Prepared by: [Arun Konagurthu]

FIT3155: Advanced Algorithms and Data Structures

Weeks 1 & 2: **Linear-time string pattern matching**

Faculty of Information Technology, Monash University

What is covered here?

Linear-time approaches to exact pattern matching problem on strings

- Gusfield's Z-algorithm
- Boyer-Moore's algorithm
- Knuth-Morris-Pratt's algorithm

Source material and recommended reading

- Dan Gusfield, Algorithms on Strings, Trees and Sequences, Cambridge University Press. (Chapters 1-2).

Exact pattern matching: Introduction

The exact pattern matching problem

Given a reference text **txt**$[1 \ldots n]$ and a pattern **pat**$[1 \ldots m]$, find **ALL** occurrences, if any, of **pat** in **txt**.

Example:

	1	2	3	4	5	6	7	8	9	10	11	12
txt =	b	b	a	b	a	x	a	b	a	b	a	y

	1	2	3
pat =	a	b	a

matched positions of **pat** in **txt** are at positions 3, 7, and 9

- The practical importance of this problem should be plainly obvious to anyone who uses a computer.
- Problem arises in innumerable applications
  - Word processing – `grep` command in Unix
  - Search Engines – Google
  - Library catalogs
  - ⋮

Naïve algorithm

```
n = |txt|
m = |pat|
for i from 1 to n-m+1  do
    for j from 1 to m do
        if txt[i+j-1] != pat[j] then
            break  // mismatch
        endif
    endfor;
    if (j == m+1) print i;
endfor
```

How many comparison of symbols does this approach perform in worst case?

Consider **txt**$[1\ldots 10] = \begin{array}{cccccccccc} {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} \\ \mathtt{a} & \mathtt{a} & \mathtt{a} & \mathtt{a} & \mathtt{a} & \mathtt{a} & \mathtt{a} & \mathtt{a} & \mathtt{a} & \mathtt{a} \end{array}$ and **pat**$[1\ldots 3] = \begin{array}{ccc} {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} \\ \mathtt{a} & \mathtt{a} & \mathtt{a} \end{array}$.
Total number of comparisons, in the **worst case**, is $(n - m + 1) * m = 24$.
In general the number of comparison grows as $O(mn)$.

Early ideas for speeding up the naïve method

- try to shift **pat** by > 1 character w.r.t. **txt** when mismatch occurs ...
  - ...but never shift so far as to miss any occurrence of **pat** in **txt**;
  - if this can be achieved, we save unnecessary comparisons, and move **pat** along **txt** more rapidly.
- Specifically, where possible, we would want to shift by skipping/jumping over parts of **txt** unrelated to **pat**.

Naïve approach makes too many unnecessary comparisons

Example

	1	2	3	4	5	6	7	8	9	10	11	12	13
txt:	x	a	b	x	y	a	b	x	y	a	b	x	z
pat:	a	b	x	y	a	b	x	z					
iteration-1	✗												
pat:		a	b	x	y	a	b	x	z				
iteration-2		✓	✓	✓	✓	✓	✓	✓	✗				
pat:			a	b	x	y	a	b	x	z			
iteration-3			✗										
pat:				a	b	x	y	a	b	x	z		
iteration-4				✗									
pat:					a	b	x	y	a	b	x	z	
iteration-5					✗								
pat:						a	b	x	y	a	b	x	z
iteration-6						✓	✓	✓	✓	✓	✓	✓	✓

Overall 20 comparisons in the naïve approach, on this example.

A smarter approach reduces unnecessary comparisons

Scenario 1:

A smarter algorithm can gather that, after the ninth comparison, the next three comparisons of the naïve algorithm will be mismatches.

	1	2	3	4	5	6	7	8	9	10	11	12	13
txt:	x	a	b	x	y	a	b	x	y	a	b	x	z
pat:	a	b	x	y	a	b	x	z					
iteration-1	✗												
pat:		a	b	x	y	a	b	x	z				
iteration-2		✓	✓	✓	✓	✓	✓	✓	✗				
pat:						a	b	x	y	a	b	x	z
iteration-3						✓	✓	✓	✓	✓	✓	✓	✓

Overall, this 'smarter' algorithm saves 3 comparisons, on this example.

How does this algorithm achieve this?

After the ninth comparison, the algorithm knows that the first seven characters of **pat** match characters 2 through to 8 of **txt**. It can gather that the first character of **pat** ('a') does not occur until position 6 in **txt**. This is enough information to conclude that there are no possible matches in **txt** of **pat** to the left of position 6, allowing larger jumps.

An even smarter approach ...

Scenario 2:

in fact, an 'even smarter' algorithm can gather more info after the ninth comparison, beyond scenario 1, and save 3 more comparisons.

	1	2	3	4	5	6	7	8	9	10	11	12	13
txt:	x	a	b	x	y	a	b	x	y	a	b	x	z
pat:	a	b	x	y	a	b	x	z					
iteration-1	✗												
pat:		a	b	x	y	a	b	x	z				
iteration-2		✓	✓	✓	✓	✓	✓	✓	✗				
pat:						a	b	x	y	a	b	x	z
iteration-3									✓	✓	✓	✓	✓

Overall, this 'even smarter' algorithm saves 6 comparisons over 'naïve'.

How does this algorithm achieve this?

An even smarter algorithm can preprocess **pat**, and from it know that **pat**$[1..3]$ (i.e 'abx') appears again at **pat**$[5..7]$. So, after the ninth comparison, the algorithm realizes **pat**[5..7]=**txt**[6..8]. But since **pat**[1..3]=**pat**[5..7], after shift, when **pat**$[1..]$ is being compared with **txt**[6 ...], we already know **pat**$[1..3]$ = **txt**$[6..8]$, saving us 3 unnecessary comparisons.

Take home message from these illustrated examples

- The examples shown in the previous slides illustrate the kind of ideas that allow unnecessary comparisons to be skipped/jumped over.
  - Note, we haven't yet rationalized how an algorithm can **`efficiently`** implement such ideas.
- There are some algorithms that permit the efficient realization of the above ideas ...
- ...and we will study 3 remarkable algorithms that can be implemented to run in **`linear time`**.
  - **Gusfield's Z-algorithm**
  - **`Boyer-Moore's algorithm`**
  - **`Knuth-Morris-Pratt's algorithm`**

1. Gusfield's Z-algorithm

Comments:

- This is actually a linear-time algorithm to **preprocess** a given string.
- After preprocessing, the output from this algorithm (Z_i-values) can be used to address a versatile set of problems that arise in strings.
- In this unit you will witness its various uses.

Gusfield's Z-algorithm – Defining Z_i

Definition of Z_i:

For a string **str**$[1 \dots n]$, define Z_i (for each position $i > 1$ in **str**) as the **length** of the **longest substring** starting at **position** i of **str** that **matches its prefix** (i.e., **str**$[i \dots i{+}Z_i\text{-}1]$ = **str**$[1 \dots Z_i]$).

Example

	1	2	3	4	5	6	7	8	9	10	11
str=	a	a	b	c	a	a	b	x	a	a	y

$Z_2 = 1 \quad Z_7 = 0$

$Z_3 = 0 \quad Z_8 = 0$

$Z_4 = 0 \quad Z_9 = 2$

$Z_5 = 3 \quad Z_{10} = 1$

$Z_6 = 1 \quad Z_{11} = 0$

Gusfield's Z-algorithm – Defining Z_i-box

Definition of Z-box:

For a string **str**$[1..n]$, **and for any** $i > 1$ **such that** $Z_i > 0$, a Z_i-box is defined as the interval $[i \ldots i+Z_i\text{-}1]$ of **str**.

Example

$$\texttt{str} = \overset{1}{a}\ \overset{2}{a}\ \overset{3}{b}\ \overset{4}{c}\ \overset{5}{a}\ \overset{6}{a}\ \overset{7}{b}\ \overset{8}{x}\ \overset{9}{a}\ \overset{10}{a}\ \overset{11}{y}$$

$Z_2 = 1$	Z_2-box $= [2..2]$	$Z_7 = 0$	undefined
$Z_3 = 0$	undefined	$Z_8 = 0$	undefined
$Z_4 = 0$	undefined	$Z_9 = 2$	Z_9-box $= [9..10]$
$Z_5 = 3$	Z_5-box $= [5..7]$	$Z_{10} = 1$	Z_{10}-box $= [10..10]$
$Z_6 = 1$	Z_6-box $= [6..6]$	$Z_{11} = 0$	undefined

Gusfield's Z-algorithm – Defining r_i

Definition of r_i:

For a string **str**$[1..n]$, and for all $i > 1$, r_i is the **right-most endpoint** of all Z-boxes that begin at or before position i.

Alternately, r_i is the largest value of j+Z_j-1 over all $1 < j \leq i$, such that $Z_j > 0$.

Example

```
      1 2 3 4 5 6 7 8 9 10 11
str=  a a b c a a b x a a  y
```

$Z_2 = 1$	$Z_7 = 0$	Z_2-box = [2..2]		$r_2 = 2$	$r_7 = 7$
$Z_3 = 0$	$Z_8 = 0$	Z_5-box = [5..7]		$r_3 = 2$	$r_8 = 7$
$Z_4 = 0$	$Z_9 = 2$	Z_6-box = [6..6]		$r_4 = 2$	$r_9 = 10$
$Z_5 = 3$	$Z_{10} = 1$	Z_9-box = [9..10]		$r_5 = 7$	$r_{10} = 10$
$Z_6 = 1$	$Z_{11} = 0$	Z_{10}-box = [10..10]		$r_6 = 7$	$r_{11} = 10$

Gusfield's Z-algorithm – Defining l_i

Definition of l_i:

For a string **str**$[1..n]$, and for all $i > 1$, l_i is the **left end** of the Z-box that ends at r_i.

In case there is more than one Z-box ending at r_i, then l_i can be chosen to be the left end of **any** of those Z-boxes.

Example

	1	2	3	4	5	6	7	8	9	10	11
str=	a	a	b	c	a	a	b	x	a	a	y

$$
\begin{array}{ll||c||ll}
Z_2 = 1 & Z_7 = 0 & Z_2\text{-box} = [2..2] & l_2 = 2 & l_7 = 5 \\
Z_3 = 0 & Z_8 = 0 & Z_5\text{-box} = [5..7] & l_3 = 2 & l_8 = 5 \\
Z_4 = 0 & Z_9 = 2 & Z_6\text{-box} = [6..6] & l_4 = 2 & l_9 = 9 \\
Z_5 = 3 & Z_{10} = 1 & Z_9\text{-box} = [9..10] & l_5 = 5 & l_{10} = 9 \\
Z_6 = 1 & Z_{11} = 0 & Z_{10}\text{-box} = [10..10] & l_6 = 5 & l_{11} = 9
\end{array}
$$

Another worked out example: calculating Z_i, Z_i-box, and (l_i, r_i) values — digest this during self-study

Example

```
      1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
str= a a b a a b c a x a  a  b  a  a  b  c  y
```

$Z_2 = 1$	$Z_{10} = 7$	Z_2-box = [2..2]	$(l_2, r_2) = (2, 2)$	$(l_{10}, r_{10}) = (10, 16)$
$Z_3 = 0$	$Z_{11} = 1$	Z_4-box = [4..6]	$(l_3, r_3) = (2, 2)$	$(l_{11}, r_{11}) = (10, 16)$
$Z_4 = 3$	$Z_{12} = 0$	Z_5-box = [5..5]	$(l_4, r_4) = (4, 6)$	$(l_{12}, r_{12}) = (10, 16)$
$Z_5 = 1$	$Z_{13} = 3$	Z_8-box = [8..8]	$(l_5, r_5) = (4, 6)$	$(l_{13}, r_{13}) = (10, 16)$
$Z_6 = 0$	$Z_{14} = 1$	Z_{10}-box = [10..16]	$(l_6, r_6) = (4, 6)$	$(l_{14}, r_{14}) = (10, 16)$
$Z_7 = 0$	$Z_{15} = 0$	Z_{11}-box = [11..11]	$(l_7, r_7) = (4, 6)$	$(l_{15}, r_{15}) = (10, 16)$
$Z_8 = 1$	$Z_{16} = 0$	Z_{13}-box = [13..15]	$(l_8, r_8) = (8, 8)$	$(l_{16}, r_{16}) = (10, 16)$
$Z_9 = 0$	$Z_{17} = 0$	Z_{14}-box = [14..14]	$(l_9, r_9) = (8, 8)$	$(l_{17}, r_{17}) = (10, 16)$

Main point of Gusfield's Z-algorithm!

- In the previous slides, for any given string, we have defined: $\{Z_i$, Z_i-box, l_i, $r_i\}$.
- The fundamental **preprocessing task** of Gusfield's Z-algorithm relies on computing these values, given some string, in **linear** time.
- That is, for a string **str**$[1..n]$, we would like to compute
  $\{Z_i$, Z_i-box, l_i, $r_i\}$
  for each position $i > 1$ in **str** in $O(n)$-time.

Main point of Gusfield's Z-algorithm!

- In the previous slides, for any given string, we have defined: $\{Z_i$, Z_i-box, l_i, $r_i\}$.
- The fundamental **preprocessing task** of Gusfield's Z-algorithm relies on computing these values, given some string, in **linear** time.
- That is, for a string **str**$[1..n]$, we would like to compute $\{Z_i$, Z_i-box, l_i, $r_i\}$ for each position $i > 1$ in **str** in $O(n)$-time.

Plan ahead

Once we convince ourselves of the linear time preprocessing, we can then use this for linear-time exact pattern matching (one of its many uses).

Overview of the linear-time preprocessing

- In this preprocessing phase, we compute $\{Z_i,\ Z_i\text{-box},\ l_i,\ r_i\}$ values **`for each successive position`** i, starting from $i = 2$.*
- All successively computed Z_i values are remembered.
  - Note: Each Z_i-box interval can be computed from its corresponding Z_i value in $O(1)$ time
- At each iteration, to compute (l_i,r_i), this preprocessing only needs values of (l_j, r_j) for $j = i - 1$.
  - Note: no earlier (l_j, r_j) values are needed ...
  - ...so, temporary variables (l, r) can be used to keep track of the most recently computed (l_{i-1}, r_{i-1}) values to update (l_i,r_i).

Let's see how this all works out in practice.

*We are using 1-based array indexing throughout this unit, unless specified otherwise.

preprocessing in practice – base case

- To begin, compute Z_2 by explicit **left-to-right** comparison of characters **str**$[2 \ldots]$ with **str**$[1 \ldots]$ until a mismatch is found.
  - Note: Z_2 is the length of the **matching** substring.
- If $Z_2 > 0$
  - set r (i.e., r_2) to $Z_2 + 1$
  - set l (i.e., l_2) to 2
- else (i.e., if $Z_2 == 0$)
  - set r (i.e., r_2) to 0
  - set l (i.e., l_2) to 0

preprocessing in practice – inductive assumption and general cases

Assume inductively ...

- ...we have correctly computed the values Z_2 through to Z_{k-1}.
- ...that r currently holds r_{k-1},
- ...that l currently holds l_{k-1}.

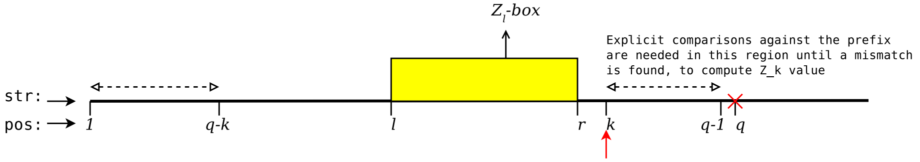
For computing Z_k at position k, these two scenarios arise:

Case 1: If $k > r$

Case 2: Else $k \leq r$

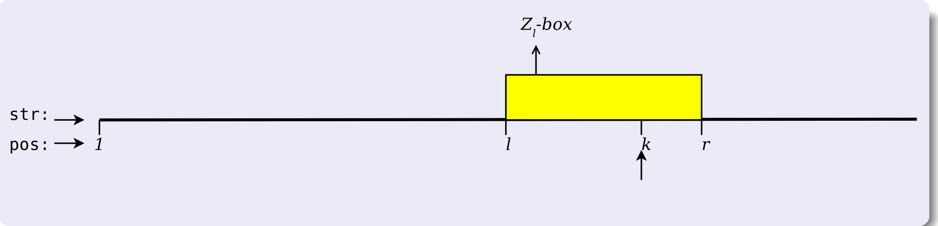
Let's see how to have to handle these two cases.

preprocessing in practice – Case 1: $k > r$



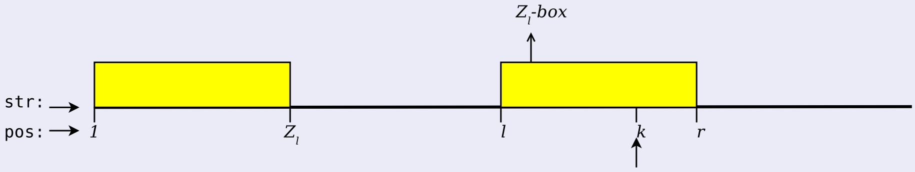
- CASE 1, if $k > r$:
  - Compute Z_k by explicitly comparing characters **str**$[k \dots q-1]$ with **str**$[1 \dots q-k]$ until mismatch is found at some $q \geq k$.
  - If $Z_k > 0$:
    - ★ set r (i.e., r_k) to $q-1$.
    - ★ set l (i.e., l_k) to k.
  - // Otherwise they retain the same (l, r) values as before

preprocessing in practice – Case 2: $k \leq r$



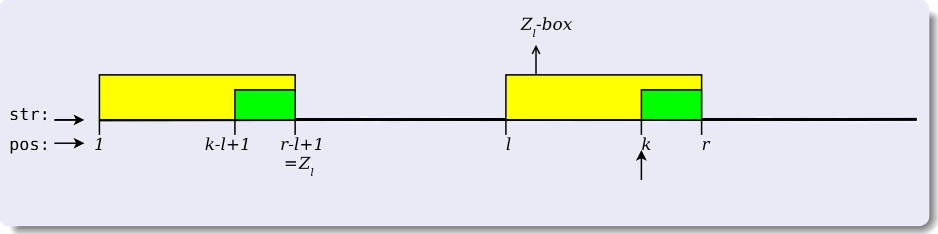
- CASE 2, if $k \leq r$:
  - The character **str**$[k]$ lies in the substring **str**$[l \ldots r]$ (i.e., within Z_l-box).

preprocessing in practice – Case 2: $k \leq r$



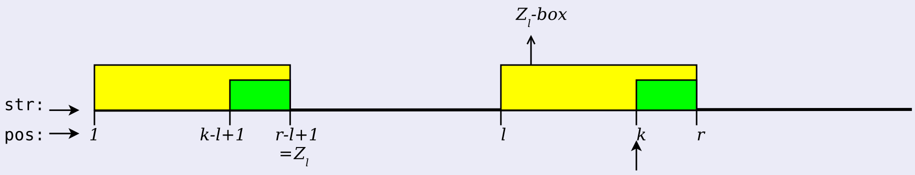
- CASE 2, if $k \leq r$:
  - The character **str**$[k]$ lies in the substring **str**$[l \ldots r]$ (i.e., within Z_l-box).
  - By the definition of Z_l-box, **str**$[l \ldots r]$=**str**$[1 \ldots Z_l]$.

preprocessing in practice – Case 2: $k \leq r$



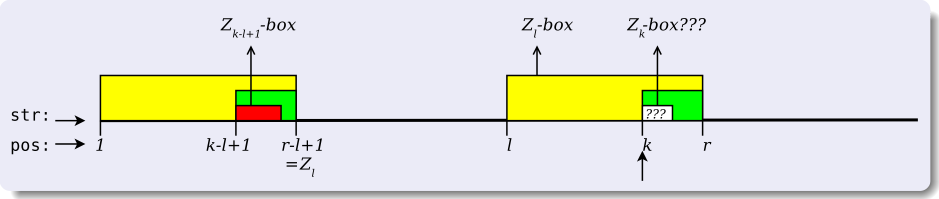
- CASE 2, if $k \leq r$:
  - The character **str**$[k]$ lies in the substring **str**$[l \ldots r]$ (i.e., within Z_l-box).
  - By the definition of Z_l-box, **str**$[l \ldots r]$=**str**$[1 \ldots Z_l]$.
  - This implies the character **str**$[k]$ is identical to **str**$[k - l + 1]$

preprocessing in practice – Case 2: $k \leq r$



- CASE 2, if $k \leq r$:
  - The character **str**$[k]$ lies in the substring **str**$[l \dots r]$ (i.e., within Z_l-box).
  - By the definition of Z_l-box, **str**$[l \dots r]$=**str**$[1 \dots Z_l]$.
  - This implies the character **str**$[k]$ is identical to **str**$[k - l + 1]$
  - By extending this logic, it also implies that the substring **str**$[k \dots r]$ is identical to **str**$[k - l + 1 \dots Z_l]$.

preprocessing in practice – Case 2: $k \leq r$



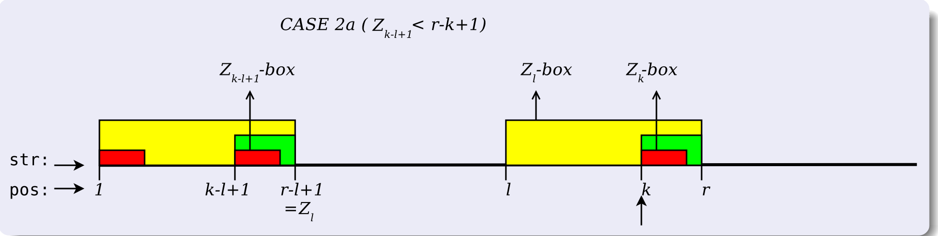
- CASE 2, if $k \leq r$:
  - The character **str**$[k]$ lies in the substring **str**$[l \ldots r]$ (i.e., within Z_l-box).
  - By the definition of Z_l-box, **str**$[l \ldots r]$=**str**$[1 \ldots Z_l]$.
  - This implies the character **str**$[k]$ is identical to **str**$[k-l+1]$
  - By extending this logic, it also implies that the substring **str**$[k \ldots r]$ is identical to **str**$[k-l+1 \ldots Z_l]$.
  - But, in previous iterations, we already have computed Z_{k-l+1} value.
    - ★ can the value of Z_{k-l+1} inform the computation of Z_k?

preprocessing – case 2 (continued)

In the previous slide, we asked "*can the value of* Z_{k-l+1} *inform the computation of* Z_k*?*". The answer is **yes**, and this can be handled by two **sub**-cases" CASES 2a and 2b (described below):

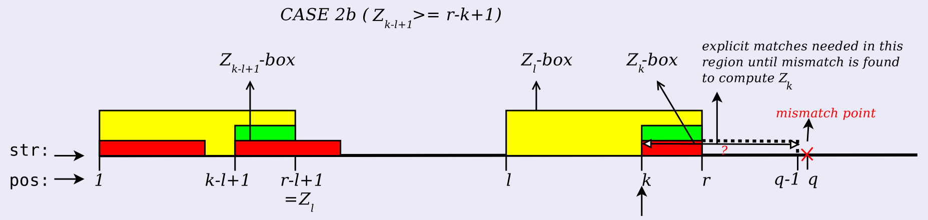
preprocessing – case 2 (continued)

- CASE 2a, if $Z_{k-l+1} < r - k + 1$:
  - set Z_k to Z_{k-l+1}.
  - r and l remain unchanged.



preprocessing – case 2 (continued)

- CASE 2b, if $Z_{k-l+1} \geq r - k + 1$:
  - Z_k must also be $\geq r - k + 1$
  - So, start explicitly comparing **str**$[r + 1]$ with **str**$[r - k + 2]$ and so on until mismatch occurs.
  - Say the mismatch occurred at position $q \geq r + 1$, then:
    - ★ set Z_k to $q - k$,
    - ★ set r to $q - 1$.
    - ★ set l to k.



Preprocessing (of Z_i values) for **str**$[1..n]$ takes $O(n)$ time

- Total time is directly proportional to the SUM of:

Preprocessing (of Z_i values) for $\texttt{str}[1..n]$ takes $O(n)$ time

- Total time is directly proportional to the SUM of:
  1. the number of iterations

Preprocessing (of Z_i values) for **str**$[1..n]$ takes $O(n)$ time

- Total time is directly proportional to the SUM of:
  1. the number of iterations
  2. the number of character comparisons (**matches** or **mismatches**).

Preprocessing (of Z_i values) for **str**$[1..n]$ takes $O(n)$ time

- Total time is directly proportional to the SUM of:
  1. the number of iterations
  2. the number of character comparisons (**matches** or **mismatches**).
- This preprocessing has $n - 1$ iterations.

Preprocessing (of Z_i values) for **str**$[1..n]$ takes $O(n)$ time

- Total time is directly proportional to the SUM of:
  1. the number of iterations
  2. the number of character comparisons (**matches** or **mismatches**).
- This preprocessing has $n-1$ iterations.
- The number of **matches** and **mismatches** are both at most n, because:

Preprocessing (of Z_i values) for **str**$[1..n]$ takes $O(n)$ time

- Total time is directly proportional to the SUM of:
  1. the number of iterations
  2. the number of character comparisons (**matches** or **mismatches**).
- This preprocessing has $n-1$ iterations.
- The number of **matches** and **mismatches** are both at most n, because:
  - $r_k \geq r_{k-1}$ (for all iterations)

Preprocessing (of Z_i values) for **str**$[1..n]$ takes $O(n)$ time

- Total time is directly proportional to the SUM of:
  1. the number of iterations
  2. the number of character comparisons (**matches** or **mismatches**).
- This preprocessing has $n-1$ iterations.
- The number of **matches** and **mismatches** are both at most n, because:
  - $r_k \geq r_{k-1}$ (for all iterations)
  - update to r_k is of the form $r_k = r_{k-1} + \delta$ (where $\delta \geq 0$).

Preprocessing (of Z_i values) for **str**$[1..n]$ takes $O(n)$ time

- Total time is directly proportional to the SUM of:
  1. the number of iterations
  2. the number of character comparisons (**matches** or **mismatches**).
- This preprocessing has $n-1$ iterations.
- The number of **matches** and **mismatches** are both at most n, because:
  - $r_k \geq r_{k-1}$ (for all iterations)
  - update to r_k is of the form $r_k = r_{k-1} + \delta$ (where $\delta \geq 0$).
  - But $r_k \leq n$.

Preprocessing (of Z_i values) for **str**$[1..n]$ takes $O(n)$ time

- Total time is directly proportional to the SUM of:
  1. the number of iterations
  2. the number of character comparisons (**matches** or **mismatches**).
- This preprocessing has $n-1$ iterations.
- The number of **matches** and **mismatches** are both at most n, because:
  - $r_k \geq r_{k-1}$ (for all iterations)
  - update to r_k is of the form $r_k = r_{k-1} + \delta$ (where $\delta \geq 0$).
  - But $r_k \leq n$.
  - Thus, there are at most n **matches** or **mismatches**.

Using Z-algorithm for linear-time exact pattern matching

Recall the exact pattern matching problem

Given a reference text **txt**$[1 \ldots n]$ and a pattern **pat**$[1 \ldots m]$, find **ALL** occurrences, if any, of **pat** in **txt**.

Using Z-algorithm for linear-time exact pattern matching

Recall the exact pattern matching problem

Given a reference text **txt**$[1 \ldots n]$ and a pattern **pat**$[1 \ldots m]$, find **ALL** occurrences, if any, of **pat** in **txt**.

Realizing a linear-time solution using Gusfield's Z-algorithm/preprocessing

- Construct a new string **str** by concatenation as follows:
  $\texttt{str} = \texttt{pat}[1 \ldots m] + \$ + \texttt{txt}[1 \ldots n]$.
  Note, $|\texttt{str}| = m + 1 + n$.

Using Z-algorithm for linear-time exact pattern matching

Recall the exact pattern matching problem

Given a reference text $\texttt{txt}[1 \ldots n]$ and a pattern $\texttt{pat}[1 \ldots m]$, find **ALL** occurrences, if any, of **pat** in **txt**.

Realizing a linear-time solution using Gusfield's Z-algorithm/preprocessing

- Construct a new string **str** by concatenation as follows:
  $\texttt{str} = \texttt{pat}[1 \ldots m] + \$ + \texttt{txt}[1 \ldots n]$.
  Note, $|\texttt{str}| = m + 1 + n$.
- Preprocess Z_i values corresponding to **str**, for $1 < i \leq m + n + 1$.

Using Z-algorithm for linear-time exact pattern matching

Recall the exact pattern matching problem

Given a reference text $\texttt{txt}[1 \ldots n]$ and a pattern $\texttt{pat}[1 \ldots m]$, find **ALL** occurrences, if any, of $\texttt{pat}$ in $\texttt{txt}$.

Realizing a linear-time solution using Gusfield's Z-algorithm/preprocessing

- Construct a new string $\texttt{str}$ by concatenation as follows:
  $\texttt{str} = \texttt{pat}[1 \ldots m] + \$ + \texttt{txt}[1 \ldots n]$.
  Note, $|\texttt{str}| = m + 1 + n$.
- Preprocess Z_i values corresponding to $\texttt{str}$, for $1 < i \leq m + n + 1$.
- For any $i > m + 1$, all $Z_i = m$ identifies an occurrence of $\texttt{pat}$[1 ... m] at position i in $\texttt{str}$, and hence at position $i - (m + 1)$ in $\texttt{txt}$. That is, $\texttt{pat}[1 \ldots m] = (\texttt{str}[i \ldots i + m - 1] \equiv \texttt{txt}[i - (m + 1) \ldots i - 2])$.

Using Z-algorithm for linear-time exact pattern matching

Recall the exact pattern matching problem

Given a reference text $\texttt{txt}[1 \dots n]$ and a pattern $\texttt{pat}[1 \dots m]$, find **ALL** occurrences, if any, of **pat** in **txt**.

Realizing a linear-time solution using Gusfield's Z-algorithm/preprocessing

- Construct a new string **str** by concatenation as follows:
  $\texttt{str} = \texttt{pat}[1 \dots m] + \$ + \texttt{txt}[1 \dots n]$.
  Note, $|\texttt{str}| = m + 1 + n$.
- Preprocess Z_i values corresponding to **str**, for $1 < i \leq m + n + 1$.
- For any $i > m + 1$, all $Z_i = m$ identifies an occurrence of **pat**[1 . . . m] at position i in **str**, and hence at position $i - (m + 1)$ in **txt**. That is, $\texttt{pat}[1 \dots m] = (\texttt{str}[i \dots i + m - 1] \equiv \texttt{txt}[i - (m + 1) \dots i - 2])$.
- We already, showed that the computation of Z_i values for any sting **str** takes $O(|\texttt{str}|)$ time.

Using Z-algorithm for linear-time exact pattern matching

Recall the exact pattern matching problem

Given a reference text $\texttt{txt}[1 \ldots n]$ and a pattern $\texttt{pat}[1 \ldots m]$, find **ALL** occurrences, if any, of $\texttt{pat}$ in $\texttt{txt}$.

Realizing a linear-time solution using Gusfield's Z-algorithm/preprocessing

- Construct a new string $\texttt{str}$ by concatenation as follows:
  $\texttt{str} = \texttt{pat}[1 \ldots m] + \$ + \texttt{txt}[1 \ldots n]$.
  Note, $|\texttt{str}| = m + 1 + n$.
- Preprocess Z_i values corresponding to $\texttt{str}$, for $1 < i \leq m + n + 1$.
- For any $i > m + 1$, all $Z_i = m$ identifies an occurrence of $\texttt{pat}$[1 ... m] at position i in $\texttt{str}$, and hence at position $i - (m + 1)$ in $\texttt{txt}$. That is, $\texttt{pat}[1 \ldots m] = (\texttt{str}[i \ldots i + m - 1] \equiv \texttt{txt}[i - (m + 1) \ldots i - 2])$.
- We already, showed that the computation of Z_i values for any sting $\texttt{str}$ takes $O(|\texttt{str}|)$ time.
- Thus, this pattern matching algorithm takes $O(m + n)$ time. **QED**

2. Boyer-Moore Algorithm

Comments:

- Defines the standard benchmark for string pattern matching.
- Many find/search utilities that come packaged within programming languages/operating systems rely on this algorithm.
- GNU's implementation of **grep** is one popular example.

Boyer-Moore Algorithm – Introduction

Boyer-Moore algorithm incorporate three clever ideas:

1. right-to-left scanning

Boyer-Moore Algorithm – Introduction

Boyer-Moore algorithm incorporate three clever ideas:

1. right-to-left scanning
2. bad character shift rule

Boyer-Moore Algorithm – Introduction

Boyer-Moore algorithm incorporate three clever ideas:

1. right-to-left scanning
2. bad character shift rule
3. good suffix shift rule

Boyer-Moore Algorithm – Introduction

Boyer-Moore algorithm incorporate three clever ideas:

1. right-to-left scanning
2. bad character shift rule
3. good suffix shift rule

Boyer-Moore Algorithm – Introduction

Boyer-Moore algorithm incorporate three clever ideas:

1. right-to-left scanning
2. bad character shift rule
3. good suffix shift rule

Caveat!

An additional rule, termed in the field as the **`Galil's optimization`**, ensures linear runtime across all possible scenarios of **`txt`** and **`pat`**.

Boyer-Moore Algorithm – right-to-left scan

For any comparison of **pat**$[1 \ldots m]$ against **txt**$[j \ldots j+m-1]$, the Boyer-Moore algorithm checks/scans for matched characters right to left (instead of the normal left to right scan, as in the naïve algorithm).

Boyer-Moore Algorithm – right-to-left scan

For any comparison of **pat**$[1 \ldots m]$ against **txt**$[j \ldots j+m-1]$, the Boyer-Moore algorithm checks/scans for matched characters right to left (instead of the normal left to right scan, as in the naïve algorithm).

Example: right to left scanning (in some arbitrary iteration)

```
      1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
txt:  x p b c t b x a b p  q  x  c  t  b  p  q

        1 2 3 4 5 6 7
pat:    t p a b x a b
```

Boyer-Moore Algorithm – right-to-left scan

For any comparison of **pat**$[1 \ldots m]$ against **txt**$[j \ldots j+m-1]$, the Boyer-Moore algorithm checks/scans for matched characters right to left (instead of the normal left to right scan, as in the naïve algorithm).

Example: right to left scanning (in some arbitrary iteration)

```
       1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
txt:   x p b c t b x a b p  q  x  c  t  b  p  q

           1 2 3 4 5 6 7
pat:       t p a b x a b
```

Scanning from right to left in the above example:

Boyer-Moore Algorithm – right-to-left scan

For any comparison of **pat**$[1 \dots m]$ against **txt**$[j \dots j+m-1]$, the Boyer-Moore algorithm checks/scans for matched characters right to left (instead of the normal left to right scan, as in the naïve algorithm).

Example: right to left scanning (in some arbitrary iteration)

```
      1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
txt:  x p b c t b x a b p  q  x  c  t  b  p  q

          1 2 3 4 5 6 7
pat:      t p a b x a b
```

Scanning from right to left in the above example:

- Compare **pat**$[7] \equiv$ b with **txt**$[9] \equiv$ b: match.
- Compare **pat**$[6] \equiv$ a with **txt**$[8] \equiv$ a: match.
- Compare **pat**$[5] \equiv$ x with **txt**$[7] \equiv$ x: match.
- Compare **pat**$[4] \equiv$ b with **txt**$[6] \equiv$ b: match.
- Compare **pat**$[3] \equiv$ a with **txt**$[5] \equiv$ t: mismatch.

Boyer-Moore Algorithm – right-to-left scan

For any comparison of **pat**$[1 \ldots m]$ against **txt**$[j \ldots j+m-1]$, the Boyer-Moore algorithm checks/scans for matched characters right to left (instead of the normal left to right scan, as in the naïve algorithm).

Example: right to left scanning (in some arbitrary iteration)

```
      1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
txt:  x p b c t b x a b p  q  x  c  t  b  p  q

          1 2 3 4 5 6 7
pat:      t p a b x a b
```

Scanning from right to left in the above example:

- Compare **pat**$[7] \equiv$ b with **txt**$[9] \equiv$ b: match.
- Compare **pat**$[6] \equiv$ a with **txt**$[8] \equiv$ a: match.
- Compare **pat**$[5] \equiv$ x with **txt**$[7] \equiv$ x: match.
- Compare **pat**$[4] \equiv$ b with **txt**$[6] \equiv$ b: match.
- Compare **pat**$[3] \equiv$ a with **txt**$[5] \equiv$ t: mismatch.

So, after a mismatch during right-to-left scanning , to avoid naïvely shifting **pat** rightwards by **1 position**, BM algorithm employs two additional ideas/tricks discussed below.

Boyer-Moore Algorithm – Bad character shift rule

Example

```
      1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
txt:  x p b c t b x a b p  q  x  c  t  b  p  q

          1 2 3 4 5 6 7
pat:      t p a b x a b
```

- Scanning right-to-left, we found a mismatch comparing **pat**$[3] \equiv$ a with **txt**$[5] \equiv$ t.
- But the rightmost occurrence in the entire **pat** of the mismatched character in **txt** (i.e. **txt**$[5] \equiv$ t) is at position 1 of **pat** (i.e., **pat**$[1] \equiv$ t).

Boyer-Moore Algorithm – Bad character shift rule

Example

```
      1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
txt:  x p b c t b x a b p  q  x  c  t  b  p  q

              1 2 3 4 5 6 7
pat:          t p a b x a b
```

- Scanning right-to-left, we found a mismatch comparing **pat**$[3] \equiv$ a with **txt**$[5] \equiv$ t.
- But the rightmost occurrence in the entire **pat** of the mismatched character in **txt** (i.e. **txt**$[5] \equiv$ t) is at position 1 of **pat** (i.e., **pat**$[1] \equiv$ t).
- So, in this case, one case safely shift **pat** by **two places** to the right so as to match characters **pat**$[1] \equiv$ t and **txt**$[5] \equiv$ t (instead of naïvely shifting by only one place).

Boyer-Moore Algorithm – Formalizing ‘Bad character’ shift rule

- Let **pat**$[1 \ldots m]$ and **txt**$[1 \ldots n]$ be strings from the alphabet $\aleph$.

Boyer-Moore Algorithm – Formalizing 'Bad character' shift rule

- Let **pat**$[1 \ldots m]$ and **txt**$[1 \ldots n]$ be strings from the alphabet $\aleph$.
- Preprocess **pat**, and store for each character $x \in \aleph$, the rightmost position of occurrence of character x in **pat**.

Boyer-Moore Algorithm – Formalizing 'Bad character' shift rule

- Let **pat**$[1 \ldots m]$ and **txt**$[1 \ldots n]$ be strings from the alphabet $\aleph$.
- Preprocess **pat**, and store for each character $x \in \aleph$, the rightmost position of occurrence of character x in **pat**.
  - Call this position, $R(x)$.

Boyer-Moore Algorithm – Formalizing 'Bad character' shift rule

- Let **pat**$[1 \ldots m]$ and **txt**$[1 \ldots n]$ be strings from the alphabet $\aleph$.
- Preprocess **pat**, and store for each character $x \in \aleph$, the rightmost position of occurrence of character x in **pat**.
  - Call this position, $R(x)$.
  - Note, $R(x) = 0$ when x does not occur in **pat**.

Boyer-Moore Algorithm – Formalizing 'Bad character' shift rule

- Let **pat**$[1 \ldots m]$ and **txt**$[1 \ldots n]$ be strings from the alphabet $\aleph$.
- Preprocess **pat**, and store for each character $x \in \aleph$, the rightmost position of occurrence of character x in **pat**.
  - Call this position, $R(x)$.
  - Note, $R(x) = 0$ when x does not occur in **pat**.
- These preprocessed $R(x)$ values will be used (for > 1 position shifts of **pat** under **txt**, were possible) as we will see in the slides to follow.

Boyer-Moore Algorithm – Formalizing ‘Bad character’ shift rule

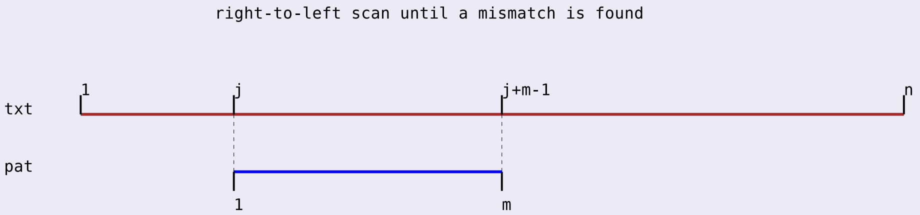
- Let **pat**$[1 \ldots m]$ and **txt**$[1 \ldots n]$ be strings from the alphabet $\aleph$.
- Preprocess **pat**, and store for each character $x \in \aleph$, the rightmost position of occurrence of character x in **pat**.
  - Call this position, $R(x)$.
  - Note, $R(x) = 0$ when x does not occur in **pat**.
- These preprocessed $R(x)$ values will be used (for > 1 position shifts of **pat** under **txt**, were possible) as we will see in the slides to follow.

Boyer-Moore Algorithm – Formalizing 'Bad character' shift rule

- Let **pat**$[1 \ldots m]$ and **txt**$[1 \ldots n]$ be strings from the alphabet $\aleph$.
- Preprocess **pat**, and store for each character $x \in \aleph$, the rightmost position of occurrence of character x in **pat**.
  - Call this position, $R(x)$.
  - Note, $R(x) = 0$ when x does not occur in **pat**.
- These preprocessed $R(x)$ values will be used (for > 1 position shifts of **pat** under **txt**, were possible) as we will see in the slides to follow.

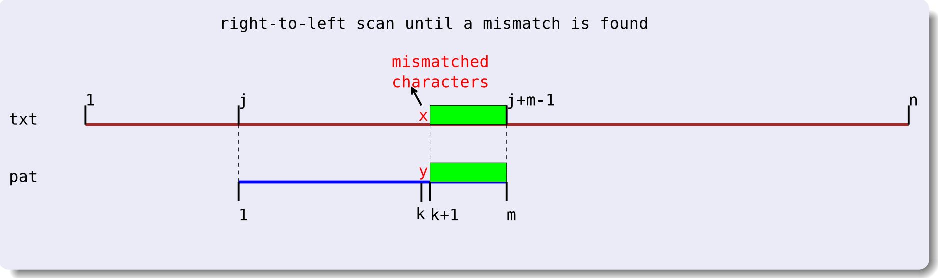
Before that, note, storing $R(x)$ values for **pat** requires at most $O(|\aleph|)$ space, and one lookup per mismatch.

Boyer-Moore Algorithm – Formalizing 'Bad character' shift rule (continued)



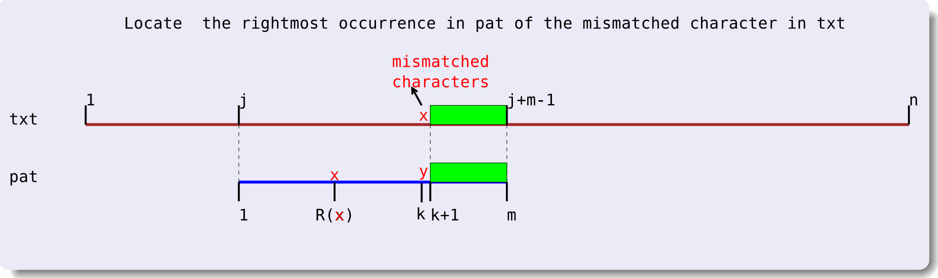
- In some iteration of this algorithm, let's say **txt**$[j \ldots j+m-1]$ and **pat**$[1 \ldots m]$ are being compared via right-to-left scan.

Boyer-Moore Algorithm – Formalizing 'Bad character' shift rule (continued)



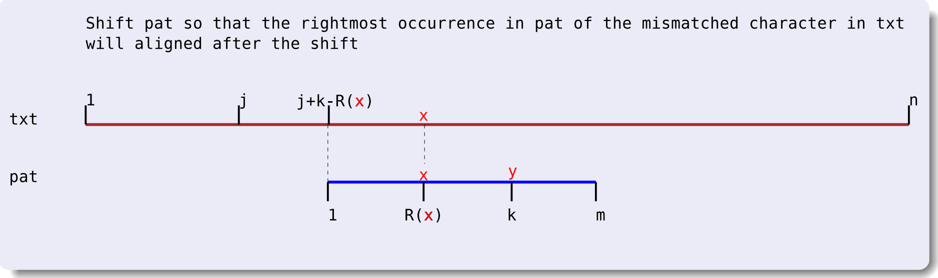
- In some iteration of this algorithm, let's say **txt**$[j \ldots j+m-1]$ and **pat**$[1 \ldots m]$ are being compared via right-to-left scan.
- Let the kth character of **txt** counting from position j, i.e., $\mathbf{x} = \mathbf{txt}[j+k-1]$, be mismatched with the kth character of the pattern $\mathbf{y} = \mathbf{pat}[k]$.

Boyer-Moore Algorithm – Formalizing 'Bad character' shift rule (continued)



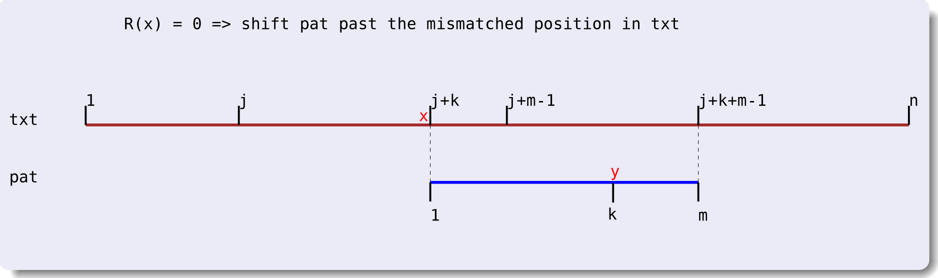
- In some iteration of this algorithm, let's say **txt**$[j \ldots j+m-1]$ and **pat**$[1 \ldots m]$ are being compared via right-to-left scan.
- Let the kth character of **txt** counting from position j, i.e., $\mathbf{x} = \mathbf{txt}[j+k-1]$, be mismatched with the kth character of the pattern $\mathbf{y} = \mathbf{pat}[k]$.
- **Shift/Jump RULE:** Then, the bad-character shift rule asks us to shift rightwards the **pat** along the **txt** by $\max\{1, k - R(\mathbf{x})\}$ **positions**.

Boyer-Moore Algorithm – Formalizing ‘Bad character’ shift rule (continued)



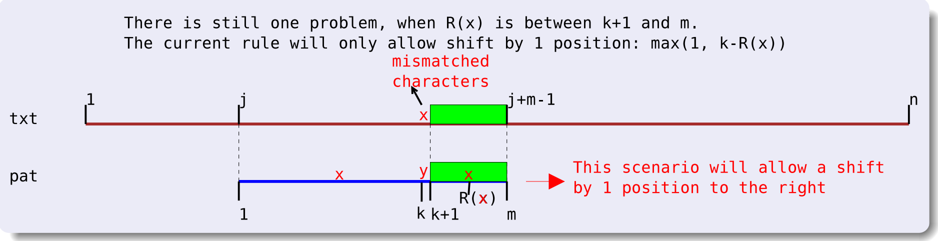
- In some iteration of this algorithm, let's say **txt**$[j \ldots j+m-1]$ and **pat**$[1 \ldots m]$ are being compared via right-to-left scan.
- Let the kth character of **txt** counting from position j, i.e., $\mathbf{x} = \mathbf{txt}[j+k-1]$, be mismatched with the kth character of the pattern $\mathbf{y} = \mathbf{pat}[k]$.
- **Shift/Jump RULE:** Then, the bad-character shift rule asks us to shift rightwards the **pat** along the **txt** by $\max\{1, k - R(\mathbf{x})\}$ **positions**.

Boyer-Moore Algorithm – Formalizing 'Bad character' shift rule (continued)



- In some iteration of this algorithm, let's say **txt**$[j \ldots j+m-1]$ and **pat**$[1 \ldots m]$ are being compared via right-to-left scan.
- Let the kth character of **txt** counting from position j, i.e., $\mathbf{x} = \mathbf{txt}[j+k-1]$, be mismatched with the kth character of the pattern $\mathbf{y} = \mathbf{pat}[k]$.
- **Shift/Jump RULE:** Then, the bad-character shift rule asks us to shift rightwards the **pat** along the **txt** by $\max\{1, k - R(\mathbf{x})\}$ **positions**.
- This implies, if **x** does not occur in **pat**$[1..m]$ $(R(\mathbf{x}) = 0)$, then the entire **pat** can be shifted one position past the point of mismatch in **txt**.

Boyer-Moore Algorithm – Formalizing 'Bad character' shift rule (continued)



- In some iteration of this algorithm, let's say **txt**$[j \ldots j+m-1]$ and **pat**$[1 \ldots m]$ are being compared via right-to-left scan.
- Let the kth character of **txt** counting from position j, i.e., $\mathbf{x} = \mathbf{txt}[j+k-1]$, be mismatched with the kth character of the pattern $\mathbf{y} = \mathbf{pat}[k]$.
- **Shift/Jump RULE:** Then, the bad-character shift rule asks us to shift rightwards the **pat** along the **txt** by $\max\{1, k - R(\mathbf{x})\}$ **positions**.
- This implies, if **x** does not occur in **pat**$[1..m]$ ($R(\mathbf{x}) = 0$), then the entire **pat** can be shifted one position past the point of mismatch in **txt**.
- We have a **problem**! What to do when $R(\mathbf{x}) > k$? **Solution** in next slide.

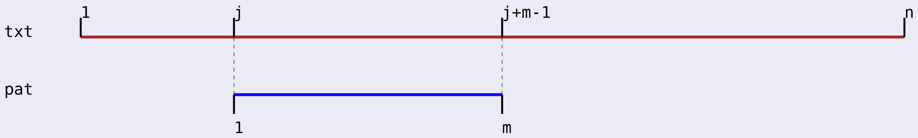
Improving/extending this bad-character (shift) rule

Extended Bad-Character Rule

When a mismatch occurs at some position k in **pat**$[1 \dots m]$, and the corresponding mismatched character is $\mathbf{x} = \mathbf{txt}[j + k - 1]$, then **shift** **pat**$[1..m]$ to the right so that **the closest x in pat that is to the left of position** k is now below the (previously mismatched) **x** in **txt**.

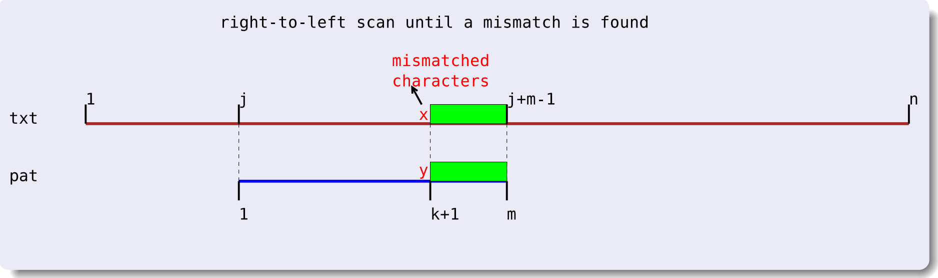
- To achieve this, preprocess **pat**$[1 \dots m]$ so that, for each position $1 \leq k \leq m$ in **pat**, and for each character $x \in \aleph$, the position of the closest occurrence of x to the left of each position k can be efficiently looked up.
- A 2D array (**shift/jump table**) of size $m \times |\aleph|$ can store this information. (Think how this can be implemented more space-efficiently?)

Boyer-Moore Algorithm – Good suffix rule



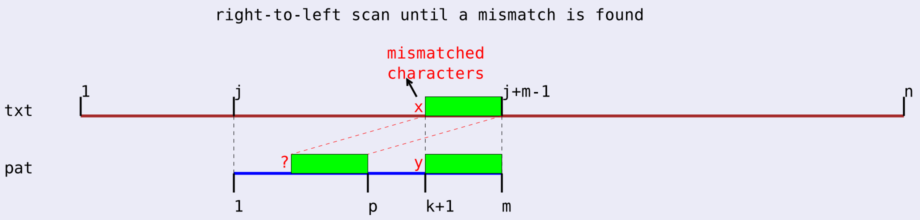
- In some iteration, say **txt**$[j \ldots j+m-1]$ and **pat**$[1 \ldots m]$ are being compared via right-to-left scan.

Boyer-Moore Algorithm – Good suffix rule



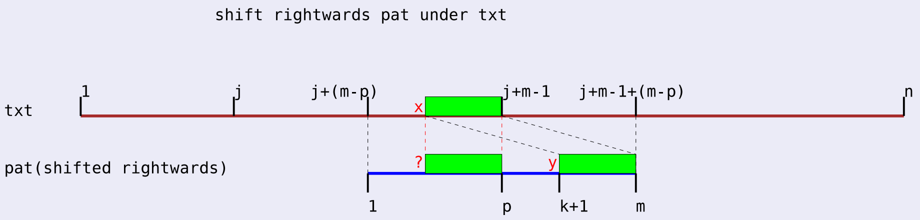
- In some iteration, say **txt**$[j \ldots j+m-1]$ and **pat**$[1 \ldots m]$ are being compared via right-to-left scan.
- Let the kth character of **txt**, i.e., **x** $=$ **txt**$[j+k-1]$, be mismatched with the kth character of the pattern **y** $=$ **pat**$[k]$.

Boyer-Moore Algorithm – Good suffix rule



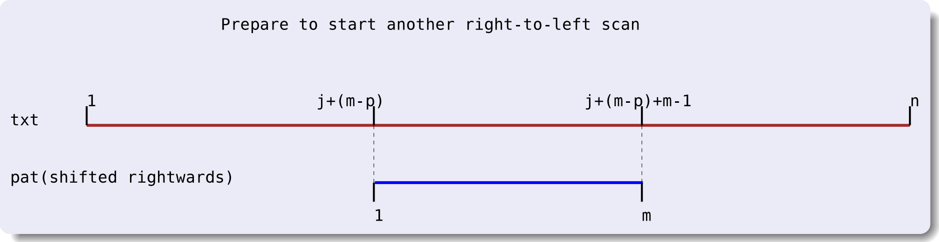
- In some iteration, say **txt**$[j \dots j+m-1]$ and **pat**$[1 \dots m]$ are being compared via right-to-left scan.
- Let the kth character of **txt**, i.e., **x** $=$ **txt**$[j+k-1]$, be mismatched with the kth character of the pattern **y** $=$ **pat**$[k]$.
- If we knew that $p < m$ is the rightmost position in **pat** where the longest substring (of length $>= 1$) ending at position p matches its suffix, that is:
  - **pat**$[p-m+k+1 \dots p] \equiv$ **pat**$[k+1 \dots .m]$.
  - **pat**$[p-m+k] \neq$ **pat**$[k]$.

Boyer-Moore Algorithm – Good suffix rule



- In some iteration, say **txt**$[j \ldots j+m-1]$ and **pat**$[1 \ldots m]$ are being compared via right-to-left scan.
- Let the kth character of **txt**, i.e., $\mathbf{x} = \mathbf{txt}[j+k-1]$, be mismatched with the kth character of the pattern $\mathbf{y} = \mathbf{pat}[k]$.
- If we knew that $p < m$ is the rightmost position in **pat** where the longest substring (of length $>= 1$) ending at position p matches its suffix, that is:
  - $\mathbf{pat}[p-m+k+1 \ldots p] \equiv \mathbf{pat}[k+1 \ldots .m]$.
  - $\mathbf{pat}[p-m+k] \neq \mathbf{pat}[k]$.
- then, **pat** can be safely shifted right by $m-p$ positions,

Boyer-Moore Algorithm – Good suffix rule



- In some iteration, say $\texttt{txt}[j \ldots j+m-1]$ and $\texttt{pat}[1 \ldots m]$ are being compared via right-to-left scan.
- Let the kth character of $\texttt{txt}$, i.e., $\texttt{x} = \texttt{txt}[j+k-1]$, be mismatched with the kth character of the pattern $\texttt{y} = \texttt{pat}[k]$.
- If we knew that $p < m$ is the rightmost position in $\texttt{pat}$ where the longest substring (of length $>= 1$) ending at position p matches its suffix, that is:
  - $\texttt{pat}[p-m+k+1 \ldots p] \equiv \texttt{pat}[k+1 \ldots .m]$.
  - $\texttt{pat}[p-m+k] \neq \texttt{pat}[k]$.
- then, $\texttt{pat}$ can be safely shifted right by $m-p$ positions,
- and a new iteration can be restarted.

Boyer-Moore Algorithm – Ideas to implement the 'good suffix' rule efficiently

Boyer-Moore Algorithm – Ideas to implement the 'good suffix' rule efficiently

To efficiently implement this 'good suffix' rule, we take 'inspiration' from the computation of Z_i values in Gusfield's algorithm (refer slide 13), and define Z_i^{suffix} (specifically on **`pat`**) as follows:

Boyer-Moore Algorithm – Ideas to implement the 'good suffix' rule efficiently

To efficiently implement this 'good suffix' rule, we take 'inspiration' from the computation of Z_i values in Gusfield's algorithm (refer slide 13), and define Z_i^{suffix} (specifically on **pat**) as follows:

Definition of Z_i^{suffix}

Given a **pat**$[1 \dots m]$, define Z_i^{suffix} (for each position $i < m$) as the **length** of the **longest substring ending at position** i of **pat** that matches its suffix (i.e., **pat**[i-Z_i^{suffix}+1 … i] = **pat**[m-Z_i^{suffix}+1 … m]).

Boyer-Moore Algorithm – Ideas to implement the 'good suffix' rule efficiently

To efficiently implement this 'good suffix' rule, we take 'inspiration' from the computation of Z_i values in Gusfield's algorithm (refer slide 13), and define Z_i^{suffix} (specifically on **pat**) as follows:

Definition of Z_i^{suffix}

Given a **pat**$[1 \ldots m]$, define Z_i^{suffix} (for each position $i < m$) as the **length** of the **longest substring ending at position** i of **pat** that matches its suffix (i.e., **pat**[i-Z_i^{suffix}+1 ... i] = **pat**[m-Z_i^{suffix}+1 ... m]).

- Note, computation of Z_i^{suffix} values on **pat** corresponds to the computation of Z_i values on reverse(**pat**).
- Thus, Z_i^{suffix} values can be computed in $O(m)$ time for **pat**$[1 \ldots m]$.

Boyer-Moore Algorithm – Ideas to implement 'good suffix' rule efficiently (continued)

In fact, for each **suffix** starting at position j in **pat**, we want to store the rightmost position p in **pat** such that:

- $\texttt{pat}[j..m] \equiv \texttt{pat}[p - Z_p^{\text{suffix}} + 1 \ldots p]$.
- $\texttt{pat}[j-1] \neq \texttt{pat}[p - Z_p^{\text{suffix}}]$.

Boyer-Moore Algorithm – Ideas to implement 'good suffix' rule efficiently (continued)

In fact, for each **suffix** starting at position j in **pat**, we want to store the rightmost position p in **pat** such that:

- $\texttt{pat}[j..m] \equiv \texttt{pat}[p - Z_p^{\text{suffix}} + 1 \ldots p]$.
- $\texttt{pat}[j-1] \neq \texttt{pat}[p - Z_p^{\text{suffix}}]$.

Store these rightmost positions as $\texttt{goodsuffix}(j) = p$. These can be computed as:

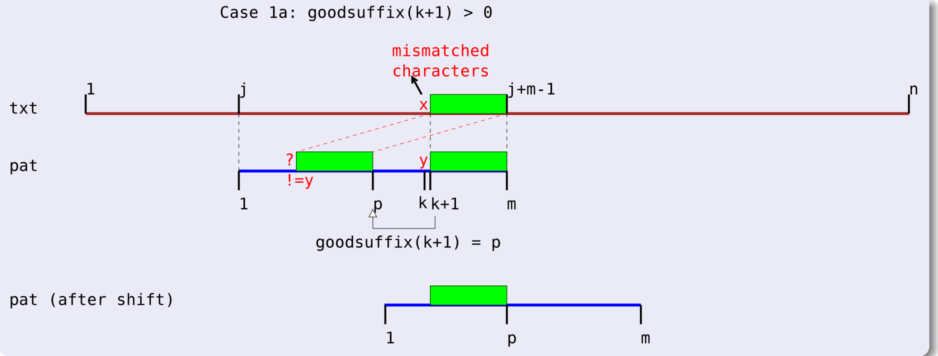
```
m := |pat|
for j from 1 to m+1  do
    goodsuffix(j) := 0
endfor

for p from 1 to m-1  do
    j := m - Z^{suffix}_p + 1
    goodsuffix(j) := p
endfor
```

Boyer-Moore Algorithm – Using ‘good suffix’ rule during search

In any iteration, to use the ‘good suffix’ rule, the following cases have to handled:

Boyer-Moore Algorithm – Using 'good suffix' rule during search

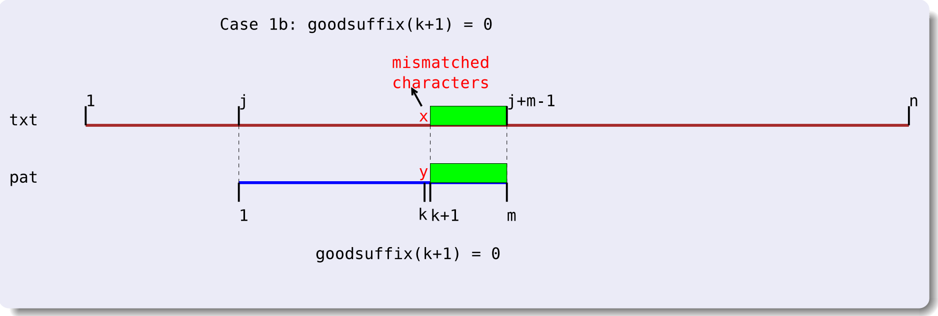
In any iteration, to use the 'good suffix' rule, the following cases have to handled:



Case 1a: if a mismatch occurs at some **pat**$[k]$, and **goodsuffix**$(k+1)>0$ then

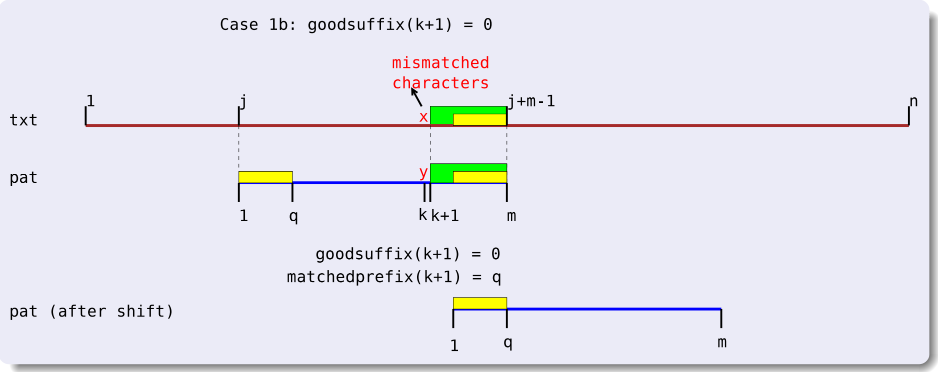
- shift **pat** by $m -$ **goodsuffix**$(k+1)$ places.

Boyer-Moore Algorithm – Using 'good suffix' rule during search (cont'd)



Case 1b: if a mismatch occurs at some **pat**$[k]$, and **goodsuffix**$(k+1)=0$ then

Boyer-Moore Algorithm – Using 'good suffix' rule during search (cont'd)



Case 1b: if a mismatch occurs at some **pat**$[k]$, and **goodsuffix**$(k+1)=0$ then

- shift **pat** by $m -$ **matchedprefix**$(k+1)$ places
  - **matchedprefix**$(k+1)$ denotes the length of the **largest** suffix of **pat**$[k+1..m]$ that is identical to the prefix of **pat**$[1..m-k]$.
  - **matchedprefix**$(.)$ values for **pat** can be precomputed using Z-algorithm in $O(m)$ time – how?

Boyer-Moore Algorithm – Using 'good suffix' rule during search (cont'd)

Case 2: when **pat**$[1 \ldots m]$ fully matches **txt**$[j \ldots j+m-1]$

Boyer-Moore Algorithm – Using 'good suffix' rule during search (cont'd)

Case 2: when $\texttt{pat}[1 \dots m]$ fully matches $\texttt{txt}[j \dots j + m - 1]$

- shift **pat** by $m - \texttt{matchedprefix}(2)$ places. Why?

Boyer-Moore Algorithm – Bringing all pieces together

Preprocessing step

- Preprocess **pat** . . .
  - . . . for jump tables (eg. $R(\cdot)$ values) needed for 'bad-character' shifts (see slides 30-33)
  - . . . for **goodsuffix**$(\cdot)$ and **matchedprefix**$(\cdot)$ values needed for 'good suffix' shifts (see slides 34-39)

Boyer-Moore Algorithm – Bringing all pieces together

Preprocessing step

- Preprocess **pat** ...
  - ...for jump tables (eg. $R(\cdot)$ values) needed for 'bad-character' shifts (see slides 30-33)
  - ...for **goodsuffix**$(\cdot)$ and **matchedprefix**$(\cdot)$ values needed for 'good suffix' shifts (see slides 34-39)

Algorithm

- Starting with **pat**$[1..m]$vs.**txt**$[1..m]$, in each iteration, scan 'right-to-left'

Boyer-Moore Algorithm – Bringing all pieces together

Preprocessing step

- Preprocess **pat** ...
  - ...for jump tables (eg. $R(\cdot)$ values) needed for 'bad-character' shifts (see slides 30-33)
  - ...for **goodsuffix**$(\cdot)$ and **matchedprefix**$(\cdot)$ values needed for 'good suffix' shifts (see slides 34-39)

Algorithm

- Starting with **pat**$[1..m]$vs.**txt**$[1..m]$, in each iteration, scan 'right-to-left'
- Use (extended) bad-character rule to find how many places to the right **pat** should be shifted under **txt**. Call this amount $n_{\text{badcharacter}}$.

Boyer-Moore Algorithm – Bringing all pieces together

Preprocessing step

- Preprocess **pat** ...
  - ...for jump tables (eg. $R(\cdot)$ values) needed for 'bad-character' shifts (see slides 30-33)
  - ...for **goodsuffix**$(\cdot)$ and **matchedprefix**$(\cdot)$ values needed for 'good suffix' shifts (see slides 34-39)

Algorithm

- Starting with **pat**$[1..m]$vs.**txt**$[1..m]$, in each iteration, scan 'right-to-left'
- Use (extended) bad-character rule to find how many places to the right **pat** should be shifted under **txt**. Call this amount $n_{\text{badcharacter}}$.
- Use good suffix rule to find how many places to the right **pat** should be shifted under **txt**. Call this amount $n_{\text{goodsuffix}}$.

Boyer-Moore Algorithm – Bringing all pieces together

Preprocessing step

- Preprocess **pat** ...
  - ...for jump tables (eg. $R(\cdot)$ values) needed for 'bad-character' shifts (see slides 30-33)
  - ...for **goodsuffix**$(\cdot)$ and **matchedprefix**$(\cdot)$ values needed for 'good suffix' shifts (see slides 34-39)

Algorithm

- Starting with **pat**$[1..m]$vs.**txt**$[1..m]$, in each iteration, scan 'right-to-left'
- Use (extended) bad-character rule to find how many places to the right **pat** should be shifted under **txt**. Call this amount $n_{\text{badcharacter}}$.
- Use good suffix rule to find how many places to the right **pat** should be shifted under **txt**. Call this amount $n_{\text{goodsuffix}}$.
- Shift **pat** to the right under **txt** by $\max(n_{\text{badcharacter}}, n_{\text{goodsuffix}})$ places.

Boyer-Moore Algorithm – Galil's optimization to ensure linear runtime always!

- Suppose, in some iteration, we are comparing **pat**$[1 \ldots m]$ with **txt**$[j \ldots j+m-1]$, via right-to-left scanning.
- Say **pat**$[k] \neq$ **txt**$[j+k-1]$ (or even say the entire **pat** matches in **txt**)...
- ...in the next iteration (after applying some appropriate shift)...
- ...if the left end of **pat**[1] lies between **txt**$[j+k-1 \ldots j+m-1]$...
- ...then there is definitely a prefix **pat**$[1 \ldots]$ that **matches** **txt**$[\ldots j+m-1]$, which need not be explicitly compared, after the shift.
- Thus, in the next iteration, the right-to-left scanning can stop prematurely if position **txt**$[j+m-1]$ is reached, to conclude there is an occurrence of **pat** in **txt**.

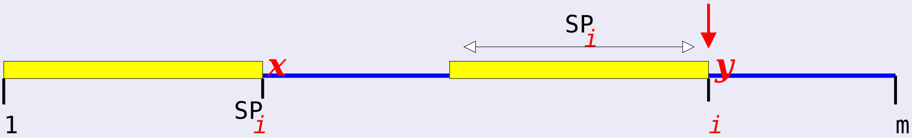
- Employing Galil's optimization during shifting between iterations, the **Boyer Moore algorithm** guarantees *worst-case time-complexity* of $O(m+n)$.

3. Knuth-Morris-Pratt (KMP) Algorithm

Comments:

- KMP is a very popular algorithm covered in undergrad CS courses.
- Although, it is rarely the method of choice..
- . . . and is inferior in performance to Boyer-Moore we discussed above.
- It's explanation is rather simple, as we will see below.

KMP algorithm – Defining SP_i values for $\texttt{pat}[1 \ldots m]$



Definition of SP_i:

Given a pattern $\texttt{pat}[1 \ldots m]$, define SP_i (for each position i in **pat**) as the **length** of the **longest proper suffix** of $\texttt{pat}[1 \ldots i]$ that **matches a prefix** of **pat**, such that $\texttt{pat}[i+1] \neq \texttt{pat}[SP_i+1]$.

Example

$$\texttt{pat} = \overset{1}{b}\ \overset{2}{b}\ \overset{3}{c}\ \overset{4}{c}\ \overset{5}{a}\ \overset{6}{e}\ \overset{7}{b}\ \overset{8}{b}\ \overset{9}{c}\ \overset{10}{a}\ \overset{11}{b}\ \overset{12}{d}$$

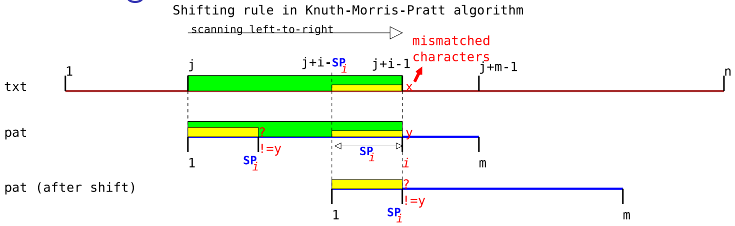
$SP_1 = 0$ $SP_2 = 1$ $SP_3 = 0$ $SP_4 = 0$ $SP_5 = 0$ $SP_6 = 0$

$SP_7 = 0$ $SP_8 = 1$ $SP_9 = 3$ $SP_{10} = 0$ $SP_{11} = 1$ $SP_{12} = 0$

Computing $\mathbf{SP}_i$ values for $\mathbf{pat}[1 \ldots m]$ using Z-algorithm

```
m := |pat|
for i from 1 to m  do
    SP_i := 0
endfor

for j from m down to 2  do
    i := j + Z_j - 1
    SP_i := Z_j
endfor
```

KMP Algorithm overview



- KMP Algorithm is described in terms of the $\mathbf{SP}_i$ values.
- The general procedure/iteration of KMP involves:
  - Compare **pat**$[1 \ldots m]$ against any region of **txt**$[j \ldots j+m-1]$ in the **natural** left-to-right direction (unlike Boyer-Moore).
  - if the first mismatch, while scanning left-to-right, occurs at pos $i+1$: that is, **pat**$[1 \ldots i] \equiv$ **txt**$[j \ldots j+i-1]$
    - Shift **pat** to the right (relative to **txt**) so that ...
    - **pat**$[1 \ldots \mathbf{SP}_i]$ is now aligned with **txt**$[j+i-\mathbf{SP}_i \ldots j+i-1]$
    - **KMP shift rule** In other words, shift **pat** by exactly $i - \mathbf{SP}_i$ places to the right.
  - else, in the case an occurrence of **pat** is found in **txt** (i.e., no mismatch), then shift **pat** by $m - \mathbf{SP}_m$ places.

Lecture Summary

- Naïve algorithm takes $O(m * n)$-time.
- Gusfield's Z algorithm guaranteed in $O(n + m)$-time, worst case
- Boyer-Moore's algorithm takes
  - $O(n + m)$-time worst case ...
  - ... but $O(\frac{n}{m})$-time (sublinear) in most 'realworld' usage.
- Knuth-Morris-Pratt algorithm also takes $O(n + m)$-time worst-case, but inferior in performance to Boyer-Moore in practice.

Next topic(s) ...

Linear-time suffix tree (Ukkonen's algorithm) construction. Revise suffix trees from FIT2004.

-=o0o=-

END

-=o0o=-