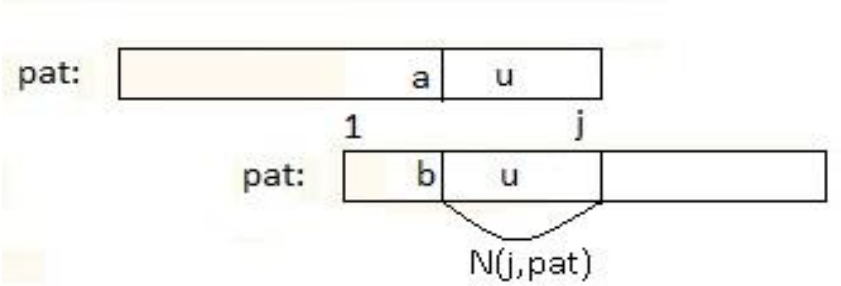
- The algorithm scans pat from left to right.
- When computing PREF(j), the values PREF(t) with t<j are already computed
- the number s<j is such that s+PREF(s)-1 is maximal



Computing the values L(i)

- L(i) gives the right end position of the rightmost copy of pat[i..m], which is not a suffix of pat, and is not preceded by pat[i-1]





- Theorem: If L(i) > 0, then L(i) is the largest j < m for which N(j,pat) = |pat[i..m]| (= m-i+1).

Proof: Such j is the right endpoint of the rightmost copy of pat[i..m] which is not preceded by pat[i-1].

Computing the values L(i)

- Since N(j,pat) = PREF(m-j+1,pat^r), we can compute the N(j,pat) values in O(m) time, where m=|pat|.

- Thus, the L(i) values can be computed in O(m) time by locating the largest j s.t. N(j,pat) =m-i+1.

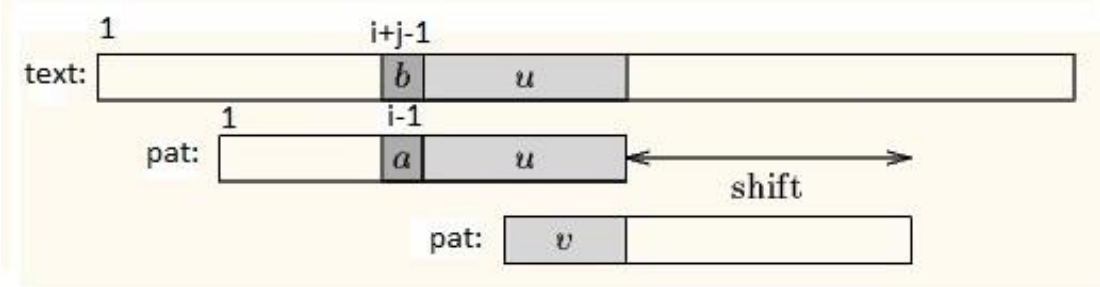
Computing the values L(i) in O(m) time

- L(i) is the largest j such that N(j,pat) =m-i+1, i.e., i = m-N(j,pat)+1

```
procedure Compute_Table_L
    for i := 2 to m+1 do L(i) := 0;
    for j := 1 to m-1 do begin
        i := m - N(j, pat) + 1;
        L(i) := j;
    end;
```

Pre-processing for the "good suffix shift rule"

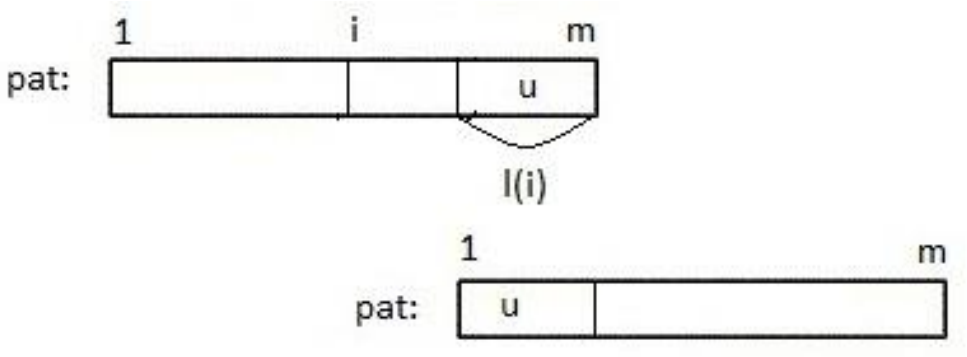
- If the factor text[*i*+*j* .. *j*+*m*] does not occur anywhere else in the pattern, the shift consists in aligning the longest suffix *v* of text[*i*+*j* .. *j*+*m*] with a matching prefix of *pat:*



Pre-processing for the "good suffix shift rule"

- How to compute the smallest shift that aligns a matching prefix of pat with a suffix of the successfully matched substring u = pat[i..m]?

- For i≥2, let l(i) = the length of the largest suffix of pat[i..m] that is also a prefix of pat, it one exists. Otherwise, l(i)=0.



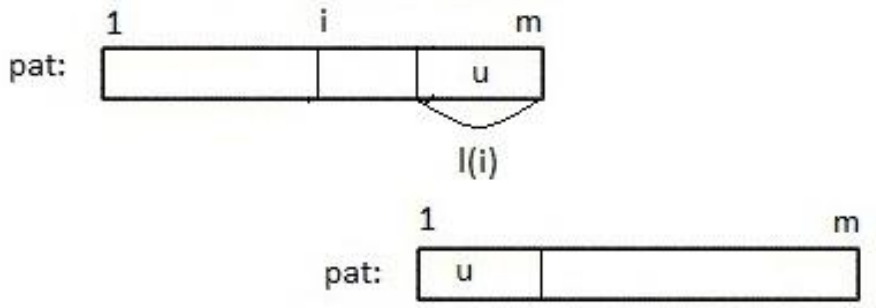
Pre-processing for the "good suffix shift rule"

- Example: Let pat=ababa, with m=|pat|=5

l(6)=0 (since pat[m+1..m]=ε)

l(5)=l(4)=1 (pat[5]=a, pat[4..5]=ba, the largest suffix is "a")

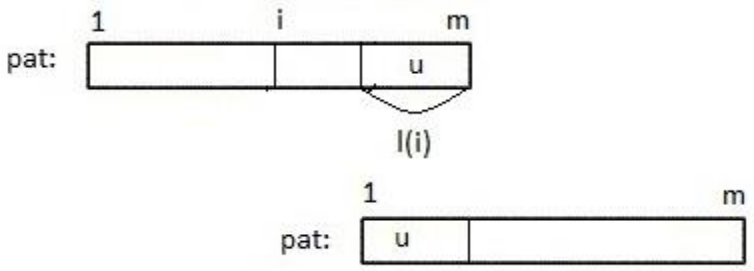
l(3)=l(2)=3 (pat[3..5]=aba, pat[2..5]=baba, the largest suffix is "aba")

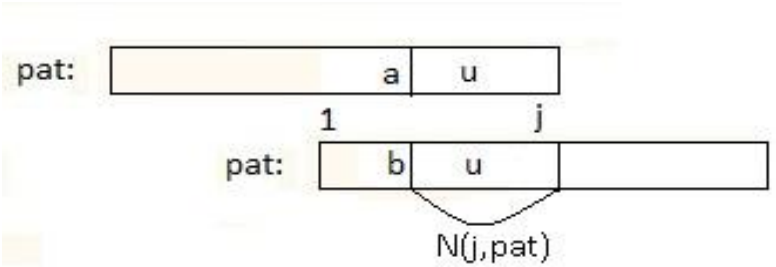


- l(i) = the length of the largest suffix of pat[i..m] that is also a prefix of pat, it one exists. Otherwise, l(i)=0.

Pre-processing for the "good suffix shift rule"

- Theorem: The value l(i) is the largest j≤|pat[i..m]| such that N(j,pat)=j.



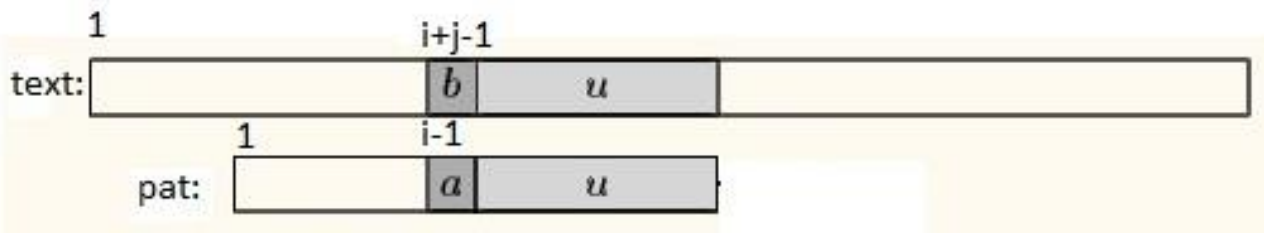


- This allows us to compute the l(i) values in time O(m), where m=|pat|.

Shifts by the Good Suffix Rule

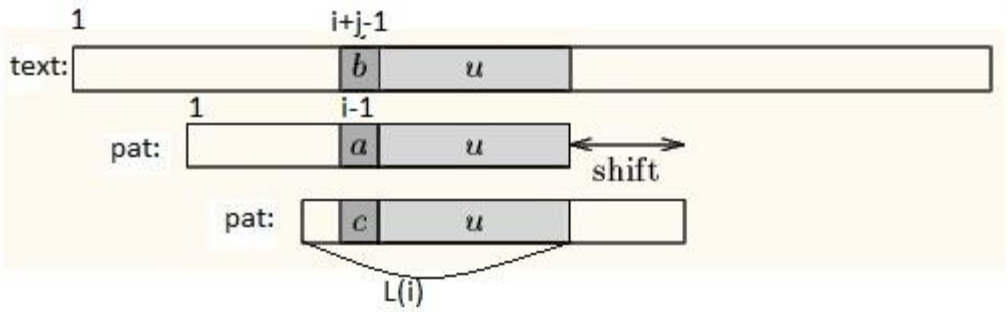
- The computed values L(i) and l(i) are used in the Boyer-Moore algorithm to compute the length of a shift using the good suffix rule.

- Suppose, we have found an occurrence of pat[i..m] but there is a mismatch on pat[i-1]

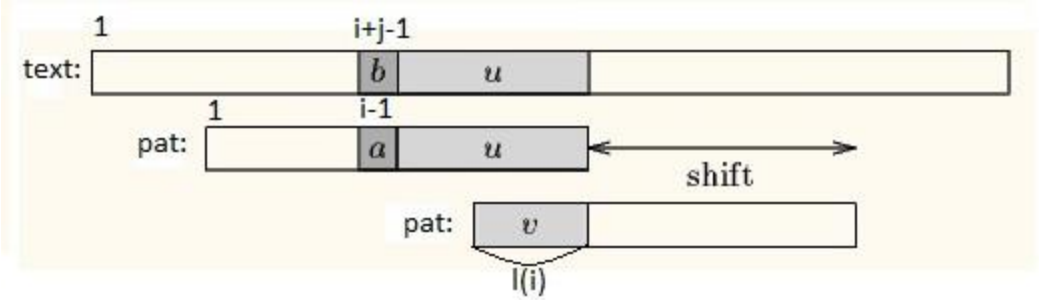


Shifts by the Good Suffix Rule

- If L(i)>0, then BM algorithm shifts pat by m-L(i) characters to the right, i.e., the prefix of length L(i) of the shifted pat aligns with the suffix of length L(i) of the unshifted pat.



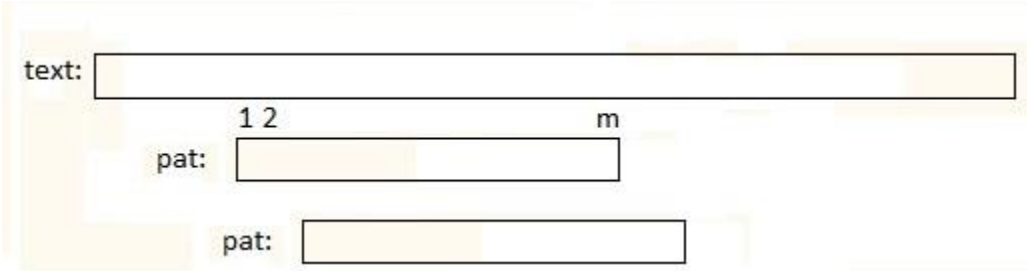
- If L(i)=0, then the good shift rule shifts pat by m-l(i) characters.



Shifts by the Good Suffix Rule

- If we have a mismatch already for the first comparison, i.e., on pat[m], then we shift the pattern pat by one position to the right.

- When an occurrence of pat has been found, then shift pat to the right by m-l(2) positions, to align a prefix of pat with the longest matching proper suffix of the occurrence.
  - l(2) = the length of the largest suffix of pat[2..m] that is also a prefix of pat, it one exists.



Shifts in the Boyer-Moore algorithm

- Which shift to use in the Boyer-Moore algorithm?

- Since neither the bad character rule nor the good suffix rule misses any occurrence, the Boyer-Moore algorithm shifts by the largest amount given by either of the two rules.

Boyer-Moore Algorithm

- **Pre-processing stage:**

- Given a pattern pat of length m, we compute the values L(i) and l(i) for all positions $2 \leq i \leq m+1$.

- We also compute R(x) for all characters x from the alphabet.

Boyer-Moore Algorithm

Boyer-Moore algorithm

$k := m$;
while $k \leq n$ do begin
  $i := m$; $h := k$;
  while $i > 0$ and $pat[i] = text[h]$ do begin
    $i := i - 1$;
    $h := h - 1$;
  end;
  if $i = 0$ then begin
    report an occurrence of *pat* in *text* at position k;
    $k := k + m - l(2)$;
  end;
  else shift pat (increase k) by the maximum amount
    determined by the bad character rule
    and the good suffix rule;
end

- In the algorithm, the index k represents the right-end of the current occurrence of pat that we try to match to the text.
- Thus, a shift of pat is implemented by increasing k with the appropriate amount of positions.

Boyer-Moore Algorithm

- If BM algorithm uses only the strong good suffix rule, then it has O(n) worst-case time complexity if the pattern does not occur in the text.

- If the pattern does appear in the text, then the algorithm runs in O(mn) worst-case.

- However, by slightly modifying this algorithm, it can achieve O(n) worst-case time complexity in all cases.
  - This was first proved by Galil in 1979.

Boyer-Moore Algorithm

- It can be proved that the BM algorithm has O(n) time complexity also when it uses both shift rules
  - Bad character shift rule
  - Good suffix shift rule

- On natural language texts the running time is almost always sub-linear