Algorithmic Challenges: Knuth-Morris-Pratt Algorithm

Michael Levin

Higher School of Economics

Algorithms on Strings
Data Structures and Algorithms

Outline

1. Exact Pattern Matching
2. Safe Shift
3. Prefix Function
4. Computing Prefix Function
5. Knuth-Morris-Pratt Algorithm

Exact Pattern Matching

Input: Strings T (Text) and P (Pattern).

Output: All such positions in T (Text) where P (Pattern) appears as a substring.

(For all strings in this module we use 0-based indices)

Brute Force Algorithm

- Slide the Pattern down Text

Brute Force Algorithm

- Slide the Pattern down Text
- Running time $\Theta(|T||P|)$

Brute Force Algorithm

a	b	r	a	c	a	d	a	b	r	a

a	b	r	a

Brute Force Algorithm

0	1	2	3	4	5	6	7	8	9	10
a	b	r	a	c	a	d	a	b	r	a

a	b	r	a

Output: []

Brute Force Algorithm

0	1	2	3	4	5	6	7	8	9	10
a	b	r	a	c	a	d	a	b	r	a
a	b	r	a							

Output: [0]

Brute Force Algorithm

0	1	2	3	4	5	6	7	8	9	10
a	b	r	a	c	a	d	a	b	r	a
	a	b	r	a						

Output: [0]

Brute Force Algorithm

0	1	2	3	4	5	6	7	8	9	10
a	b	r	a	c	a	d	a	b	r	a
	a	b	r	a						

Output: [0]

Brute Force Algorithm

0	1	2	3	4	5	6	7	8	9	10
a	b	r	a	c	a	d	a	b	r	a
		a	b	r	a					

Output: [0]

Brute Force Algorithm

0	1	2	3	4	5	6	7	8	9	10
a	b	r	a	c	a	d	a	b	r	a
		a	b	r	a					

Output: [0]

Brute Force Algorithm

0	1	2	3	4	5	6	7	8	9	10
a	b	r	a	c	a	d	a	b	r	a
			a	b	r	a				

Output: [0]

Brute Force Algorithm

0	1	2	3	4	5	6	7	8	9	10
a	b	r	a	c	a	d	a	b	r	a
			a	b	r	a				

Output: [0]

Brute Force Algorithm

0	1	2	3	4	5	6	7	8	9	10
a	b	r	a	c	a	d	a	b	r	a
				a	b	r	a			

Output: [0]

Brute Force Algorithm

0	1	2	3	4	5	6	7	8	9	10
a	b	r	a	c	a	d	a	b	r	a
				a	b	r	a			

Output: [0]

Brute Force Algorithm

0	1	2	3	4	5	6	7	8	9	10
a	b	r	a	c	a	d	a	b	r	a
					a	b	r	a		

Output: [0]

Brute Force Algorithm

0	1	2	3	4	5	6	7	8	9	10
a	b	r	a	c	a	d	a	b	r	a

a	b	r	a

Output: [0]

Brute Force Algorithm

0	1	2	3	4	5	6	7	8	9	10
a	b	r	a	c	a	d	a	b	r	a
						a	b	r	a	

Output: [0]

Brute Force Algorithm

0	1	2	3	4	5	6	7	8	9	10
a	b	r	a	c	a	d	a	b	r	a
						a	b	r	a	

Output: [0]

Brute Force Algorithm

0	1	2	3	4	5	6	7	8	9	10
a	b	r	a	c	a	d	a	b	r	a
							a	b	r	a

Output: [0]

Brute Force Algorithm

0	1	2	3	4	5	6	7	8	9	10
a	b	r	a	c	a	d	a	b	r	a
							a	b	r	a

Output: [0,7]

Skipping Positions

a	b	r	a	c	a	d	a	b	r	a

a	b	r	a

Skipping Positions

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

Skipping Positions

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

Skipping Positions

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

Skipping Positions

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

Skipping Positions

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

Skipping Positions

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

Skipping Positions

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

Skipping Positions

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

Skipping Positions

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

Skipping Positions

a	b	c	d	a	b	c	d	a	b	e	f

a	b	c	d	a	b	e	f

Skipping Positions

a	b	c	d	a	b	c	d	a	b	e	f

a	b	c	d	a	b	e	f

Skipping Positions

a	b	c	d	a	b	c	d	a	b	e	f

a	b	c	d	a	b	e	f

Skipping Positions

a	b	c	d	a	b	c	d	a	b	e	f
				a	b	c	d	a	b	e	f

Skipping Positions

a	b	c	d	a	b	c	d	a	b	e	f
				a	b	c	d	a	b	e	f

Skipping Positions

a	b	a	b	a	b	a	b	a	b	e	f

a	b	a	b	a	b	e	f

Skipping Positions

a	b	a	b	a	b	a	b	a	b	e	f

a	b	a	b	a	b	e	f

Skipping Positions

a	b	a	b	a	b	a	b	a	b	e	f

a	b	a	b	a	b	e	f

Skipping Positions

a	b	a	b	a	b	a	b	a	b	e	f

a	b	a	b	a	b	e	f

Skipping Positions

a	b	a	b	a	b	a	b	a	b	e	f

a	b	a	b	a	b	e	f

Skipping Positions

a	b	a	b	a	b	a	b	a	b	e	f

a	b	a	b	a	b	e	f

Skipping Positions

a	b	a	b	a	b	a	b	a	b	e	f

				a	b	a	b	a	b	e	f

Skipping Positions

a	b	a	b	a	b	a	b	a	b	e	f
				a	b	a	b	a	b	e	f

Definitions

Definition

Border of string S is a prefix of S which is equal to a suffix of S, but not equal to the whole S.

Example

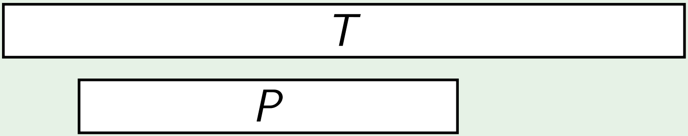
"a" is a border of "arba"

"ab" is a border of "abcdab"

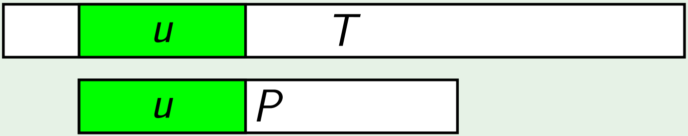
"abab" is a border of "ababab"

"ab" is not a border of "ab" because border should not coincide with the whole string

Shifting Pattern

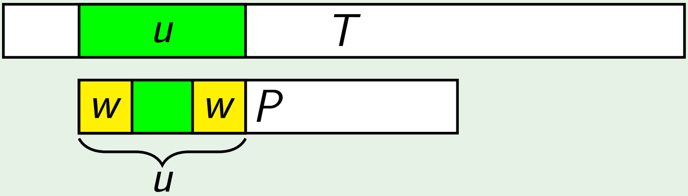


Shifting Pattern



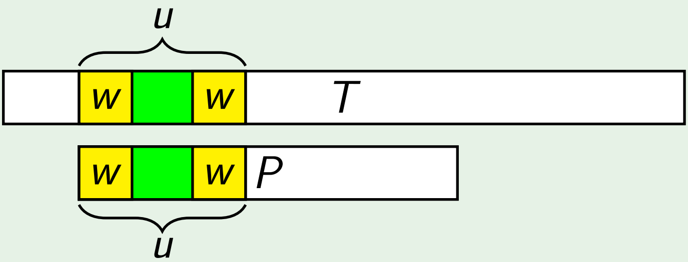
- Find longest common prefix u

Shifting Pattern



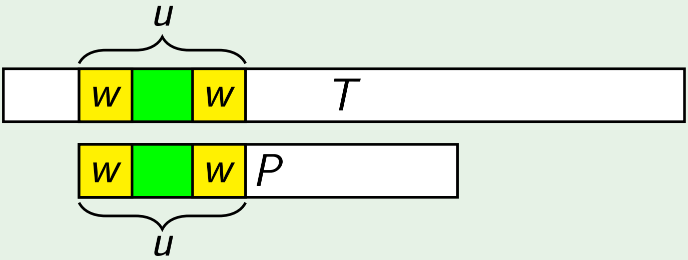
- Find longest common prefix u
- Find w — the longest border of u

Shifting Pattern



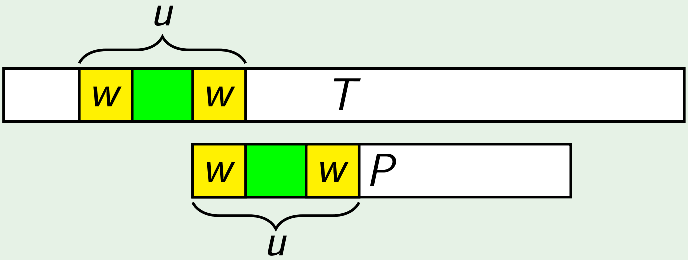
- Find longest common prefix u
- Find w — the longest border of u

Shifting Pattern



- Find longest common prefix u
- Find w — the longest border of u
- Move P such that prefix w in P aligns with suffix w of u in T

Shifting Pattern



- Find longest common prefix u
- Find w — the longest border of u
- Move P such that prefix w in P aligns with suffix w of u in T

- Now you know we can skip some of the comparisons

- Now you know we can skip some of the comparisons
- But we shouldn't miss any of the pattern occurrences in the text

- Now you know we can skip some of the comparisons
- But we shouldn't miss any of the pattern occurrences in the text
- Is it safe to shift the pattern this way?

Outline

1. Exact Pattern Matching
2. Safe Shift
3. Prefix Function
4. Computing Prefix Function
5. Knuth-Morris-Pratt Algorithm

Suffix notation

Definition

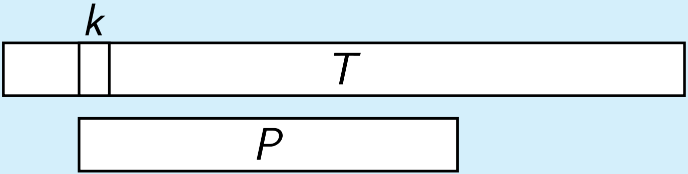
Denote by S_k suffix of string S starting at position k.

Examples

S = "abcd" $\Rightarrow S_2$ = "cd"
T = "abc" $\Rightarrow T_0$ = "abc"
P = "aa" $\Rightarrow P_1$ = "a"

Safe shift

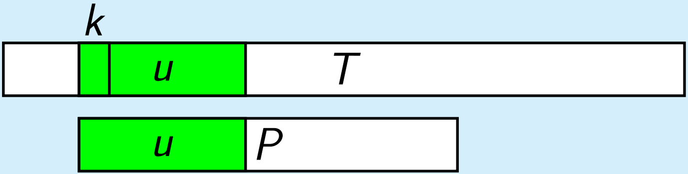
Lemma



Let u be the longest common prefix of P and T_k. Let w be the longest border of u. Then there are no occurrences of P in T starting between positions k and $(k + |u| - |w|)$ — the start of suffix w in the prefix u of T_k.

Safe shift

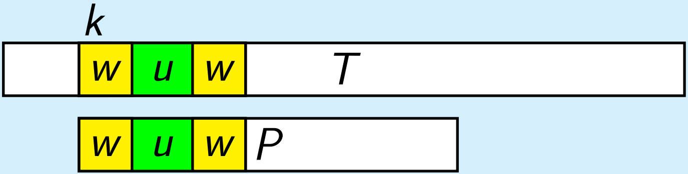
Lemma



Let u be the longest common prefix of P and T_k. Let w be the longest border of u. Then there are no occurrences of P in T starting between positions k and $(k + |u| - |w|)$ — the start of suffix w in the prefix u of T_k.

Safe shift

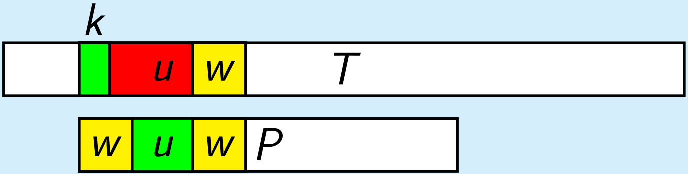
Lemma



Let u be the longest common prefix of P and T_k. Let w be the longest border of u. Then there are no occurrences of P in T starting between positions k and $(k + |u| - |w|)$ — the start of suffix w in the prefix u of T_k.

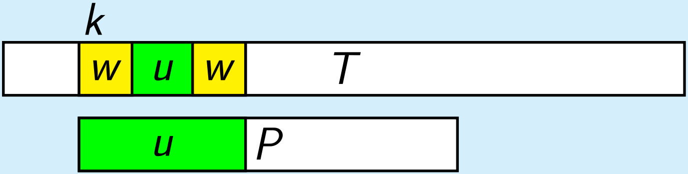
Safe shift

Lemma

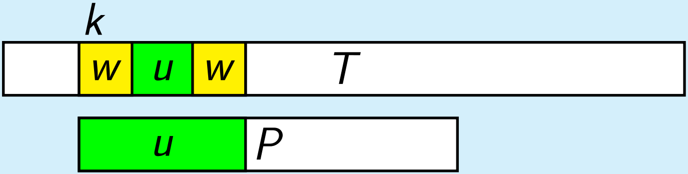


Let u be the longest common prefix of P and T_k. Let w be the longest border of u. Then there are no occurrences of P in T starting between positions k and $(k + |u| - |w|)$ — the start of suffix w in the prefix u of T_k.

Proof

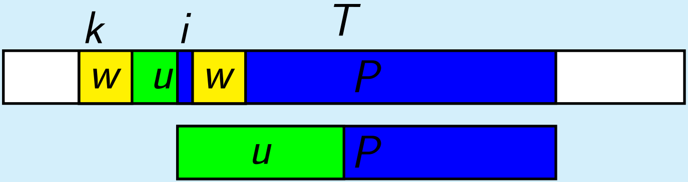


Proof



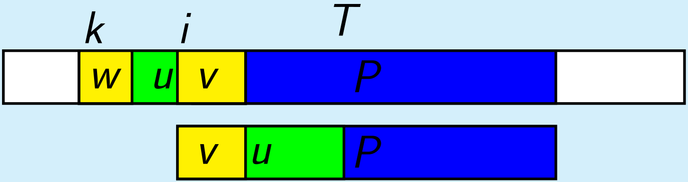
- Suppose P occurs in T in position i between k and start of suffix w

Proof



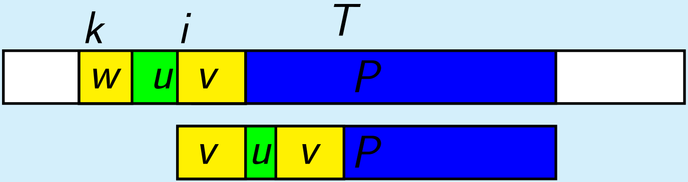
- Suppose P occurs in T in position i between k and start of suffix w

Proof



- Suppose P occurs in T in position i between k and start of suffix w
- Then there is prefix v of P equal to suffix in u, and v is longer than w $\square$

Proof



- Then there is prefix v of P equal to suffix in u, and v is longer than w
- v is a border longer than w, but w is longest border of u — contradiction □

- Now you know it is possible to avoid many of the comparisons which Brute Force algorithm does

- Now you know it is possible to avoid many of the comparisons which Brute Force algorithm does
- But how to determine the best pattern shifts?

Outline

1. Exact Pattern Matching
2. Safe Shift
3. Prefix Function
4. Computing Prefix Function
5. Knuth-Morris-Pratt Algorithm

Prefix function

Definition

Prefix function of a string P is a function $s(i)$ that for each i returns the length of the longest border of the prefix $P[0..i]$.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1	1	2

Prefix function

Definition

Prefix function of a string P is a function $s(i)$ that for each i returns the length of the longest border of the prefix $P[0..i]$.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1	1	2

Prefix function

Definition

Prefix function of a string P is a function $s(i)$ that for each i returns the length of the longest border of the prefix $P[0..i]$.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1	1	2

Prefix function

Definition

Prefix function of a string P is a function $s(i)$ that for each i returns the length of the longest border of the prefix $P[0..i]$.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1	1	2

Prefix function

Definition

Prefix function of a string P is a function $s(i)$ that for each i returns the length of the longest border of the prefix $P[0..i]$.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1	1	2

Prefix function

Definition

Prefix function of a string P is a function $s(i)$ that for each i returns the length of the longest border of the prefix $P[0..i]$.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1	1	2

Prefix function

Definition

Prefix function of a string P is a function $s(i)$ that for each i returns the length of the longest border of the prefix $P[0..i]$.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1	1	2

Prefix function

Definition

Prefix function of a string P is a function $s(i)$ that for each i returns the length of the longest border of the prefix $P[0..i]$.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1	1	2

Prefix function

Definition

Prefix function of a string P is a function $s(i)$ that for each i returns the length of the longest border of the prefix $P[0..i]$.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1	1	2

Prefix function

Definition

Prefix function of a string P is a function $s(i)$ that for each i returns the length of the longest border of the prefix $P[0..i]$.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1	1	2

Prefix function

Definition

Prefix function of a string P is a function $s(i)$ that for each i returns the length of the longest border of the prefix $P[0..i]$.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1	1	2

Lemma

$P[0..i]$ has a border of length $s(i+1) - 1$

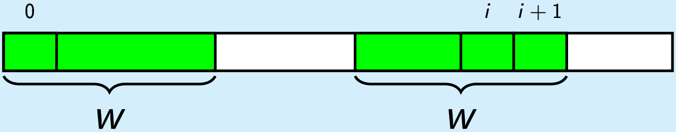
Proof

Only when the character at i+1 is equal to the next character after the longest border

Lemma

$P[0..i]$ has a border of length $s(i+1) - 1$

Proof

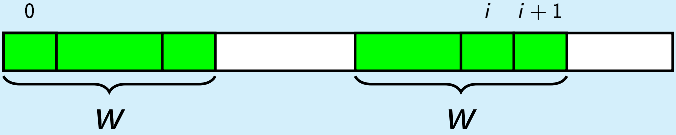


- Take the longest border w of $P[0..i+1]$

Lemma

$P[0..i]$ has a border of length $s(i+1)-1$

Proof

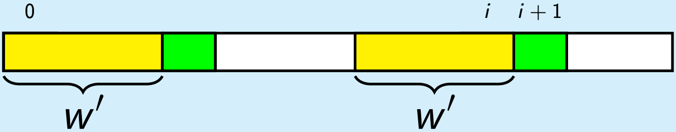


- Take the longest border w of $P[0..i+1]$
- Cut the last character from w — it's a border of $P[0..i]$ now $\square$

Lemma

$P[0..i]$ has a border of length $s(i+1)-1$

Proof



- Take the longest border w of $P[0..i+1]$
- Cut the last character from w — it's a border of $P[0..i]$ now $\square$

Corollary

$$s(i+1) \leq s(i) + 1$$

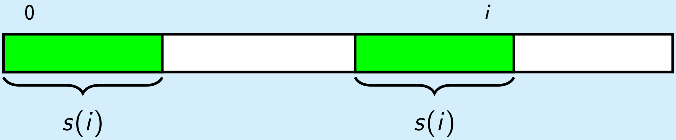
Prefix function cannot grow very fast it will be just one more, equal or less than previous prefix function

Enumerating borders

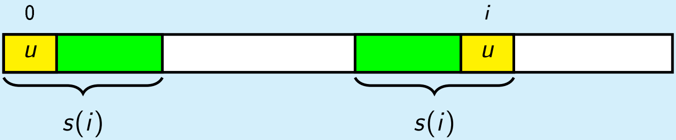
Lemma

If $s(i) > 0$, then all borders of $P[0..i]$ but for the longest one are also borders of $P[0..s(i) - 1]$.

Proof

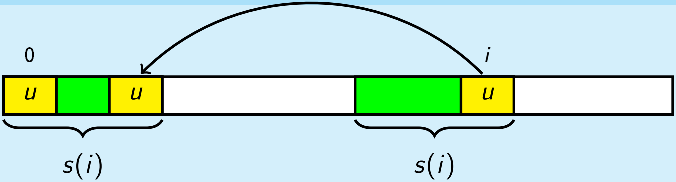


Proof



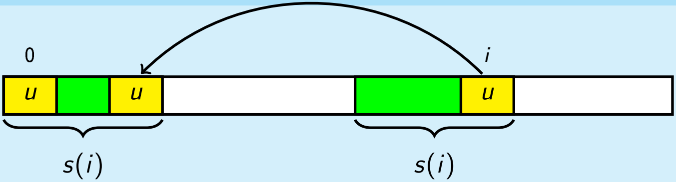
- Let u be a border of $P[0..i]$ such that $|u| < s(i)$

Proof



- Let u be a border of $P[0..i]$ such that $|u| < s(i)$
- Then u is both a prefix and a suffix of $P[0..s(i)-1]$

Proof



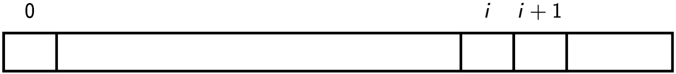
- Let u be a border of $P[0..i]$ such that $|u| < s(i)$
- Then u is both a prefix and a suffix of $P[0..s(i) - 1]$
- $u \neq P[0..s(i) - 1]$, so u is a border of $P[0..s(i) - 1]$

Enumerating borders

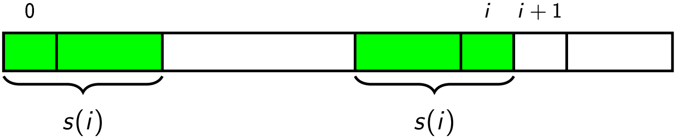
Corollary

All borders of $P[0..i]$ can be enumerated by taking the longest border b_1 of $P[0..i]$, then the longest border b_2 of b_1, then the longest border b_3 of b_2, $\ldots$, and so on.

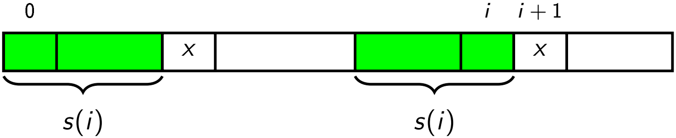
Computing $s(i+1)$



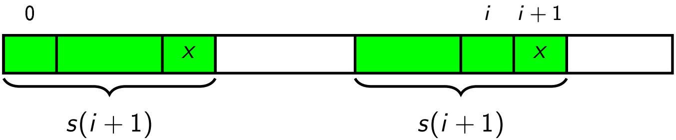
Computing $s(i+1)$



Computing $s(i+1)$



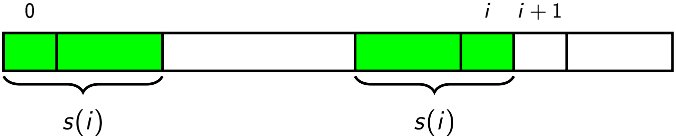
Computing $s(i+1)$



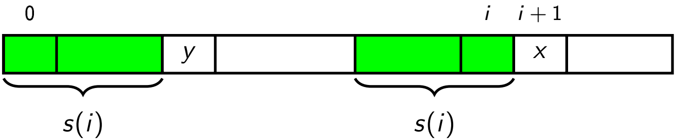
$$s(i+1) = s(i) + 1$$

Only when the character at i+1 is equal to the next character after the longest border

Computing $s(i+1)$

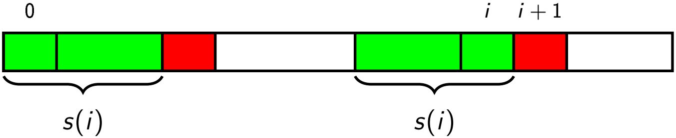


Computing $s(i+1)$

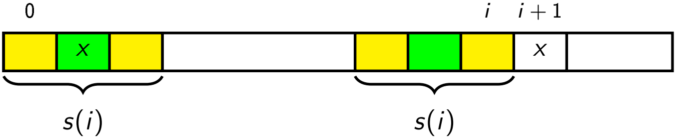


when the character at i+1 is not equal to the next character after the longest border then we need to scan left from i till we find the character similar to i+1

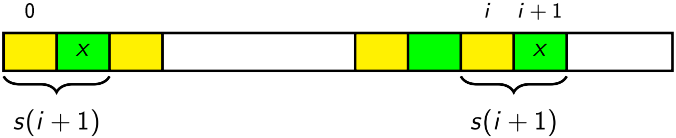
Computing $s(i+1)$



Computing $s(i+1)$



Computing $s(i+1)$



$$s(i+1) = |some\ border\ of\ P[0..s(i)-1]| + 1$$

- Now you know lots of properties of prefix function

- Now you know lots of properties of prefix function
- But how to compute all of its values??

Outline

1. Exact Pattern Matching
2. Safe Shift
3. Prefix Function
4. Computing Prefix Function
5. Knuth-Morris-Pratt Algorithm

Example

P	a	b	a	b	a	b	c	a	a	b
s										

Example

P	a	b	a	b	a	b	c	a	a	b
s										

Example

P	a	b	a	b	a	b	c	a	a	b
s	0									

We'll start with position 0,and we'll fill in 0. And S of 0 is always 0,because the prefix of pattern ending in position 0 has length 1,and has no known empty borders.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0									

Example

P	a	b	a	b	a	b	c	a	a	b
s	0									

Example

P	a	b	a	b	a	b	c	a	a	b
s	0									

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0								

Why don't we move on to the characterin position one, which is b. And we compare it with the next characterafter the end of the previous border. The end of the previous border wasbefore the pattern P starts, so the next character is a. And we compare a with b,they are different. And so the value of the prefixfunction is again zero, because we cannot takeborder of the empty border. And so we have to acknowledge thatthe prefix function is again zero.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0								

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0								

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1							

Now we'll look at the next character a and we'll look at the next character after the end of the previous border which is again the a in position 0. And we see that these characters are the same. So we increase the previous value of s by one. And our value of prefix function for position 2 is 1, and now our current border is of length 1 and it contains just the letter a in position 0.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1							

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1							

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2						

And we look at the next character which is b. And we need to compare it with the character right after the end of the current border,which is the b in position one. And those characters are the same, so we increase the length of the previous border by one. And we write down that s of three is equal to two. And our current border is of length two,and it is ab.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2						

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2						

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3					

Now we'll look at the next character a. We need to compare it with the next character after the current border, which is a in position two. And they're the same. So again, increase the value of our prefix function by a and we increase the length of the current boarder and it becomes aba.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3					

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3					

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4				

The next character is b, we need to compare it with the next character after the end of our border and they are again, the same. So, we increase the value of our prefix function by one and it becomes four. And our border is abab.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4				

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4				

Now we'll look at the character c, we need to compare it with the next character after the end of the current border. And they are different.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4				

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4				

So what we need to do is to take the longest border of our current border, and we'll look at position three,and s of 3 is two. And so the longest border of abab is just ab. And now,we need to compare our current character c with the next character after the end of that border, which is a. And they're again different

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4				

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4				

so we need to take the longest border of our current border, and that has length 0. So, that will be empty string. And so now,we need to compare our current character with the first character of the pattern,which is a. And they're again different. So we'll write down that our prefix function is 0.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0			

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0			

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0			

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1		

We move on to the next character a. We compare it with the next character after the end of the border which is the first character of pattern. And they are the same so we write down1 as the value of our prefix function. Now the border has length1 it is just string a.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1		

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1		

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1		

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1		

We look at the next character a, and we compare it with the character right after the end of the current border. They're different, so we need to take the longest border of our current border, which is empty string. And so we need to compare current character with the first character of the pattern. They're the same, so we write down 1 as the value of our prefix function and the current border has length 1.

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1	1	

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1	1	

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1	1	

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1	1	2

Example

P	a	b	a	b	a	b	c	a	a	b
s	0	0	1	2	3	4	0	1	1	2

And finally, we go to the symbol b in the end of the pattern and we need to compare it with b in position 1. They are the same so increase the value of the prefix function by 1 and write down 2 as the value of the prefix function for the last position.

ComputePrefixFunction(P)

```
s ← array of integers of length |P|
s[0] ← 0, border ← 0
for i from 1 to |P| − 1:
  while (border > 0) and (P[i] ≠ P[border]):
    border ← s[border − 1]
  if P[i] == P[border]:
    border ← border + 1
  else:
    border ← 0
  s[i] ← border
return s
```

ComputePrefixFunction(P)

```
s ← array of integers of length |P|
s[0] ← 0, border ← 0
for i from 1 to |P| − 1:
   while (border > 0) and (P[i] ≠ P[border]):
      border ← s[border − 1]
   if P[i] == P[border]:
      border ← border + 1
   else:
      border ← 0
   s[i] ← border
return s
```

ComputePrefixFunction(P)

```
s ← array of integers of length |P|
s[0] ← 0, border ← 0
for i from 1 to |P| − 1:
  while (border > 0) and (P[i] ≠ P[border]):
    border ← s[border − 1]
  if P[i] == P[border]:
    border ← border + 1
  else:
    border ← 0
  s[i] ← border
return s
```

ComputePrefixFunction(P)

```
s ← array of integers of length |P|
s[0] ← 0, border ← 0
for i from 1 to |P| − 1:
  while (border > 0) and (P[i] ≠ P[border]):
    border ← s[border − 1]
  if P[i] == P[border]:
    border ← border + 1
  else:
    border ← 0
  s[i] ← border
return s
```

ComputePrefixFunction(P)

```
s ← array of integers of length |P|
s[0] ← 0, border ← 0
for i from 1 to |P| − 1:
    while (border > 0) and (P[i] ≠ P[border]):
        border ← s[border − 1]
    if P[i] == P[border]:
        border ← border + 1
    else:
        border ← 0
    s[i] ← border
return s
```

ComputePrefixFunction(P)

```
s ← array of integers of length |P|
s[0] ← 0, border ← 0
for i from 1 to |P| − 1:
    while (border > 0) and (P[i] ≠ P[border]):
        border ← s[border − 1]
    if P[i] == P[border]:
        border ← border + 1
    else:
        border ← 0
    s[i] ← border
return s
```

ComputePrefixFunction(P)

```
s ← array of integers of length |P|
s[0] ← 0, border ← 0
for i from 1 to |P| − 1:
    while (border > 0) and (P[i] ≠ P[border]):
        border ← s[border − 1]
    if P[i] == P[border]:
        border ← border + 1
    else:
        border ← 0
    s[i] ← border
return s
```

ComputePrefixFunction(P)

```
s ← array of integers of length |P|
s[0] ← 0, border ← 0
for i from 1 to |P| − 1:
   while (border > 0) and (P[i] ≠ P[border]):
      border ← s[border − 1]
   if P[i] == P[border]:
      border ← border + 1
   else:
      border ← 0
   s[i] ← border
return s
```

ComputePrefixFunction(P)

```
s ← array of integers of length |P|
s[0] ← 0, border ← 0
for i from 1 to |P| − 1:
  while (border > 0) and (P[i] ≠ P[border]):
    border ← s[border − 1]
  if P[i] == P[border]:
    border ← border + 1
  else:
    border ← 0
  s[i] ← border
return s
```

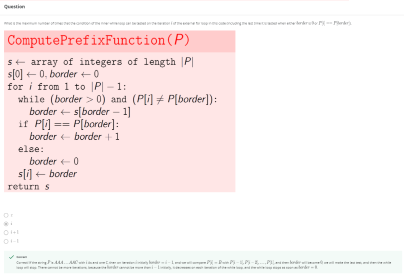
ComputePrefixFunction(P)

```
s ← array of integers of length |P|
s[0] ← 0, border ← 0
for i from 1 to |P| − 1:
  while (border > 0) and (P[i] ≠ P[border]):
    border ← s[border − 1]
  if P[i] == P[border]:
    border ← border + 1
  else:
    border ← 0
  s[i] ← border
return s
```

ComputePrefixFunction(P)

```
s ← array of integers of length |P|
s[0] ← 0, border ← 0
for i from 1 to |P| − 1:
    while (border > 0) and (P[i] ≠ P[border]):
        border ← s[border − 1]
    if P[i] == P[border]:
        border ← border + 1
    else:
        border ← 0
    s[i] ← border
return s
```

Finding out the longest border of our current border



Question

What is the maximum number of times that the condition of the inner while loop can be tested on the iteration i of the external for loop in this code (including the last time it is tested when either $border$ is 0 or $P[i] == P[border]$).

ComputePrefixFunction(P)

```
s ← array of integers of length |P|
s[0] ← 0, border ← 0
for i from 1 to |P| − 1:
    while (border > 0) and (P[i] ≠ P[border]):
        border ← s[border − 1]
    if P[i] == P[border]:
        border ← border + 1
    else:
        border ← 0
    s[i] ← border
return s
```

- ○ 2
- ◉ i
- ○ $i+1$
- ○ $i-1$

✓ Correct

Correct! If the string P is $AAA\ldots AAC$ with i As and one C, then on iteration i initially $border = i-1$, and we will compare $P[i] = B$ with $P[i-1], P[i-2], \ldots, P[1]$, and then $border$ will become 0, we will make the last test, and then the while loop will stop. There cannot be more iterations, because the $border$ cannot be more than $i-1$ initially, it decreases on each iteration of the while loop, and the while loop stops as soon as $border = 0$.

Lemma

The running time of `ComputePrefixFunction` is $O(|P|)$.

Proof

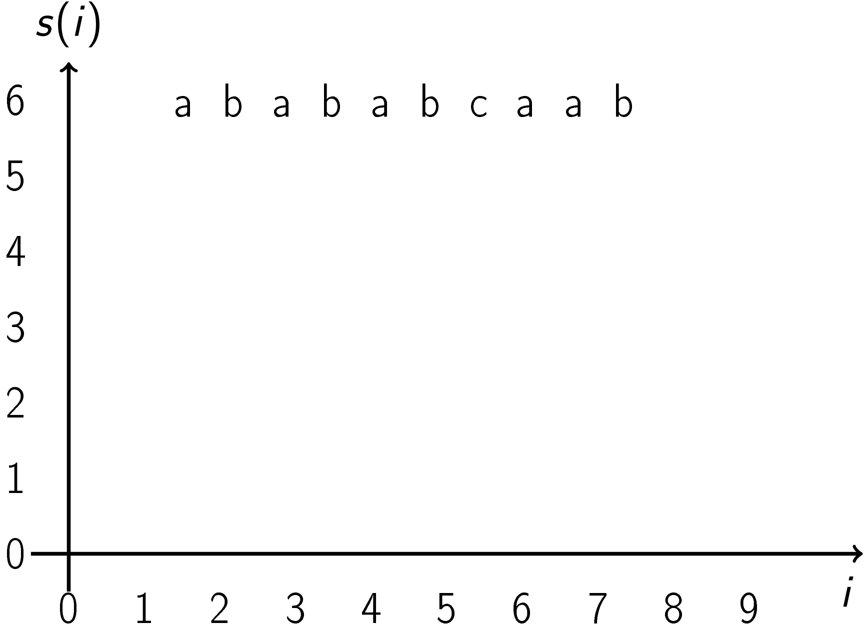
- Everything but for inner while loop is $O(|P|)$ initialization plus $O(|P|)$ iterations of the for loop with $O(1)$ assignments on each iteration

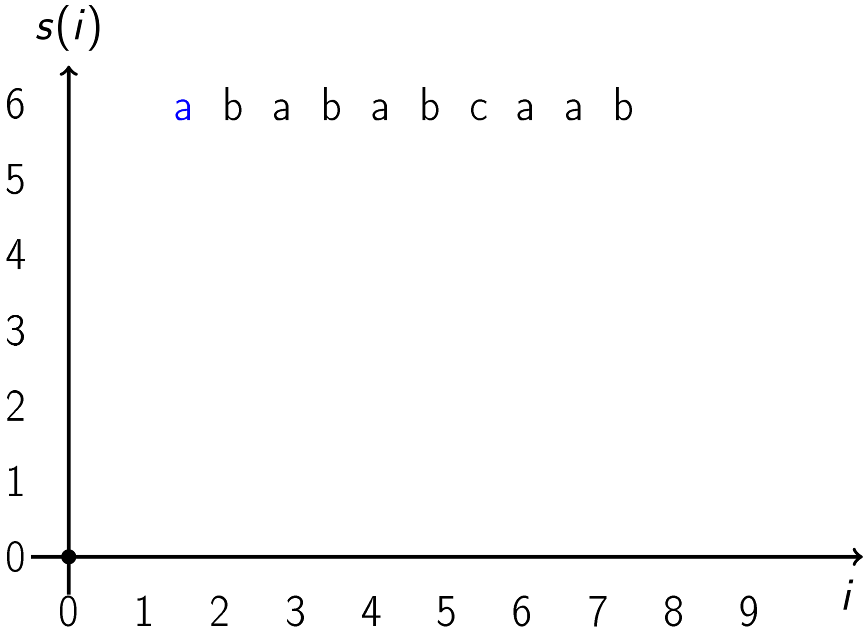
Simply means that everything except the while loop runs in O|P| time

Proof

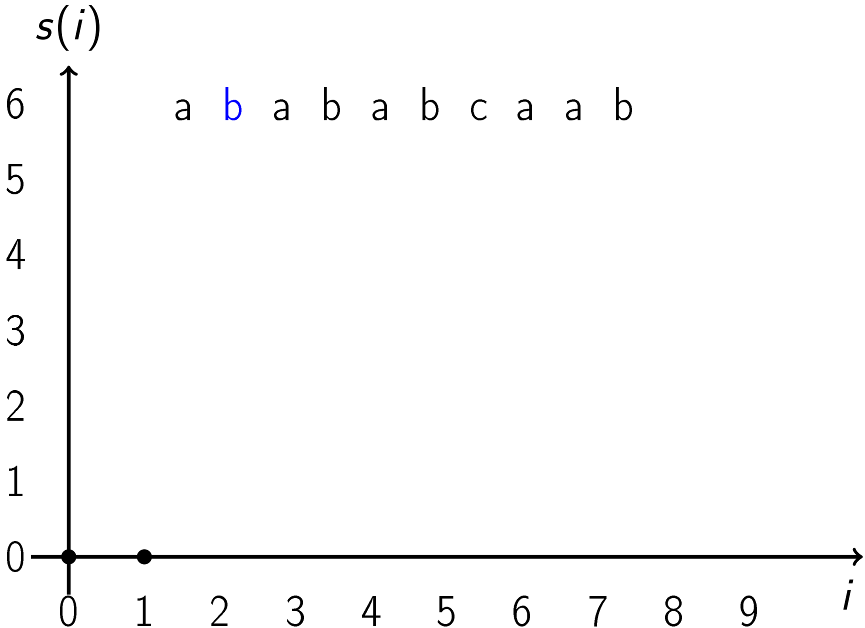
- Everything but for inner while loop is $O(|P|)$ initialization plus $O(|P|)$ iterations of the for loop with $O(1)$ assignments on each iteration
- Now we will bound the number of the while loop iterations by $O(|P|)$

To understand this consider value of the prefix function Vs positions in the string

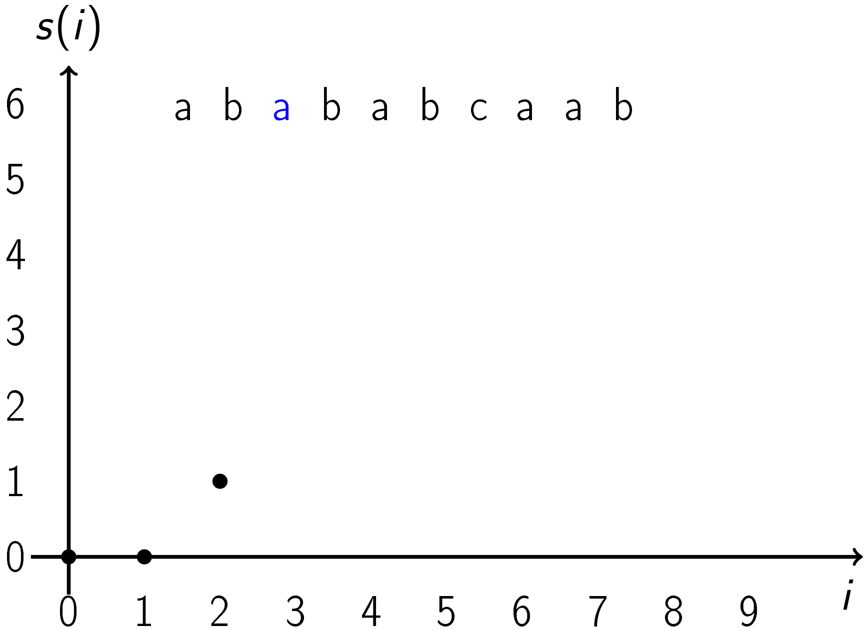

s(i)
6
5
4
3
2
1
0
a b a b a b c a a b
0 1 2 3 4 5 6 7 8 9
i



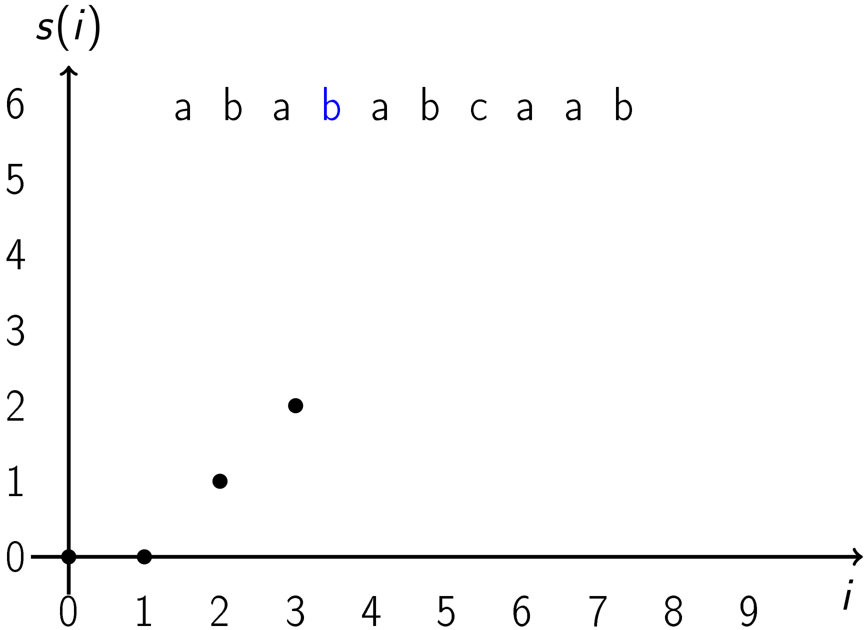
$s(i)$
6
5
4
3
2
1
0
a b a b a b c a a b
0 1 2 3 4 5 6 7 8 9
i



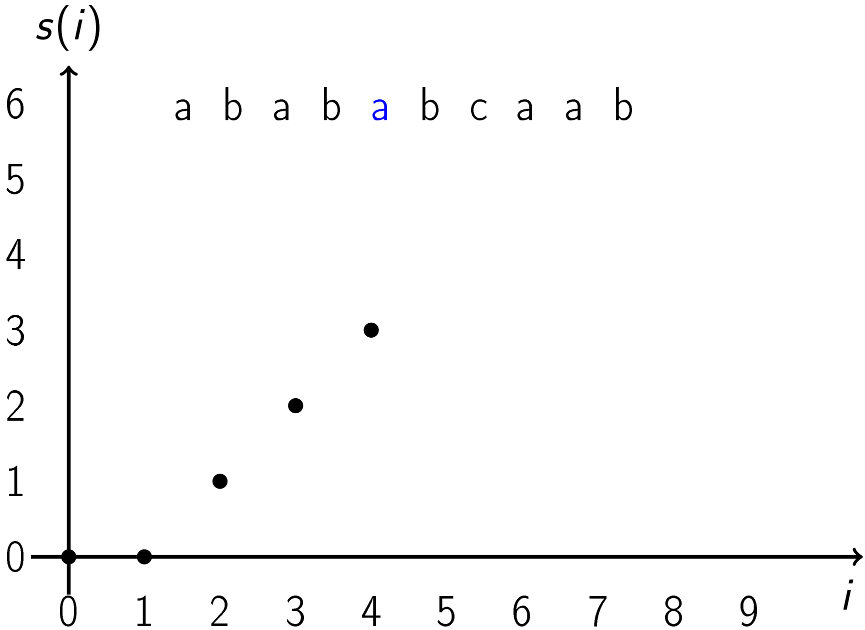
s(i)
a b a b a b c a a b
6
5
4
3
2
1
0
0 1 2 3 4 5 6 7 8 9
i



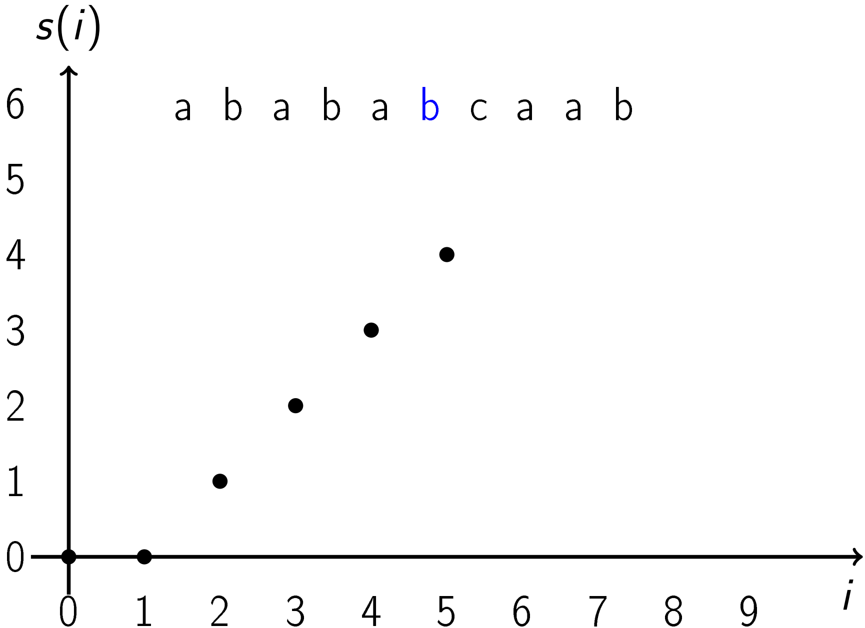
$s(i)$
a b a b a b c a a b
6
5
4
3
2
1
0
0 1 2 3 4 5 6 7 8 9
i

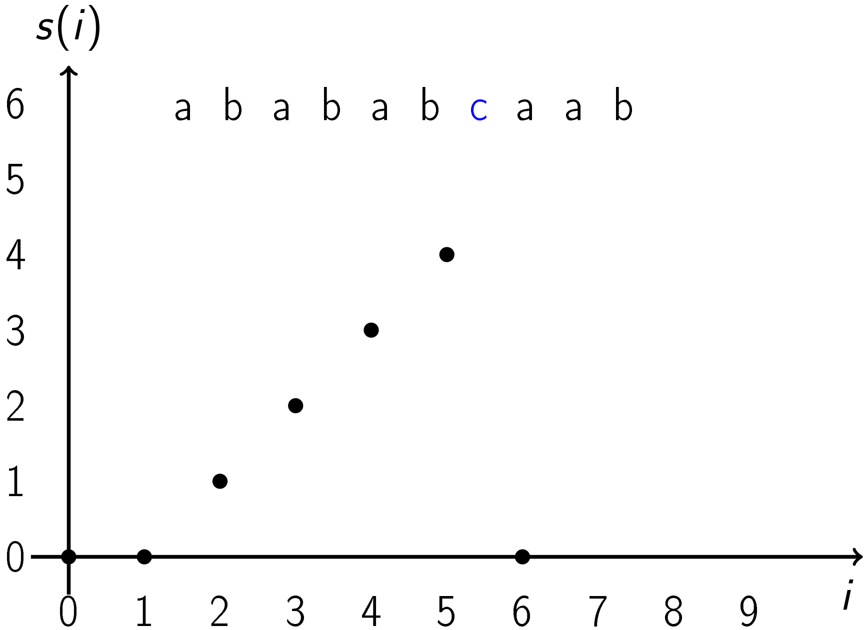


$s(i)$
a b a b a b c a a b
0 1 2 3 4 5 6
0 1 2 3 4 5 6 7 8 9
i

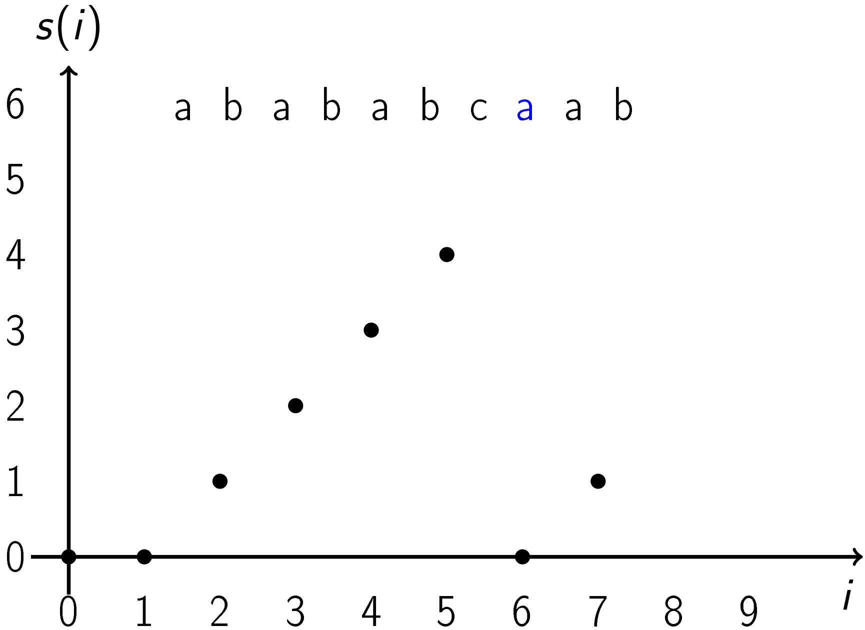


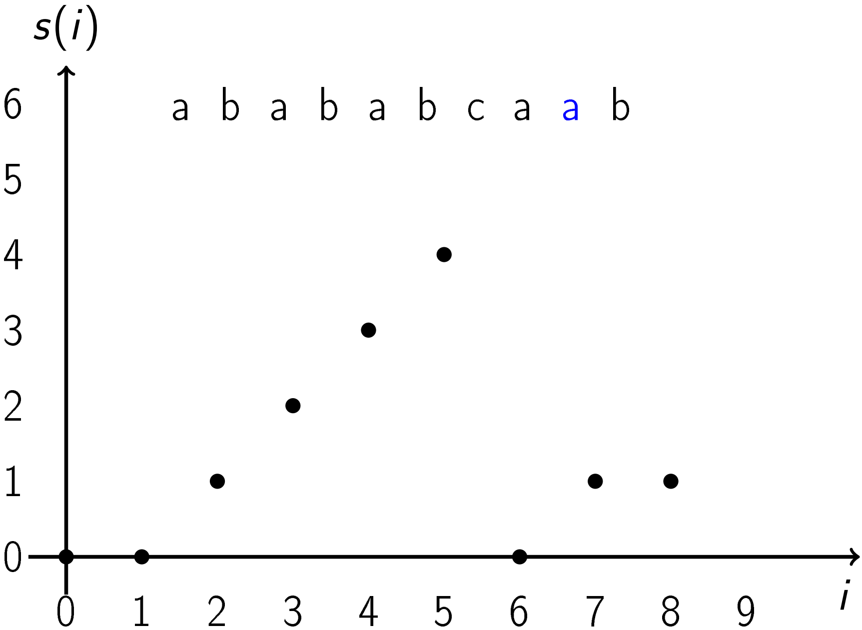
$s(i)$
a b a b a b c a a b
6
5
4
3
2
1
0
0 1 2 3 4 5 6 7 8 9
i

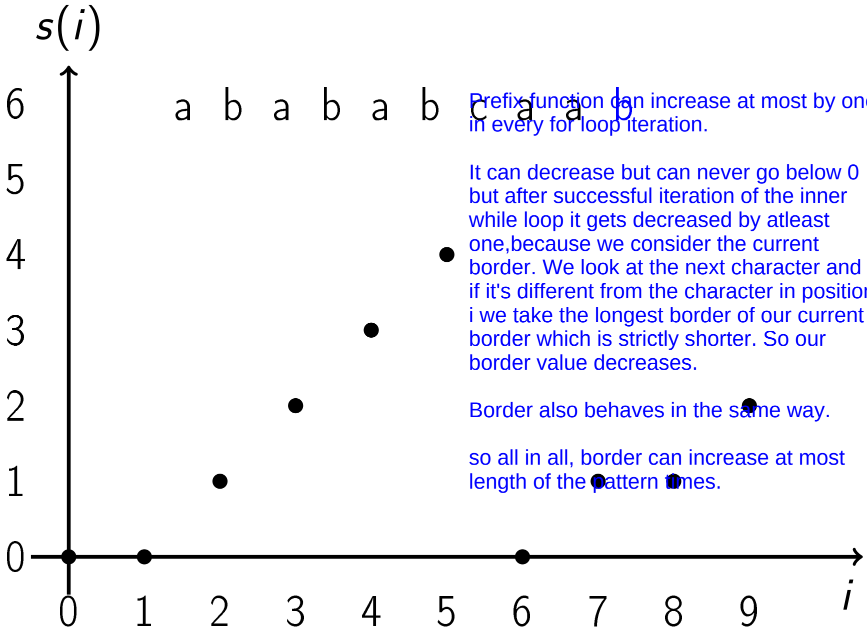

s(i)
a b a b a b c a a b
6
5
4
3
2
1
0
0 1 2 3 4 5 6 7 8 9
i



s(i)
a b a b a b c a a b
6
5
4
3
2
1
0
0 1 2 3 4 5 6 7 8 9
i


s(i)
a b a b a b c a a b
0 1 2 3 4 5 6
0 1 2 3 4 5 6 7 8 9
i


s(i)
a b a b a b c a a b
6
5
4
3
2
1
0
0 1 2 3 4 5 6 7 8 9
i



Prefix function can increase at most by one in every for loop iteration.

It can decrease but can never go below 0 but after successful iteration of the inner while loop it gets decreased by atleast one,because we consider the current border. We look at the next character and if it's different from the character in position i we take the longest border of our current border which is strictly shorter. So our border value decreases.

Border also behaves in the same way.

so all in all, border can increase at most length of the pattern times.

Proof

(continued)

- *border* can increase at most by 1 on each iteration of the for loop

Proof

(continued)

- *border* can increase at most by 1 on each iteration of the for loop
- In total, *border* is increased $O(|P|)$ times

Proof

(continued)

- *border* can increase at most by 1 on each iteration of the for loop
- In total, *border* is increased $O(|P|)$ times
- *border* is decreased at least by 1 on each iteration of the while loop

Proof

(continued)

- *border* can increase at most by 1 on each iteration of the for loop
- In total, *border* is increased $O(|P|)$ times
- *border* is decreased at least by 1 on each iteration of the while loop
- *border* ≥ 0

Proof

(continued)

- *border* can increase at most by 1 on each iteration of the for loop
- In total, *border* is increased $O(|P|)$ times
- *border* is decreased at least by 1 on each iteration of the while loop
- *border* ≥ 0
- So there are $O(|P|)$ iterations of the while loop

Hence total time will be O|P| + O|P| = 2.O|P| = O|P|

$\square$

- Now you know how to compute prefix function in linear time

- Now you know how to compute prefix function in linear time
- But how to find pattern in text??

Outline

1. Exact Pattern Matching
2. Safe Shift
3. Prefix Function
4. Computing Prefix Function
5. Knuth-Morris-Pratt Algorithm

Algorithm

	P							T								
S	a	b	r	a	$	a	b	r	a	c	a	d	a	b	r	a

To search for pattern P in text T:

- Create new string $S = P + '\$' + T$, where '$' is a special character absent from both P and T

Algorithm

	P								T							
S	a	b	r	a	$	a	b	r	a	c	a	d	a	b	r	a
s	0	0	0	1	0	1	2	3	4	0	1	0	1	2	3	4

To search for pattern P in text T:

- Compute prefix function s for string S

Algorithm

	P								T							
S	a	b	r	a	$	a	b	r	a	c	a	d	a	b	r	a
s	0	0	0	1	0	1	2	3	4	0	1	0	1	2	3	4

To search for pattern P in text T:

- Compute prefix function s for string S
- For all positions i such that $i > |P|$ and $s(i) = |P|$, add $i - 2|P|$ to the output

Algorithm

	P								T							
S	a	b	r	a	\$	a	b	r	a	c	a	d	a	b	r	a
s	0	0	0	1	0	1	2	3	4	0	1	0	1	2	3	4

To search for pattern P in text T:

- Compute prefix function s for string S
- For all positions i such that $i > |P|$ and $s(i) = |P|$, add $i - 2|P|$ to the output

Algorithm

	P								T							
S	a	b	r	a	\$	a	b	r	a	c	a	d	a	b	r	a
s	0	0	0	1	0	1	2	3	4	0	1	0	1	2	3	4

To search for pattern P in text T:

- Compute prefix function s for string S
- For all positions i such that $i > |P|$ and $s(i) = |P|$, add $i - 2|P|$ to the output

Algorithm

	P								T							
S	a	b	r	a	\$	a	b	r	a	c	a	d	a	b	r	a

s	0	0	0	1	0	1	2	3	4	0	1	0	1	2	3	4

To search for pattern P in text T:

- Compute prefix function s for string S
- For all positions i such that $i > |P|$ and $s(i) = |P|$, add $i - 2|P|$ to the output

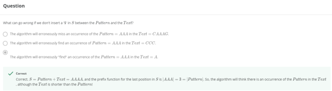
Algorithm

	P								T							
S	a	b	r	a	\$	a	b	r	a	c	a	d	a	b	r	a
s	0	0	0	1	0	1	2	3	4	0	1	0	1	2	3	4

To search for pattern P in text T:

- Compute prefix function s for string S
- For all positions i such that $i > |P|$ and $s(i) = |P|$, add $i - 2|P|$ to the output

Explanation


Question

What can go wrong if we don't insert a '$' in S between the $Pattern$ and the $Text$?

- The algorithm will erroneously miss an occurrence of the $Pattern = AAA$ in the $Text = CAAAG$.
- The algorithm will erroneously find an occurrence of $Pattern = AAA$ in the $Text = CCC$.
- The algorithm will erroneously "find" an occurrence of the $Pattern = AAA$ in the $Text = A$.

Correct
Correct. $S = Pattern + Text = AAAA$, and the prefix function for the last position in S is $|AAA| = 3 = |Pattern|$. So, the algorithm will think there is an occurrence of the $Pattern$ in the $Text$, although the $Text$ is shorter than the $Pattern$!

- For all i, $s(i) \leq |P|$ because of the special character '$'
- If $i > |P|$ and $s(i) = |P|$, then $P = S[0..|P|-1] = S[i-|P|+1..i] = T[i-2|P|..i-|P|-1]$
- If $s(i) < |P|$, no full occurrence of $|P|$ ends in position i

And why does algorithm even works? First, we need to notice that the prefix function for this string big S is always less than or equal to the length of the pattern. Because of the dollar sign it occurs right after the end of the pattern so when the border is bigger, we would need to have another occurrence of dollar in the string big S. But dollar is only between pattern and the text and is absent from the text. So prefix function cannot be bigger than the length of the pattern.

```
FindAllOccurrences(P, T)

S ← P + '$' + T
s ← ComputePrefixFunction(S)
result ← empty list
for i from |P| + 1 to |S| − 1:
  if s[i] == |P|:
    result.Append(i − 2|P|)
return result
```

`FindAllOccurrences`(P, T)

$S \leftarrow P +$ '\$' $+ T$

$s \leftarrow$ `ComputePrefixFunction(S)`

`result` $\leftarrow$ `empty list`

`for` i `from` $|P| + 1$ `to` $|S| - 1$:

  `if` $s[i] == |P|$:

    `result.Append`$(i - 2|P|)$

`return result`

```
FindAllOccurrences(P, T)
S ← P + '$' + T
s ← ComputePrefixFunction(S)
result ← empty list
for i from |P| + 1 to |S| − 1:
  if s[i] == |P|:
    result.Append(i − 2|P|)
return result
```

```
FindAllOccurrences(P, T)

S ← P + '$' + T
s ← ComputePrefixFunction(S)
result ← empty list
for i from |P| + 1 to |S| − 1:
  if s[i] == |P|:
    result.Append(i − 2|P|)
return result
```

```
FindAllOccurrences(P, T)
S ← P + '$' + T
s ← ComputePrefixFunction(S)
result ← empty list
for i from |P| + 1 to |S| − 1:
    if s[i] == |P|:
        result.Append(i − 2|P|)
return result
```

FindAllOccurrences(P, T)

```
S ← P + '$' + T
s ← ComputePrefixFunction(S)
result ← empty list
for i from |P| + 1 to |S| − 1:
    if s[i] == |P|:
        result.Append(i − 2|P|)
return result
```

```
FindAllOccurrences(P, T)

S ← P + '$' + T
s ← ComputePrefixFunction(S)
result ← empty list
for i from |P| + 1 to |S| − 1:
    if s[i] == |P|:
        result.Append(i − 2|P|)
return result
```

Lemma

The running time of Knuth-Morris-Pratt algorithm is $O(|P| + |T|)$.

Proof

- Building string S is $O(|P| + |T|)$

Lemma

The running time of Knuth-Morris-Pratt algorithm is $O(|P| + |T|)$.

Proof

- Building string S is $O(|P| + |T|)$
- Computing prefix function is $O(|S|) = O(|P| + |T|)$

Lemma

The running time of Knuth-Morris-Pratt algorithm is $O(|P| + |T|)$.

Proof

- Building string S is $O(|P| + |T|)$
- Computing prefix function is $O(|S|) = O(|P| + |T|)$
- The for loop runs $O(|S|) = O(|P| + |T|)$ iterations □

Conclusion

- Can search pattern in text in linear time
- Can compute prefix function of a string in linear time
- Can enumerate all borders of a string