Boyer Moore Algorithm

EXAMPLE BASED SUPPLEMENTARY MATERIAL

ANUJA DHARMARATNE

MONASH UNIVERSITY MALAYSIA

Pattern matching- naïve approach

- Assume you are searching for the pattern “bzaa” in the string “aabxaayaab”.

String= aabxaayaabb, Pattern= aabzaa

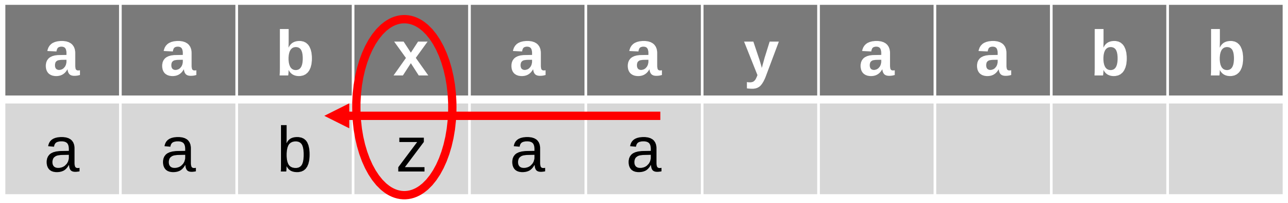
a	a	b	x	a	a	y	a	a	b	b
a	a	b	z	a	a					
	~~a~~	~~a~~	~~b~~	~~z~~	~~a~~	~~a~~				
		~~a~~	~~a~~	~~b~~	~~z~~	~~a~~	~~a~~			
			~~a~~	~~a~~	~~b~~	~~z~~	~~a~~	~~a~~		
				a	a	b	z	a	a	

SKIP

Since x doesn’t occur in the pattern, we can skip the next 3 comparisons.

Boyer Moore Algorithm

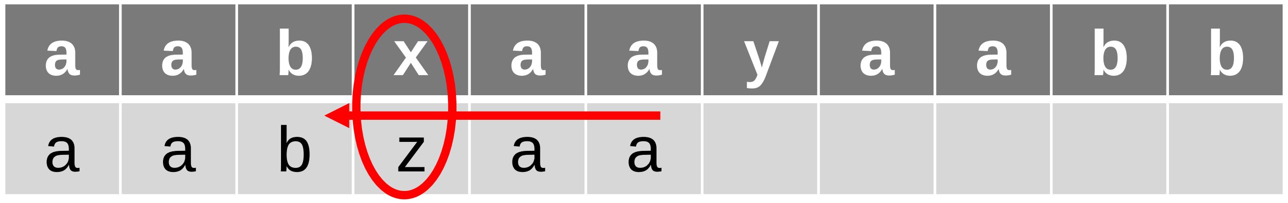
- Skips unnecessary comparisons
- Alignment from left to right
- But characters compared from right to left



a	a	b	x	a	a	y	a	a	b	b
a	a	b	z	a	a					

Boyer Moore Algorithm

- Skips unnecessary comparisons
- Alignment from <u>left to right</u>
- But characters compared from <u>right to left</u>



a	a	b	x	a	a	y	a	a	b	b
a	a	b	z	a	a					

- 2 main rules of Boyer Moore Algorithm:
  - Bad character shift rule
  - Good suffix shift rule

Boyer Moore Algorithm

It uses the outcomes already found on character comparisons to skip future alignments that definitely will not match:

Boyer Moore Algorithm

It uses the outcomes already found on character comparisons to skip future alignments that definitely will not match:

1. **Bad Character shift rule**: When a mismatch is found, use the knowledge of the mismatched letter to skip alignments

Boyer Moore Algorithm

It uses the outcomes already found on character comparisons to skip future alignments that definitely will not match:

1. **Bad Character shift rule**: When a mismatch is found, use the knowledge of the mismatched letter to skip alignments

2. **Good suffix shift rule**: If there are more than one matching characters, use knowledge of those matched characters (substring) to skip alignments

Boyer Moore Algorithm

It uses the outcomes already found on character comparisons to skip future alignments that definitely will not match:

1. **Bad Character shift rule**: When a mismatch is found, use the knowledge of the mismatched letter to skip alignments

2. **Good suffix shift rule**: If there are more than one matching characters, use knowledge of those matched characters (substring) to skip alignments

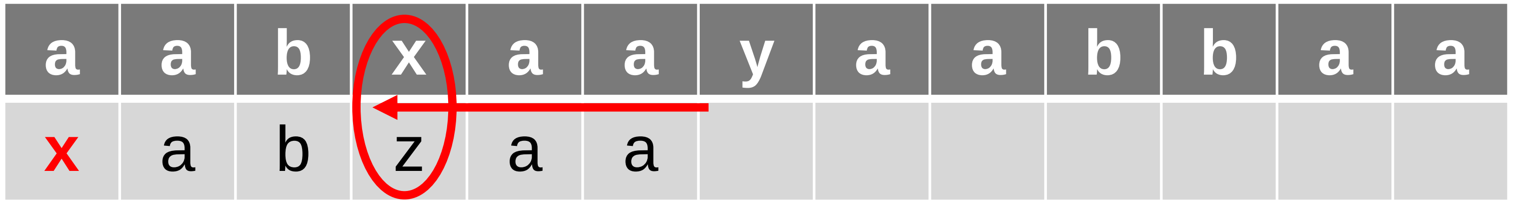
3. Alignments are done in one direction (i.e. left to right) and characters are compared in the opposite direction (i.e. right to left) **for longer skips**

1. Bad Character rule

When a mismatch happens, skip until:

a). the mismatch becomes a match, **OR**

b). the pattern moves past the mismatched character



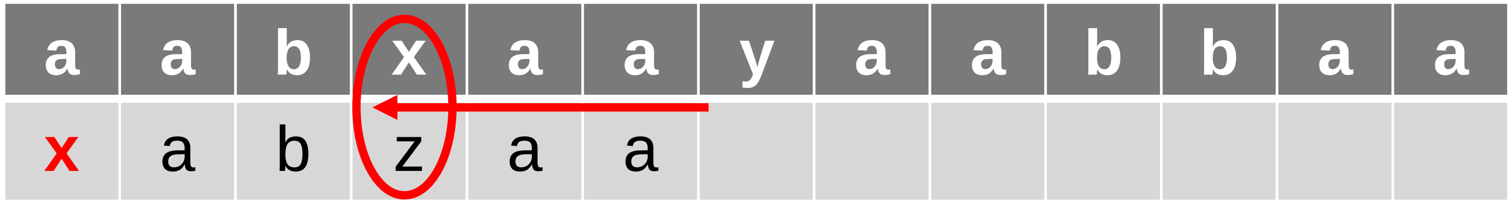
An x is found in the pattern. So shift the pattern until they match (case a)).

1. Bad Character rule

When a mismatch happens, skip until:

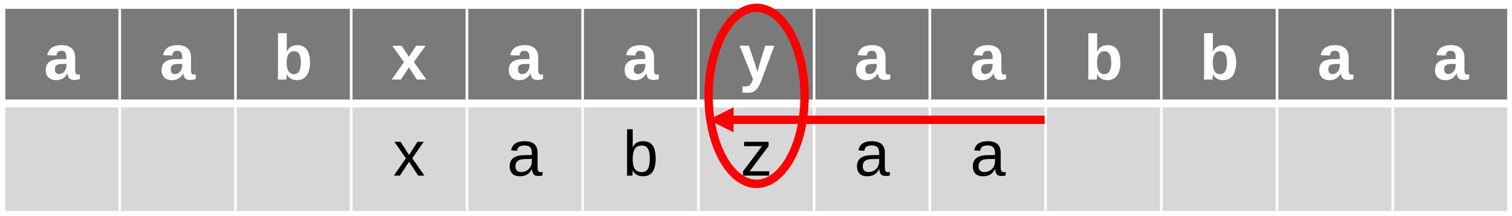
a). the mismatch becomes a match, **OR**

b). the pattern moves past the mismatched character



a	a	b	x	a	a	y	a	a	b	b	a	a
x	a	b	z	a	a							

An x is found in the pattern. So shift the pattern until they match (case a)).



a	a	b	x	a	a	y	a	a	b	b	a	a
			x	a	b	z	a	a				

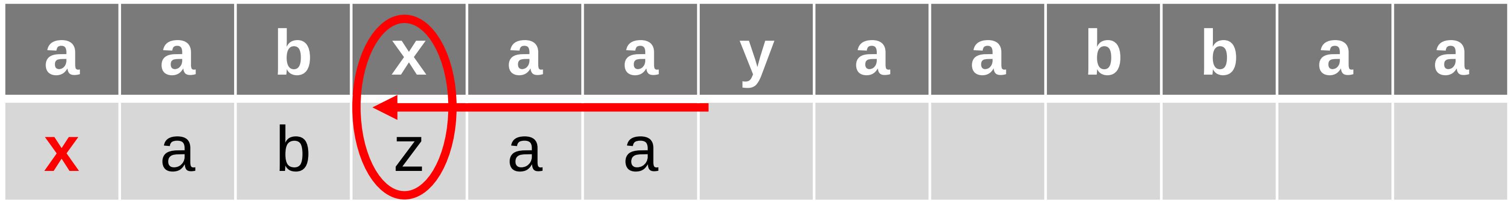
Search for a letter "y" in the pattern. Is any there? NO. Thus shift until the mismatched character is past (case b)).

1. Bad Character rule

When a mismatch happens, skip until:

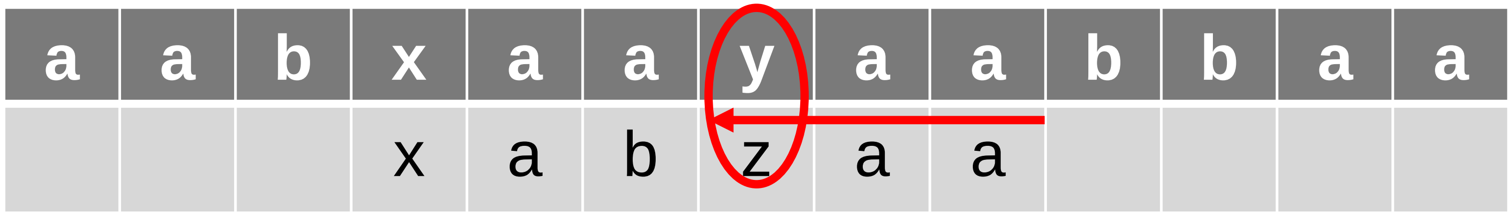
a). the mismatch becomes a match, **OR**

b). the pattern moves past the mismatched character



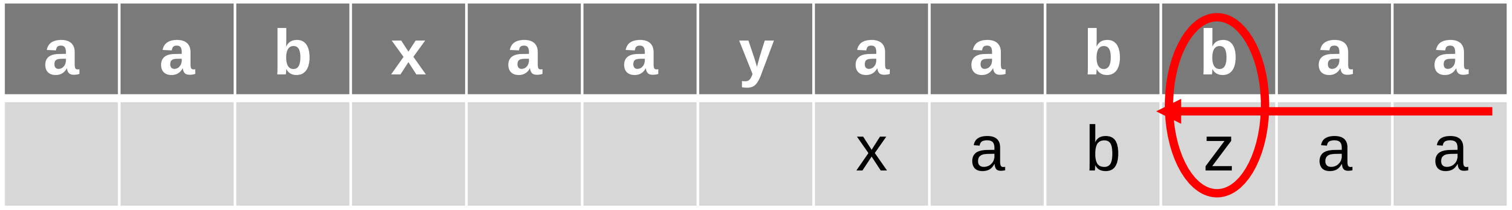
a	a	b	x	a	a	y	a	a	b	b	a	a
x	a	b	z	a	a							

An x is found in the pattern. So shift the pattern until they match (case a)).



a	a	b	x	a	a	y	a	a	b	b	a	a
			x	a	b	z	a	a				

Search for a letter “y” in the pattern. Is any there? NO. Thus shift until the mismatched character is past (case b)).



a	a	b	x	a	a	y	a	a	b	b	a	a
							x	a	b	z	a	a

Bad character rule- preprocessing

- Bad character rule needs to know the position of the matching character- this needs some preprocessing.

Bad character rule- preprocessing

- Bad character rule needs to know the position of the matching character- this needs some preprocessing.
- In preprocessing step, the rightmost position of occurrence of each character x in the pattern needs to be found and stored.

Bad character rule- preprocessing

- Bad character rule needs to know the position of the matching character- this needs some preprocessing.
- In preprocessing step, the rightmost position of occurrence of each character x in the pattern needs to be found and stored.
- Call this position, R(x).
- R(x) = 0 when x does not occur in the pattern.

Bad character rule- preprocessing

- Bad character rule needs to know the position of the matching character- this needs some preprocessing.
- In preprocessing step, the rightmost position of occurrence of each character x in the pattern needs to be found and stored.
- Call this position, R(x).
- R(x) = 0 when x does not occur in the pattern.
- <u>**Example**</u>: P= “potato”

```
                  1 2 3 4 5 6
```

  R(‘a’)= 4, R(‘o’)= 6, R(‘p’)= 1, R(‘t’)= 5.
- Let’s see how we can use these R values calculated in the preprocessing step in our algorithm.

Preprocessing contd

Bad character rule- preprocessing

- Assume that at some point, the kth character of the pattern mismatches with the kth character of the string.

Bad character rule- preprocessing

- Assume that at some point, the kth character of the pattern mismatches with the kth character of the string.
- Then, the bad character rule asks us to shift the pattern rightwards along the string by max(1, k- R(x)) positions.(x is the mismatching character in the string)

Bad character rule- preprocessing

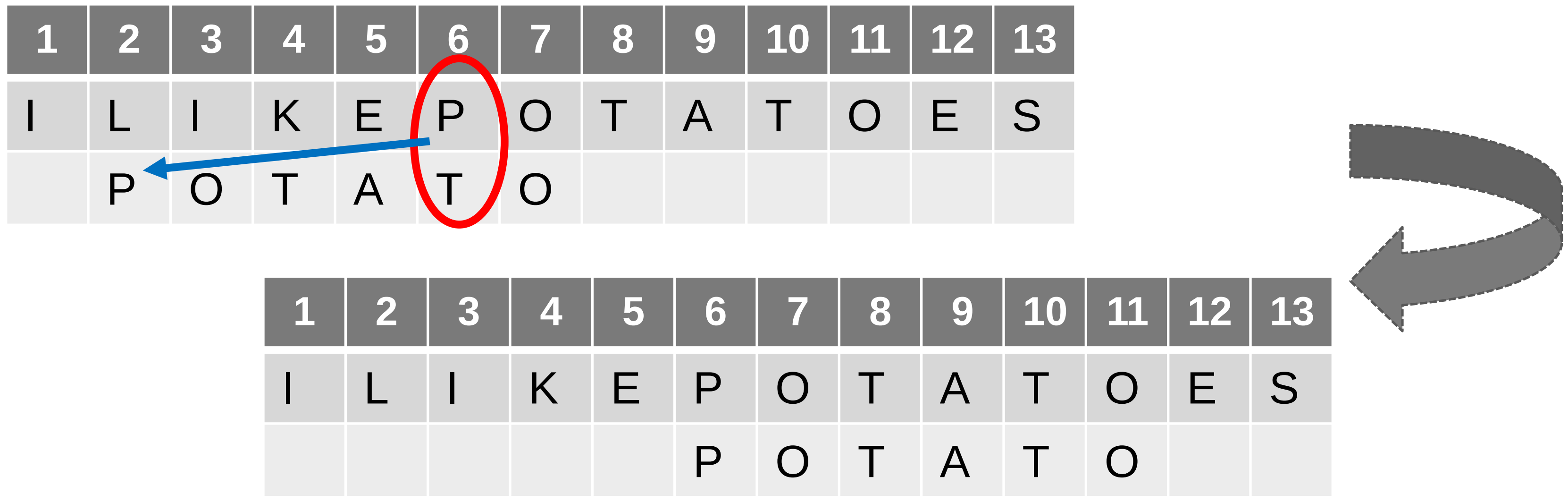
- Assume that at some point, the kth character of the pattern mismatches with the kth character of the string.
- Then, the bad character rule asks us to shift the pattern rightwards along the string by max(1, k- R(x)) positions.(x is the mismatching character in the string)

1	2	3	4	5	6	7	8	9	10	11	12	13
I	L	I	K	E	P	O	T	A	T	O	E	S
	P	O	T	A	T	O						

  - In this example, mismatching (bad) character is P.
  - R(P)= 2, k= 6, thus max(1, k-R(P))= 4 ➔ shift by 4 characters

1	2	3	4	5	6	7	8	9	10	11	12	13
I	L	I	K	E	P	O	T	A	T	O	E	S
					P	O	T	A	T	O		

Extended Bad character rule

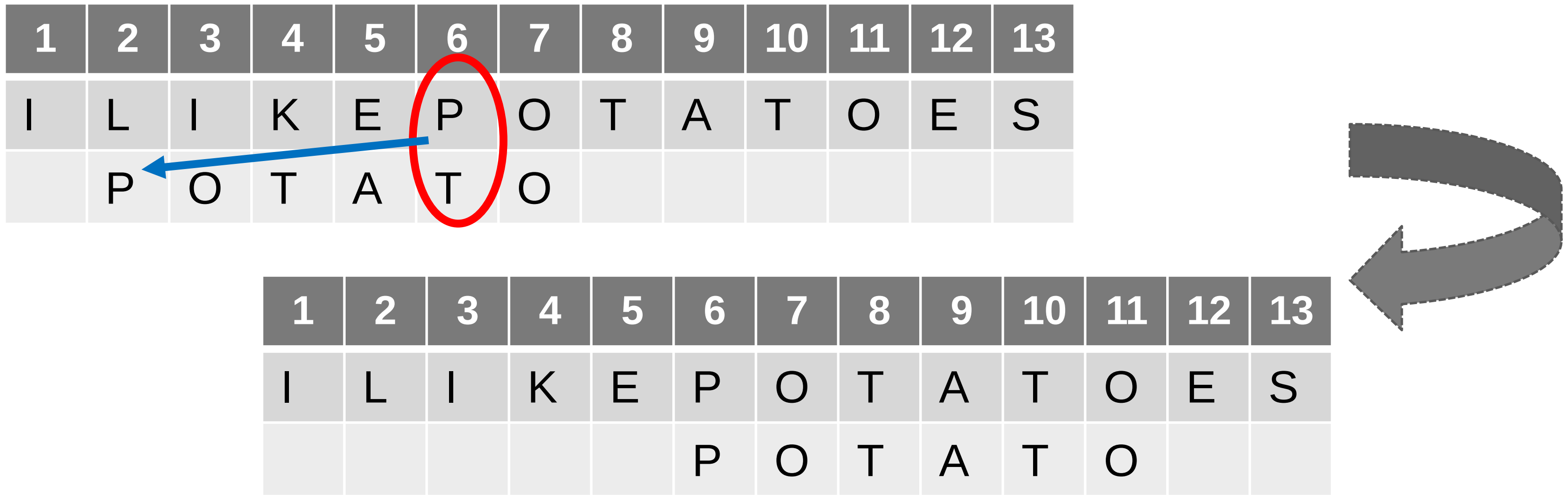


1	2	3	4	5	6	7	8	9	10	11	12	13
I	L	I	K	E	P	O	T	A	T	O	E	S
	P	O	T	A	T	O						

1	2	3	4	5	6	7	8	9	10	11	12	13
I	L	I	K	E	P	O	T	A	T	O	E	S
					P	O	T	A	T	O		

- What does this mean? Look at the above example.
  - When such a mismatch occurs, then shift the pattern to the right so that the closest P in the pattern that is to the left of position k (in this case k=6) is now below the (previously mismatched) P in the string.

Extended Bad character rule



1	2	3	4	5	6	7	8	9	10	11	12	13
I	L	I	K	E	P	O	T	A	T	O	E	S
	P	O	T	A	T	O						

1	2	3	4	5	6	7	8	9	10	11	12	13
I	L	I	K	E	P	O	T	A	T	O	E	S
					P	O	T	A	T	O		

- What does this mean? Look at the above example.
  - When such a mismatch occurs, then shift the pattern to the right so that the closest P in the pattern that is to the left of position k (in this case k=6) is now below the (previously mismatched) P in the string.
- How can you achieve this?
  - Preprocess the pattern so that for each position k in the pattern and for each character x, the position of the closest occurrence of x to the left of each position k can be efficiently looked up. **(THINK!)**

Quiz

What if the bad character does not occur in the pattern? i.e. the mismatching character (say, x) is not present in the pattern, which means R(x)=0.

Quiz- Solution

What if the bad character does not occur in the pattern? i.e. the mismatching character (say, x) is not present in the pattern, which means R(x)=0.

The entire pattern can be shifted one position past the point of mismatch in the string.

In this example, we don't have letter M in the pattern.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
I	L	O	V	E	T	O	M	A	T	O	E	S	A	L	O	T
					P	O	T	A	T	O						

Therefore, we shift the whole pattern until it past the M.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
I	L	O	V	E	T	O	M	A	T	O	E	S	A	L	O	T
								P	O	T	A	T	O			

2. Good Suffix rule

If a matching substring ***t*** between the string and the pattern exists, skip when a mismatch happens, until:

a). there are no mismatches between pattern & ***t***, **OR**

b). a prefix of the pattern matches a suffix of ***t***, **OR**

c). the pattern moves past ***t*, whichever happens first.**

2. Good Suffix rule

If a matching substring ***t*** between the string and the pattern exists, skip when a mismatch happens, until:

a). there are no mismatches between pattern & ***t***, **OR**

b). a prefix of the pattern matches a suffix of ***t***, **OR**

c). the pattern moves past ***t*****,** **whichever happens first.**

a	**a**	**a**	**x**	**a**	**a**	**y**	**a**	**a**	**b**	**b**	**a**	**a**
x	a	a	z	a	a							

Now, we are going to check whether there are any matches between the pattern and substring ***t*****. Can you find any? YES**

2. Good Suffix rule

If a matching substring ***t*** between the string and the pattern exists, skip when a mismatch happens, until:

a). there are no mismatches between pattern & ***t***, **OR**

b). a prefix of the pattern matches a suffix of ***t***, **OR**

c). the pattern moves past ***t*, whichever happens first.**

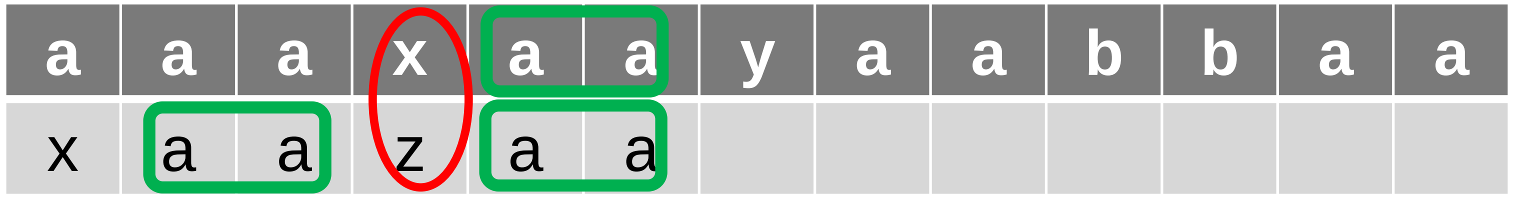
a	**a**	**a**	**x**	**a**	**a**	**y**	**a**	**a**	**b**	**b**	**a**	**a**
x	a	a	z	a	a							

Now, we are going to check whether there are any matches between the pattern and substring ***t*. Can you find any? YES**

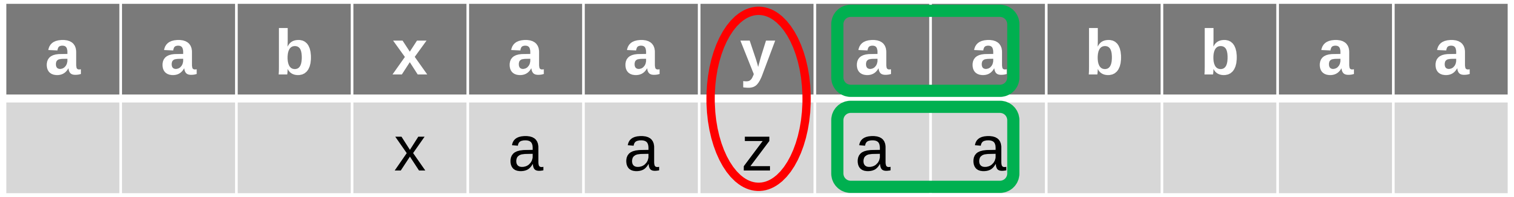
a	**a**	**a**	**x**	**a**	**a**	**y**	**a**	**a**	**b**	**b**	**a**	**a**
x	a	a	z	a	a							

Now, we can skip until the first ‘aa’ overlaps the second.

2. Good Suffix rule- contd..



a	a	a	x	a	a	y	a	a	b	b	a	a
x	a	a	z	a	a							

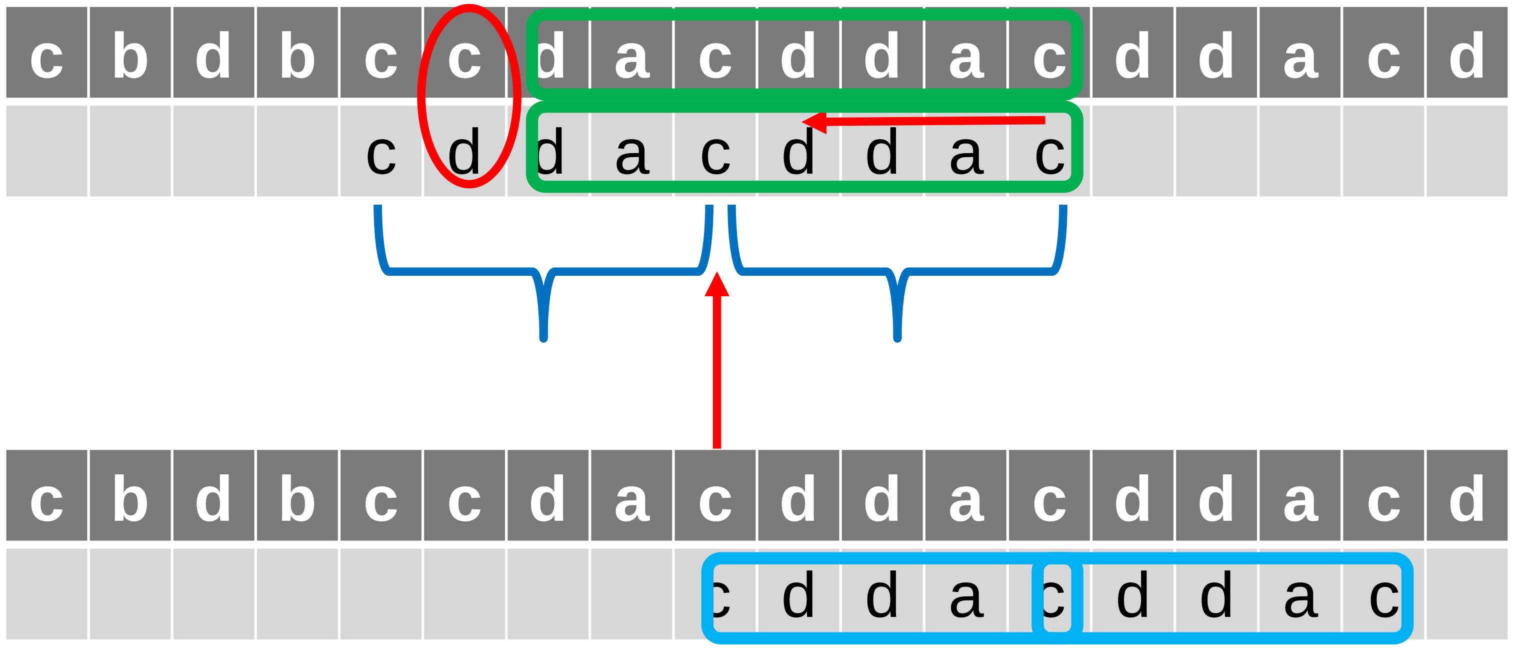


a	a	b	x	a	a	y	a	a	b	b	a	a
			x	a	a	z	a	a				

This shows us the case a).
i.e. skip when a mismatch happens, until:
there are no mismatches between the pattern & ***t***

2. Good Suffix rule- contd..

Look at the following example for case b).
i.e. skip when a mismatch happens, until:
a prefix of the pattern matches a suffix of ***t***


c b d b c c d a c d d a c d d a c d
c d d a c d d a c
c b d b c c d a c d d a c d d a c d
c d d a c d d a c

2. Good Suffix rule- contd..

Now, the last case c), which is much simpler.
i.e. skip when a mismatch happens, until:
the pattern moves past ***t***

c	b	d	b	c	c	d	a	c	d	d	a	c
				b	d	d	a					

c	b	d	b	c	c	d	a	c	d	d	a	c
								b	d	d	a	

Efficiently implementing Good Suffix Rule

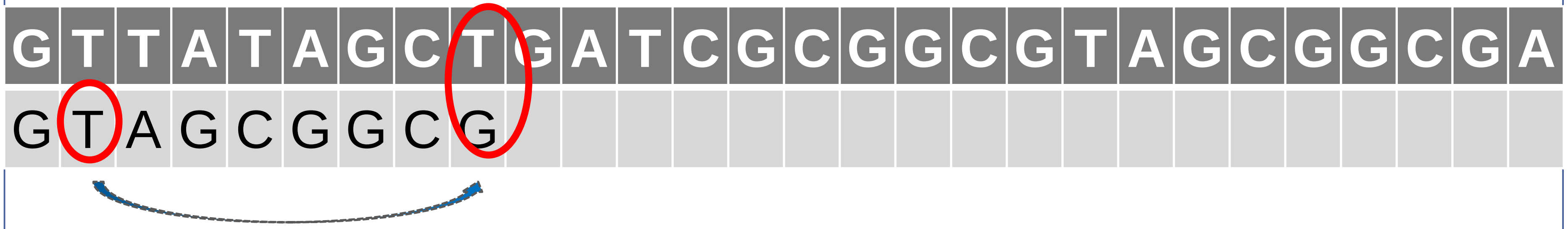
- Use Gusfield's Z-algorithm
  - Given a pattern of length 1…m, define a suffix , Z_i^{suffix} (for each position i < m) as the length of the longest substring ending at position i of the pattern that matches its suffix.
- Computation of Z_i^{suffix} values on the pattern is just the same as computation of Z_i values on the reversed pattern.
- Therefore, Z_i^{suffix} values can be computed in O(m) time.

Putting both rules together

Use either the bad character rule or the good suffix rule, whichever skips more number of characters.

Putting both rules together

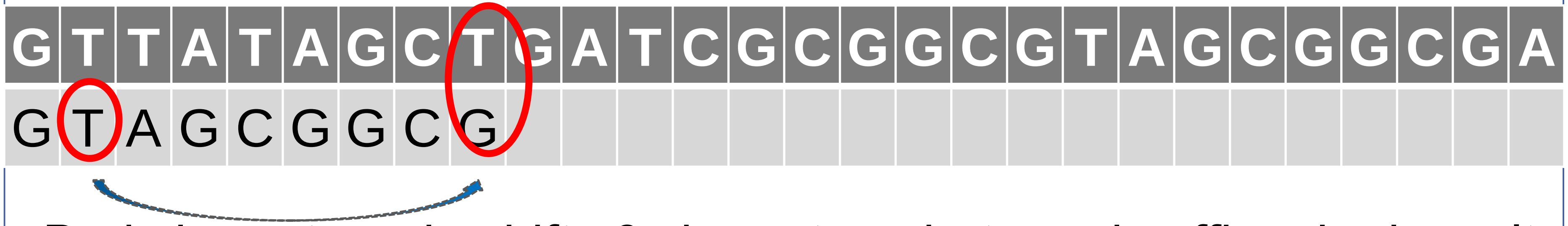
Use either the bad character rule or the good suffix rule, whichever <u>skips more </u>number of characters.



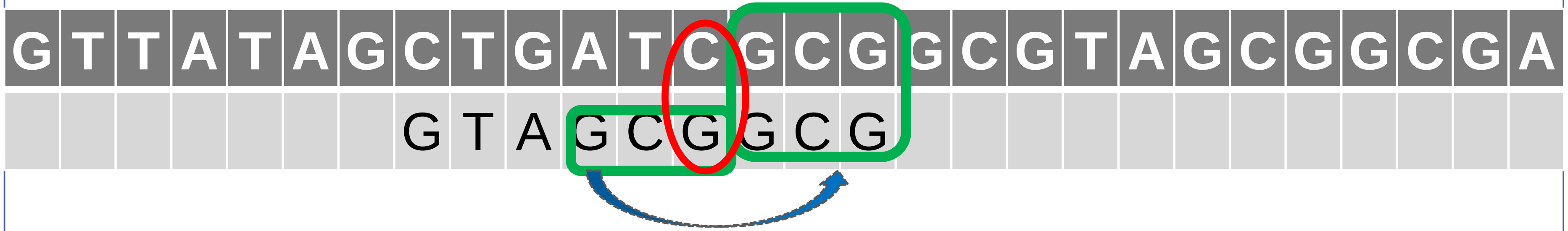
Bad character rule shifts 6 characters, but good suffix rule doesn't apply, thus 0 shifts. Therefore, **apply bad character rule.**

Putting both rules together

Use either the bad character rule or the good suffix rule, whichever skips more number of characters.

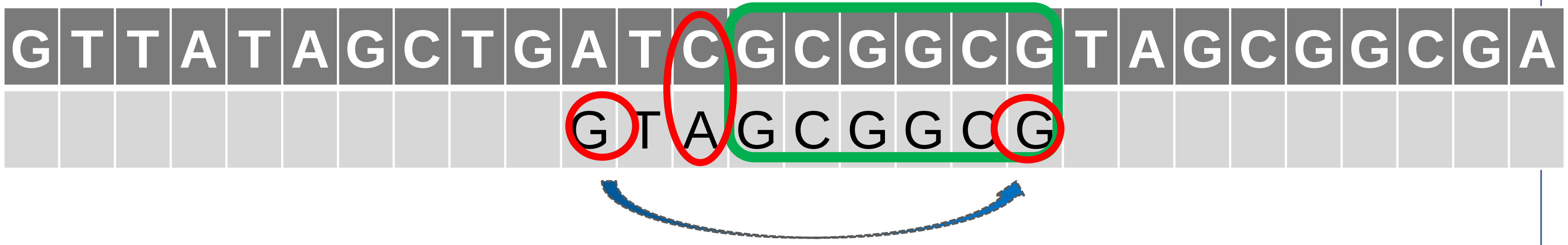


Bad character rule shifts 6 characters, but good suffix rule doesn't apply, thus 0 shifts. Therefore, **apply bad character rule**.

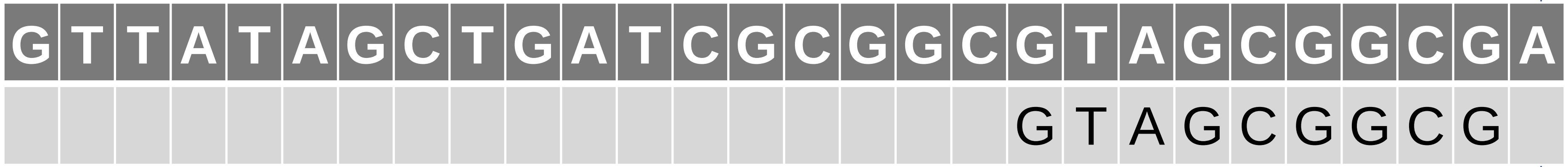


Bad character rule doesn't apply, but good suffix rule can shift 3 characters. Therefore, **apply Good Suffix rule**.

Putting both rules together- contd…



No matching character “C” found in the pattern. Thus shift until the matching component is past. i.e. good suffix rule is applied since it shifts more characters (7) rather than just 2 shifts by bad character rule.



Boyer Moore Algorithm- summary

- In each iteration, scan right-to-left from 1st character of the string (TXT) and pattern (PAT).

Boyer Moore Algorithm- summary

- In each iteration, scan right-to-left from 1st character of the string (TXT) and pattern (PAT).
- Use the (extended) **bad-character rule** to find how many places to the right PAT should be shifted under TXT. Call this amount $\boldsymbol{n_{BC}}$.

Boyer Moore Algorithm- summary

- In each iteration, scan right-to-left from 1st character of the string (TXT) and pattern (PAT).
- Use the (extended) **bad-character rule** to find how many places to the right PAT should be shifted under TXT. Call this amount $\boldsymbol{n_{BC}}$.
- Use the **good suffix rule** to find how many places to the right PAT should be shifted under TXT. Call this amount $\boldsymbol{n_{GS}}$.

Boyer Moore Algorithm- summary

- In each iteration, scan right-to-left from 1st character of the string (TXT) and pattern (PAT).
- Use the (extended) **bad-character rule** to find how many places to the right PAT should be shifted under TXT. Call this amount $\boldsymbol{n_{BC}}$.
- Use the **good suffix rule** to find how many places to the right PAT should be shifted under TXT. Call this amount $\boldsymbol{n_{GS}}$.
- Shift PAT to the right under TXT by max($\boldsymbol{n_{BC}}$, $\boldsymbol{n_{GS}}$) places.
- The Boyer Moore algorithm has the worst-case time-complexity of ***O(m + n).***