# Knuth - Morris - Pratt Algorithm (KMP)

Speedup from Naive Exact Matching Algorithm based on idea discussed in last lecture: information about the structure of pattern P can be used to avoid unecessary character comparisons:

Ex 1

```
T:   A B C A B D A B . . .
     | | | x
P:   A B C D A B D
           ↑
```

we can infer that comparing P at position T[2] is not needed (and T[3] as well), so we should compare sstarting at mismatch position instead.

Ex 2

```
T:  A B C A B C D A B C A . . .
    | | | | | x
    A B C A B D
```

Q: How far can we skip? To mismatch position?

A: No, there is a possible occurrence of P starting at T[3]

Q: How do we know that?

Substring P[3,4] matches P[1,2]. There is a substring of P ending at 4 that matches a prefix of P.

Formally:

Def: $spm_i(P)$ = length of longest substring of $P$ that ends at $i > 1$ and matches a prefix of $P$ such that $P[i+1] \neq P[spm_i + 1]$

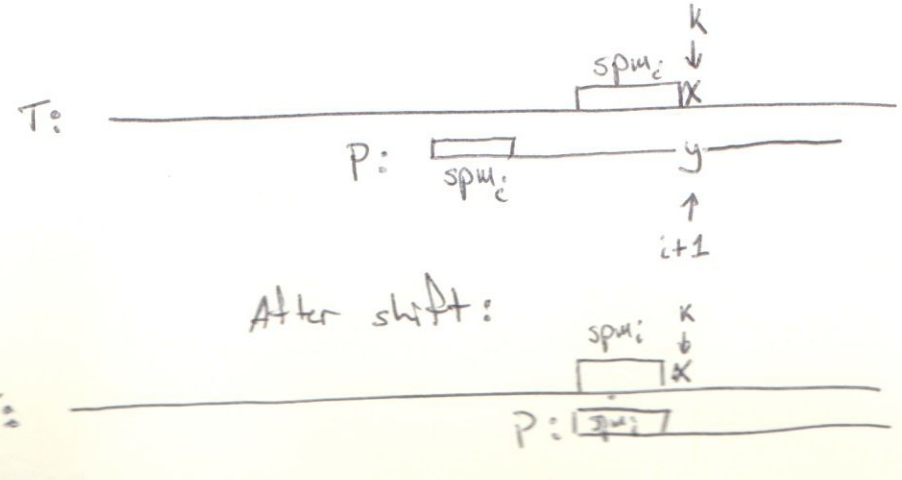

$x \neq y$

Note: (a) these are denoted as $sp'_i$ in the Gusfield book

(b) spm: suffix-prefix-mismatch

Shift Rule: For any alignment of $P$ and $T$, if the first mismatch occurs in position $i+1$ of $P$ and position $k$ of $T$, then shift $P$ $i - spm_i(P)$ places to the right. If no mismatch, shift $|P| - spm_{|P|}(P)$ places.

Note: After shift $P[1, \dots, spm_i]$ and $T[k-spm_i, \dots, k-1]$ are aligned and match



Ex:

P: ABCDEFGHABCD

$spm_{12} = 4$

```
T: ---------- ABCD EFGH ABCD E ----------
              |||| |||| |||| X
       P: ABCD EFGH ABCD C --------
                     i=12
```

shift 8 positions

```
T: ---------- ABCDEFGHABCDE ------------
                      ABCDEFGHABCDC --------
                          ↑
                      continue
                      comparing here
```

Running time

After each shift at most 1 character in T is compared again, so total # of comparisons is bounded by

$$|T| + s$$

where s is the number of shifts. But, since we always shift s to the right, s is bounded by $|T|$. So, total # of comparisons is bounded by $2|T|$.

Finally, how to calculate $spm_i(P)$

Def: $f(j)$ as the right end of z-box starting at position $j$


Def: $g(i)$ as $\min\{j \mid f(j) = i\}$ or 0 if empty set

In other words: The <u>left-most</u> starting point of z-boxes ending at position $i$

Thm: $spm_i = z_{g(i)}$ if $g(i) > 0$, otherwise 0

Pf: By definition


$\cancel{P[1, \ldots, z_{g(i)}] = P[1, \ldots, z_{g(}}$

$P[1, \ldots, z_{g(i)}] = P[g(i), \ldots, i]$ &

$x = P[z_{g(i)} + 1] \neq P[i+1] = y \implies spm_i \geq z_{g(i)}$.

Now, suppose $spm_i > z_{g(i)}$, then $\exists\, k < g(i)$ s.t. $f(k) = i$. This is a contradiction since $g(i)$ is minimum. Thus, $spm_i = z_{g(i)}$.

# Boyer-Moore Algorithm

Used in practice, has best performace in real cases.

Main Ideas:

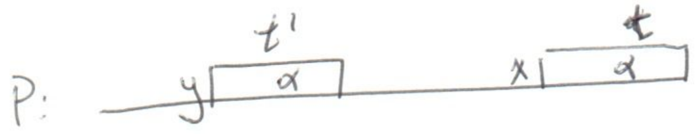
1. Compare P to T from right to left
2. Good suffix rule
3. Bad character rule

## ① Right to left comparison

the quick-brown fox
X||||
crown

## ② Good suffix rule: Apply in order

A:




shift so rightmost occurrence of matched suffix (α) is aligned to matched part in T

B:

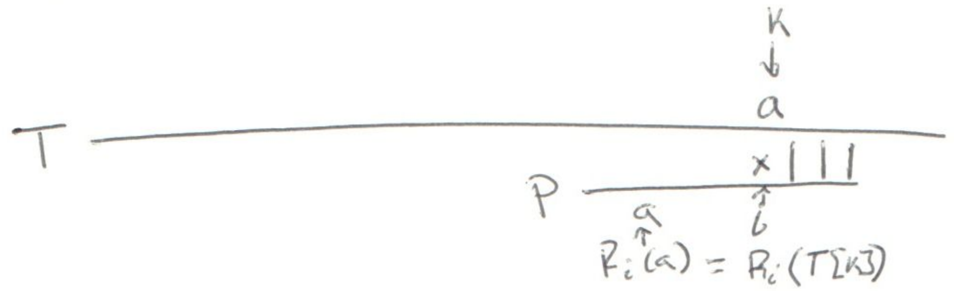

(α does not occur in P again)
Shift so longest proper prefix (β) that matches a suffix of α is matched to T.

C: If not A or C, shift |P| places

How can we use Z-algorithm to calculate these shifts?

## Bad character rule

Def: $R_i(x)$ as position of the rightmost occurrence of character $x$ before position $i$




Shift by $i - R_i(T[k])$ positions so the next occurrence of $T[k]$ is aligned to position $k$

Calculating $R_i(x)$

- Constructing table $R_i(x)$ is not desirable since depends on size of alphabet
- Instead use a collection of lists
    - $Occur[x]$ = list of positions where $x$ occurs in $P$ in decreasing order
- Find $R_i(x)$ by scanning list $Occur[x]$ until first index $< i$
- Time $O(n-i)$ since at most $n-i$ items in list can be $\geq i$

Boyer Moore

- Start at position 1 of $T$
- Compare $P$ to $T$ right to left until mismatch
- Apply biggest shift from good shift or bad character rule