

Algorithms for matching

Introduction to Computational Linguistics

Geert-Jan M. Kruijff

`<gj@dfki.de>`

Today's lecture





- Objective:
 - Efficient algorithms for finding matches of *patterns* (strings) in *texts*.
 - The focus is on *exact* matching
 - We deal with chars/Strings, but this generalizes to words/Strings



- Objective:
 - Efficient algorithms for finding matches of *patterns* (strings) in *texts*.
 - The focus is on *exact* matching
 - We deal with chars/Strings, but this generalizes to words/Strings
- Why efficient methods for pattern matching?
 - Applications of pattern matching in *search* (web search for IR, IE, Q/A), *tagging* (named entity recognition), *shallow processing* (parsing)
 - Efficiency pays off when dealing with large amounts of data!
 - Furthermore: preliminaries for finite-state automata, dynamic programming/memoization techniques in parsing

What will you learn?



Dan Gusfield. *Algorithms on Strings, Trees and Sequences*. CUP, 1997:
Chapters 1, 2, 3 [pp52-65]; Chapter 11



1. The naive method for exact string matching

- Method for finding matches of a pattern P in a text T using $O(|P| \cdot |T|)$ comparisons

Dan Gusfield. *Algorithms on Strings, Trees and Sequences*. CUP, 1997:
Chapters 1, 2, 3 [pp52-65]; Chapter 11

What will you learn?



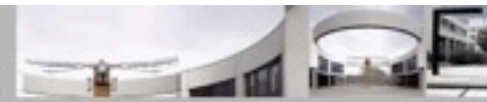
1. The naive method for exact string matching
 - Method for finding matches of a pattern P in a text T using $O(|P| \cdot |T|)$ comparisons
2. Methods for fundamental preprocessing of a pattern
 - Pre-process the pattern to make smarter shifts (i.e. longer ones) when a mismatch is found

Dan Gusfield. *Algorithms on Strings, Trees and Sequences*. CUP, 1997:
Chapters 1, 2, 3 [pp52-65]; Chapter 11



1. The naive method for exact string matching
 - Method for finding matches of a pattern P in a text T using $O(|P| \cdot |T|)$ comparisons
2. Methods for fundamental preprocessing of a pattern
 - Pre-process the pattern to make smarter shifts (i.e. longer ones) when a mismatch is found
3. The Boyer-Moore algorithm, the Knuth-Morris-Pratt algorithm
 - Smart shifts in sublinear $O(|P|+|T|)$ time (B-M) or linear time (K-M-P)

Dan Gusfield. *Algorithms on Strings, Trees and Sequences*. CUP, 1997:
Chapters 1, 2, 3 [pp52-65]; Chapter 11





- A **string** S is an ordered list of characters, written contiguously from left to right. For any string S , $S[i..j]$ is the (contiguous) **substring** of S that starts at position i and ends at position j .



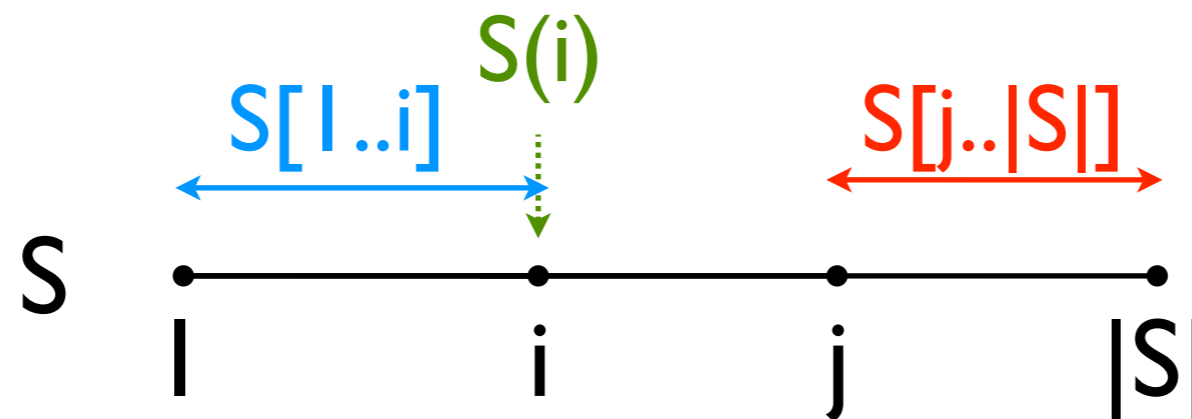
- A **string** S is an ordered list of characters, written contiguously from left to right. For any string S , $S[i..j]$ is the (contiguous) **substring** of S that starts at position i and ends at position j .
- The substring $S[1..i]$ is the **prefix** of S that ends at position i , and the substring $S[j..|S|]$ is the **suffix** of S starting at position j , with $|S|$ the length of S .



- A **string** S is an ordered list of characters, written contiguously from left to right. For any string S , $S[i..j]$ is the (contiguous) **substring** of S that starts at position i and ends at position j .
- The substring $S[1..i]$ is the **prefix** of S that ends at position i , and the substring $S[j..|S|]$ is the **suffix** of S starting at position j , with $|S|$ the length of S .
- For any string S , $S(i)$ denotes the **character** at position i in S .



- A **string** S is an ordered list of characters, written contiguously from left to right. For any string S , $S[i..j]$ is the (contiguous) **substring** of S that starts at position i and ends at position j .
- The substring $S[1..i]$ is the **prefix** of S that ends at position i , and the substring $S[j..|S|]$ is the **suffix** of S starting at position j , with $|S|$ the length of S .
- For any string S , $S(i)$ denotes the **character** at position i in S .





- Given

- a **pattern** P , and a text T in which we are looking for matches of P
- **Pointers:** p to position in P ; t to position in T ; s to start of matching P in T

- Algorithm

[Start: $p=1, t=1, s=1$]

1. Align the left of P with the left of T : set position in P , $p=1$; set position in T , $t=1$
2. Set the current left-alignment position in T to $s=1$

[Loop]

2. Compare the character at $P(p)$ with the character at $T(t)$
3. If $P(p) == T(t)$,
 4. If $p < |P|$ then set $p=p+1$ and set $t=t+1$; else report match, and set $p=1, s=s+1, t=s$;
 5. Else $p=1$ and $s=s+1, t=s$



[Start: $p=1, t=1, s=1$]

1. Align the left of P with the left of T:
2. $p=1; t=1; s=1$

[Loop]

2. Compare $P(p)$ with $T(t)$
3. If $P(p) == T(t)$,
 4. If $p < |P|$
then $p=p+1$ and $t=t+1$;
else report match, and $p=1, s=s+1, t=s$;
5. Else $p=1$ and $s=s+1, t=s$

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

[Start: $p=1, t=1, s=1$]

1. Align the left of P with the left of T:
2. $p=1; t=1; s=1$

[Loop]

2. Compare $P(p)$ with $T(t)$
3. If $P(p) == T(t)$,
 4. If $p < |P|$
then $p=p+1$ and $t=t+1$;
else report match, and $p=1, s=s+1, t=s$;
5. Else $p=1$ and $s=s+1, t=s$

The naive method for matching



	1	2	3	4	5	6	7	8	9	10	11	12	13
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z
P=	A	B	X	Y	A	B	X	Z					

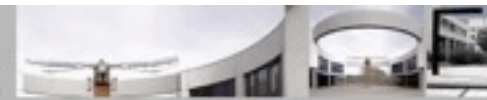
[Start: $p=1, t=1, s=1$]

1. Align the left of P with the left of T:
2. $p=1; t=1; s=1$

[Loop]

2. Compare $P(p)$ with $T(t)$
3. If $P(p) == T(t)$,
 4. If $p < |P|$
then $p=p+1$ and $t=t+1$;
else report match, and $p=1, s=s+1, t=s$;
5. Else $p=1$ and $s=s+1, t=s$

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

[Start: $p=1, t=1, s=1$]

1. Align the left of P with the left of T:
2. $p=1; t=1; s=1$

[Loop]

2. Compare $P(p)$ with $T(t)$
3. If $P(p) == T(t)$,
 4. If $p < |P|$
then $p=p+1$ and $t=t+1$;
else report match, and $p=1, s=s+1, t=s$;
5. Else $p=1$ and $s=s+1, t=s$

The naive method for matching



↓
1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z
P= A B X Y A B X Z

✗

[Start: $p=1, t=1, s=1$]

1. Align the left of P with the left of T:
2. $p=1; t=1; s=1$

[Loop]

2. Compare $P(p)$ with $T(t)$
3. If $P(p) == T(t)$,
 4. If $p < |P|$
then $p=p+1$ and $t=t+1$;
else report match, and $p=1, s=s+1, t=s$;
5. Else $p=1$ and $s=s+1, t=s$

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

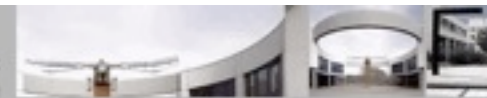
[Start: $p=1, t=1, s=1$]

1. Align the left of P with the left of T:
2. $p=1; t=1; s=1$

[Loop]

2. Compare $P(p)$ with $T(t)$
3. If $P(p) == T(t)$,
 4. If $p < |P|$
then $p=p+1$ and $t=t+1$;
else report match, and $p=1, s=s+1, t=s$;
5. Else $p=1$ and $s=s+1, t=s$

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

A B X Y A B X Z

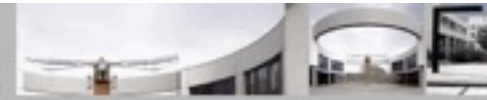
[Start: $p=1, t=1, s=1$]

1. Align the left of P with the left of T:
2. $p=1; t=1; s=1$

[Loop]

2. Compare $P(p)$ with $T(t)$
3. If $P(p) == T(t)$,
 4. If $p < |P|$
then $p=p+1$ and $t=t+1$;
else report match, and $p=1, s=s+1, t=s$;
5. Else $p=1$ and $s=s+1, t=s$

The naive method for matching



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

×

A B X Y A B X Z

[Start: $p=1, t=1, s=1$]

1. Align the left of P with the left of T:
2. $p=1; t=1; s=1$

[Loop]

2. Compare $P(p)$ with $T(t)$
3. If $P(p) == T(t)$,
 4. If $p < |P|$
then $p=p+1$ and $t=t+1$;
else report match, and $p=1, s=s+1, t=s$;
5. Else $p=1$ and $s=s+1, t=s$

The naive method for matching



↓
1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z



p=1
t=2
s=2

A B X Y A B X Z



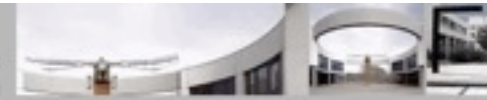
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z



p=1
t=2
s=2

A B X Y A B X Z



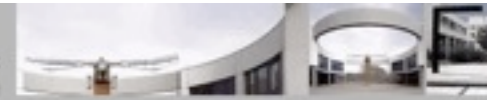
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z



p=1
t=2
s=2

A B X Y A B X Z



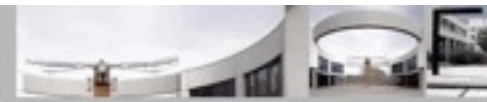
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

✗

p=1 p=2
t=2 t=3
s=2 s=2

A B X Y A B X Z

✓ ✓

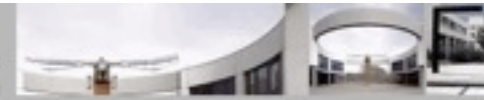
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

✗

p=1 p=2
t=2 t=3
s=2 s=2

A B X Y A B X Z

✓ ✓

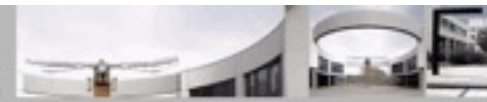
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓
1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

✗

p=1 p=2
t=2 t=3
s=2 s=2

A B X Y A B X Z

✓ ✓

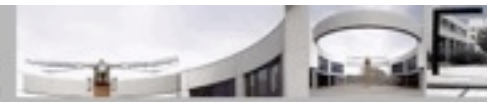
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

✗

p=1	p=2	p=3
t=2	t=3	t=4
s=2	s=2	s=2

A B X Y A B X Z

✓ ✓ ✓

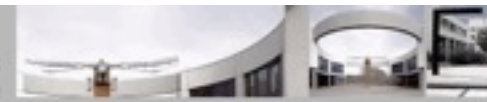
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

p=1	p=2	p=3
t=2	t=3	t=4
s=2	s=2	s=2

A B X Y A B X Z

✓ ✓ ✓

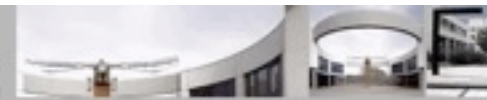
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

p=1	p=2	p=3
t=2	t=3	t=4
s=2	s=2	s=2

A B X Y A B X Z

✓ ✓ ✓

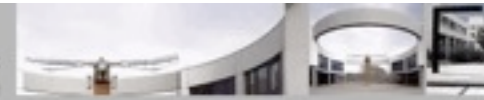
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

p=1	p=2	p=3	p=4
t=2	t=3	t=4	t=5
s=2	s=2	s=2	s=2

A B X Y A B X Z

✓ ✓ ✓ ✓

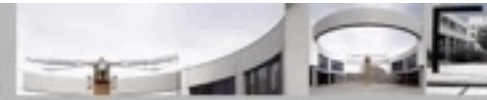
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

p=1	p=2	p=3	p=4
t=2	t=3	t=4	t=5
s=2	s=2	s=2	s=2

A B X Y A B X Z

✓ ✓ ✓ ✓

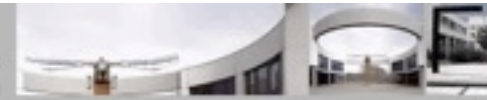
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

✗

p=1	p=2	p=3	p=4
t=2	t=3	t=4	t=5
s=2	s=2	s=2	s=2

A B X Y A B X Z

✓ ✓ ✓ ✓

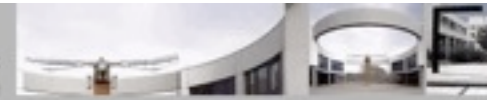
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

	1	2	3	4	5	6	7	8	9	10	11	12	13
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z
P=	A	B	X	Y	A	B	X	Z					

✗

p=1	p=2	p=3	p=4	p=5
t=2	t=3	t=4	t=5	t=6
s=2	s=2	s=2	s=2	s=2

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓

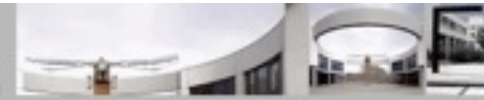
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

p=1	p=2	p=3	p=4	p=5
t=2	t=3	t=4	t=5	t=6
s=2	s=2	s=2	s=2	s=2

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓

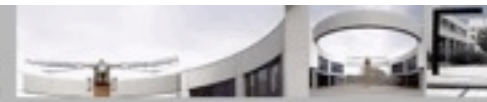
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z



p=1	p=2	p=3	p=4	p=5
t=2	t=3	t=4	t=5	t=6
s=2	s=2	s=2	s=2	s=2

A B X Y A B X Z



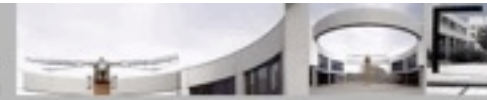
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z



p=1	p=2	p=3	p=4	p=5	p=6
t=2	t=3	t=4	t=5	t=6	t=7
s=2	s=2	s=2	s=2	s=2	s=2

A B X Y A B X Z



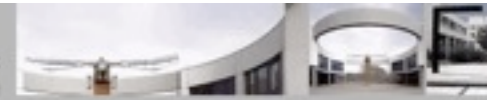
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

p=1	p=2	p=3	p=4	p=5	p=6
t=2	t=3	t=4	t=5	t=6	t=7
s=2	s=2	s=2	s=2	s=2	s=2

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓

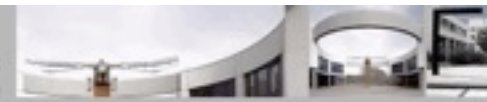
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

✗

p=1	p=2	p=3	p=4	p=5	p=6
t=2	t=3	t=4	t=5	t=6	t=7
s=2	s=2	s=2	s=2	s=2	s=2

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓

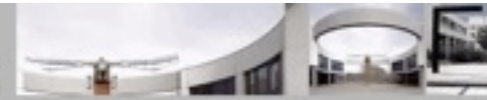
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

✗

p=1	p=2	p=3	p=4	p=5	p=6	p=7
t=2	t=3	t=4	t=5	t=6	t=7	t=8
s=2	s=2	s=2	s=2	s=2	s=2	s=2

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓

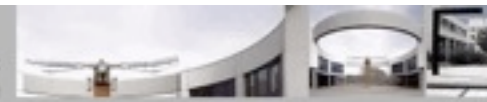
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

p=1	p=2	p=3	p=4	p=5	p=6	p=7
t=2	t=3	t=4	t=5	t=6	t=7	t=8
s=2	s=2	s=2	s=2	s=2	s=2	s=2

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓

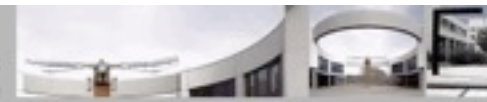
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z



p=1	p=2	p=3	p=4	p=5	p=6	p=7
t=2	t=3	t=4	t=5	t=6	t=7	t=8
s=2	s=2	s=2	s=2	s=2	s=2	s=2

A B X Y A B X Z



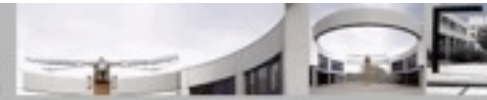
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z



p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2
A	B	X	Y	A	B	X	Z
✓	✓	✓	✓	✓	✓	✓	✗

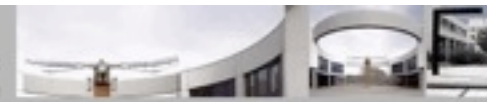
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
then p=p+1 and t=t+1;
else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

	1	2	3	4	5	6	7	8	9	10	11	12	13
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z
P=	A	B	X	Y	A	B	X	Z					

✗

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A	B	X	Y	A	B	X	Z
✓	✓	✓	✓	✓	✓	✓	✗

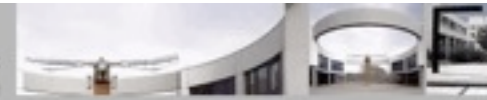
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ X

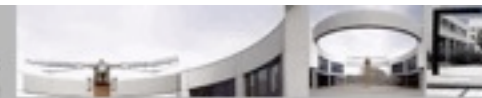
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

1	2	3	4	5	6	7	8	9	10	11	12	13	
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z

P= **A B X Y A B X Z**

✗

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✗

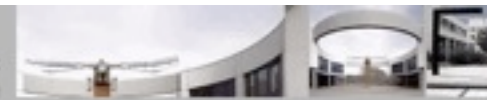
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

1	2	3	4	5	6	7	8	9	10	11	12	13	
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z

P= A B X Y A B X Z

✗

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A **B** **X** **Y** **A** **B** **X** **Z**
✓ ✓ ✓ ✓ ✓ ✓ ✓ ✗
A **B** **X** **Y** **A** **B** **X** **Z**
✗

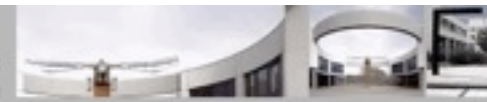
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ X

A B X Y A B X Z

X

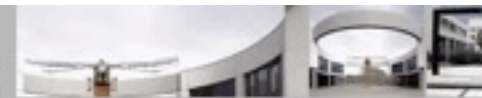
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

1	2	3	4	5	6	7	8	9	10	11	12	13	
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z

P= **A B X Y A B X Z**

✗

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A	B	X	Y	A	B	X	Z
✓	✓	✓	✓	✓	✓	✓	✗
A	B	X	Y	A	B	X	Z
✗							

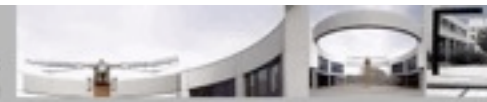
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

1	2	3	4	5	6	7	8	9	10	11	12	13	
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z

P= **A B X Y A B X Z**

✗

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

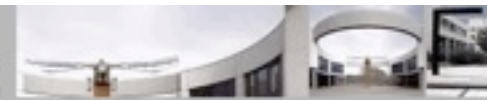
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ X

A B X Y A B X Z

X

A B X Y A B X Z

X

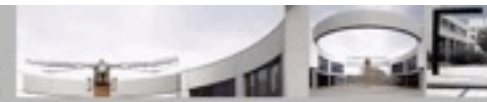
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

1	2	3	4	5	6	7	8	9	10	11	12	13	
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z

P= A B X Y A B X Z

✗

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

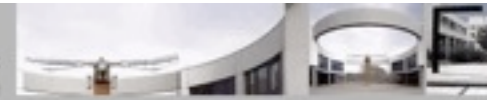
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

✗

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

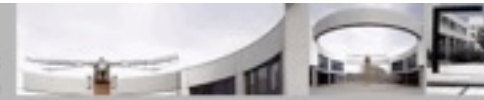
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ X

A B X Y A B X Z

X

A B X Y A B X Z

X

A B X Y A B X Z

X

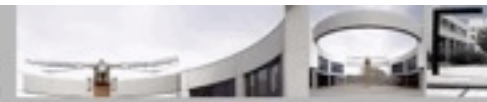
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

✗

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

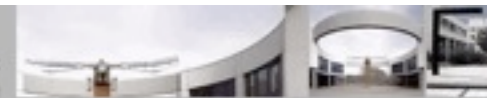
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

✗

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

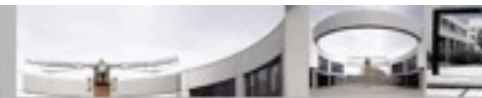
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

1	2	3	4	5	6	7	8	9	10	11	12	13	
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z

P= A B X Y A B X Z

✗

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✓

[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

1	2	3	4	5	6	7	8	9	10	11	12	13	
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z

P= A B X Y A B X Z

✗

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✓ ✓

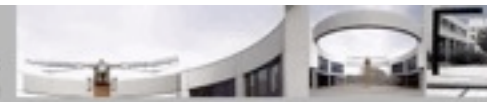
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

1	2	3	4	5	6	7	8	9	10	11	12	13	
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z

P= **A B X Y A B X Z**

✗

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✓ ✓ ✓

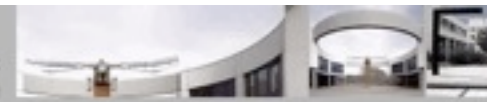
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

	1	2	3	4	5	6	7	8	9	10	11	12	13
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z

P= A B X Y A B X Z

✗

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✓ ✓ ✓ ✓

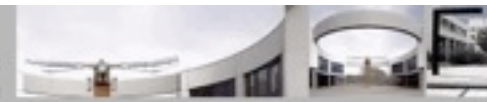
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

1	2	3	4	5	6	7	8	9	10	11	12	13	
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z

P= **A B X Y A B X Z**

✗

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓

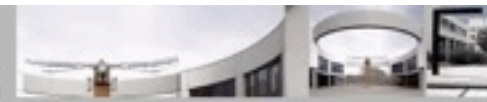
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

✗

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓

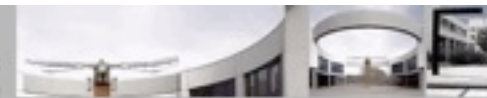
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

1	2	3	4	5	6	7	8	9	10	11	12	13	
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z

P= A B X Y A B X Z

✗

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓

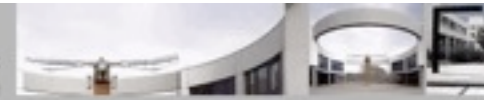
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

1	2	3	4	5	6	7	8	9	10	11	12	13	
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z

P= **A B X Y A B X Z**

✗

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓

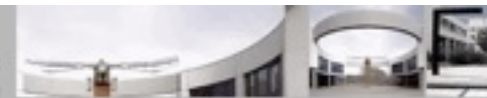
[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

The naive method for matching



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

✗

p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=7	⇒	p=1
t=2	t=3	t=4	t=5	t=6	t=7	t=8	t=8		t=3
s=2	s=2	s=2	s=2	s=2	s=2	s=2	s=2		s=3

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓

[Start: p=1, t=1, s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t),
 4. If p < |P|
 - then p=p+1 and t=t+1;
 - else report match, and p=1, s=s+1, t=s;
5. Else p=1 and s=s+1, t=s

In total,
we had to make

20

comparisons

The naive method for matching





- Observations

- The worst-case number of comparisons is $O(|P| \cdot |T|)$
- This is not so useful in real-life applications!
- E.g. $|P|=30$ and $|T|=200K$: 6M comparisons; with 1ms per comparison this would mean 6000s, or 100 minutes, i.e. 1:40h. If we manage to get linear complexity $O(|P|+|T|)$ we are down to 3.33min!



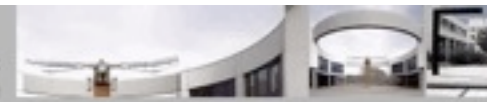
- Observations

- The worst-case number of comparisons is $O(|P| \cdot |T|)$
- This is not so useful in real-life applications!
- E.g. $|P|=30$ and $|T|=200K$: 6M comparisons; with 1ms per comparison this would mean 6000s, or 100 minutes, i.e. 1:40h. If we manage to get linear complexity $O(|P|+|T|)$ we are down to 3.33min!

- Ideas for speeding up the naive method

- Try to shift further when a mismatch occurs, but never so far as to miss an occurrence of P in T

Speeding up thru smarter shifting



Speeding up thru smarter shifting



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

Speeding up thru smarter shifting



	1	2	3	4	5	6	7	8	9	10	11	12	13
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z
P=	A	B	X	Y	A	B	X	Z					

Speeding up thru smarter shifting



↓

	1	2	3	4	5	6	7	8	9	10	11	12	13
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z
P=	A	B	X	Y	A	B	X	Z					

Speeding up thru smarter shifting



↓
1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z
P= A B X Y A B X Z
X

Speeding up thru smarter shifting



	1	2	3	4	5	6	7	8	9	10	11	12	13
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z
P=	A	B	X	Y	A	B	X	Z					
			X										

Speeding up thru smarter shifting



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

Speeding up thru smarter shifting



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X
A B X Y A B X Z

Speeding up thru smarter shifting



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

✗

A B X Y A B X Z

✓

Speeding up thru smarter shifting



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

✗

A B X Y A B X Z

✓ ✓

Speeding up thru smarter shifting



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

A B X Y A B X Z

✓ ✓ ✓

Speeding up thru smarter shifting



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

A B X Y A B X Z

✓ ✓ ✓ ✓

Speeding up thru smarter shifting



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓

Speeding up thru smarter shifting



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓

Speeding up thru smarter shifting



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓

Speeding up thru smarter shifting



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ X

Speeding up thru smarter shifting



↓
1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z
P= A B X Y A B X Z

✗

A B X Y A B X Z
✓ ✓ ✓ ✓ ✓ ✓ ✓ ✗

The next occurrence of $P(l) = \text{"A"}$
in T is not before position 5 in T,
so shift to position 6!

Speeding up thru smarter shifting



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z
P= A B X Y A B X Z

X

A B X Y A B X Z
✓ ✓ ✓ ✓ ✓ ✓ ✓ X

The next occurrence of $P(1) = "A"$
in T is not before position 5 in T,
so shift to position 6!

Speeding up thru smarter shifting



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z
P= A B X Y A B X Z

X

A B X Y A B X Z
✓ ✓ ✓ ✓ ✓ ✓ ✓ X

The next occurrence of $P(1) = \text{"A"}$
in T is not before position 5 in T,
so shift to position 6!

Speeding up thru smarter shifting



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ X

The next occurrence of $P(l) = \text{"A"}$
in T is not before position 5 in T,
so shift to position 6!

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓

Speeding up thru smarter shifting



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ X

The next occurrence of P(1)="A"
in T is not before position 5 in T,
so shift to position 6!

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓

In total,
we had to make

17

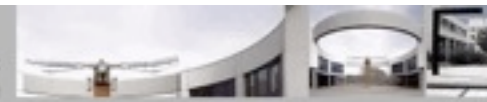
comparisons

Speeding up thru smarter shifting



	1	2	3	4	5	6	7	8	9	10	11	12	13
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z
P=	A	B	X	Y	A	B	X	Z					

Speeding up thru smarter shifting



↓

	1	2	3	4	5	6	7	8	9	10	11	12	13
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z
P=	A	B	X	Y	A	B	X	Z					

Speeding up thru smarter shifting



↓
1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z
P= A B X Y A B X Z
X

Speeding up thru smarter shifting



	1	2	3	4	5	6	7	8	9	10	11	12	13
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z
P=	A	B	X	Y	A	B	X	Z					
			X										

Speeding up thru smarter shifting



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

Speeding up thru smarter shifting



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ X

Speeding up thru smarter shifting



↓
1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z
P= A B X Y A B X Z

X
A B X Y A B X Z
✓ ✓ ✓ ✓ ✓ ✓ ✓ X

Assume: we know prefix $P[1..3]=\text{"A B X"}$
starts at $T(6)$. $P[1..3]=T[6..8]$; align at $T(6)$
but start matching $P(4)$ against $T(9)$

Speeding up thru smarter shifting



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z
P= A B X Y A B X Z

X

A B X Y A B X Z
✓ ✓ ✓ ✓ ✓ ✓ ✓ X

Assume: we know prefix $P[1..3]=\text{"A B X"}$
starts at $T(6)$. $P[1..3]=T[6..8]$; align at $T(6)$
but start matching $P(4)$ against $T(9)$

Speeding up thru smarter shifting



↓

	1	2	3	4	5	6	7	8	9	10	11	12	13
T=	X	A	B	X	Y	A	B	X	Y	A	B	X	Z
P=	A	B	X	Y	A	B	X	Z					

X

A	B	X	Y	A	B	X	Z
✓	✓	✓	✓	✓	✓	✓	X

Assume: we know prefix $P[1..3] = "A B X"$
starts at $T(6)$. $P[1..3] = T[6..8]$; align at $T(6)$
but start matching $P(4)$ against $T(9)$

Speeding up thru smarter shifting



↓

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

✗

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✗

Assume: we know prefix $P[1..3]=\text{"A B X"}$
starts at $T(6)$. $P[1..3]=T[6..8]$; align at $T(6)$
but start matching $P(4)$ against $T(9)$

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓

Speeding up thru smarter shifting



1 2 3 4 5 6 7 8 9 10 11 12 13
T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

X

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ X

Assume: we know prefix $P[1..3]=\text{"A B X"}$
starts at $T(6)$. $P[1..3]=T[6..8]$; align at $T(6)$
but start matching $P(4)$ against $T(9)$

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓

In total,
we had to make

14

comparisons

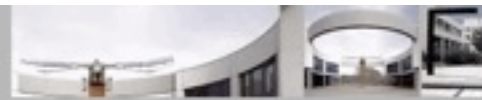


- Before searching, preprocess P (or T, or P+T)
- Fundamental preprocessing of a string S
 - At $S(i)$, $i > 1$ compute length of longest prefix of $S[i..|S|]$ that is a prefix of S
 - Let $Z_i(S)$ be that length at i



- Before searching, preprocess P (or T, or P+T)
- Fundamental preprocessing of a string S
 - At $S(i)$, $i > 1$ compute length of longest prefix of $S[i..|S|]$ that is a prefix of S
 - Let $Z_i(S)$ be that length at i

1 2 3 4 5 6 7 8 9 10 11
S = A A B C A A B X A A Z



- Before searching, preprocess P (or T, or P+T)
- Fundamental preprocessing of a string S
 - At $S(i)$, $i > 1$ compute length of longest prefix of $S[i..|S|]$ that is a prefix of S
 - Let $Z_i(S)$ be that length at i

1 2 3 4 5 6 7 8 9 10 11
S = A A B C A A B X A A Z

$Z_5(S)=3$: (A A B C...A A B X)



- Before searching, preprocess P (or T, or P+T)
- Fundamental preprocessing of a string S
 - At $S(i)$, $i > 1$ compute length of longest prefix of $S[i..|S|]$ that is a prefix of S
 - Let $Z_i(S)$ be that length at i

1 2 3 4 5 6 7 8 9 10 11
S = A A B C A A B X A A Z

$Z_5(S)=3$: (A A B C...A A B X)

$Z_6(S)=1$: (A A ...A B)



- Before searching, preprocess P (or T, or P+T)
- Fundamental preprocessing of a string S
 - At $S(i)$, $i > 1$ compute length of longest prefix of $S[i..|S|]$ that is a prefix of S
 - Let $Z_i(S)$ be that length at i

1 2 3 4 5 6 7 8 9 10 11
S = A A B C A A B X A A Z

$Z_5(S)=3$: (**A A B** C...**A A B** X)

$Z_6(S)=1$: (**A** A ...**A** B)

$Z_7(S)=Z_8(S)=0$



- Before searching, preprocess P (or T, or P+T)
- Fundamental preprocessing of a string S
 - At $S(i)$, $i > 1$ compute length of longest prefix of $S[i..|S|]$ that is a prefix of S
 - Let $Z_i(S)$ be that length at i

1 2 3 4 5 6 7 8 9 10 11
S = A A B C A A B X A A Z

$Z_5(S)=3$: (A A B C...A A B X)

$Z_6(S)=1$: (A A ...A B)

$Z_7(S)=Z_8(S)=0$

$Z_9(S)=2$: (A A B ...A A Z)



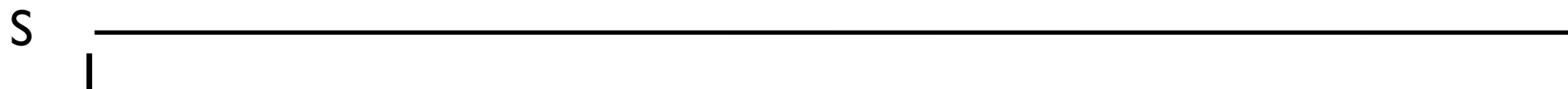
- Given a string $S=P\$T$
 - The dollar sign $\$$ is not in the languages for P or T
 - $|P|=n$, $|T|=m$, $n \leq m$, so $S=n+m+1$
- Compute $Z_i(S)$ for $2 < i < n+m+1$
 - Because “ $\$$ ” is not in the language for P , $Z_i(S) \leq n$ for every $i > 1$
 - $Z_i(S)=n$ for $i > n+1$ identifies an occurrence of P starting at $i-(n+1)$ in T
 - Also: If P occurs in T starting at position j , then it must be that $Z_{(n+1)+j}(S)=n$
- If $Z_i(S)$ is computable in linear time, then we have linear time matching
 - Matching = search \Rightarrow matching = preprocessing + search



- The task: Compute $Z_i(S)$ in linear time, i.e. $O(|S|)$
- The notion of a **Z-box**
 - For every $i > 1$ with $Z_i(S) > 0$, define a Z-box to be the substring from i until $i+Z_i(S)-1$, i.e. $S[i\dots i+Z_i(S)-1]$
 - For every $i > 1$, r_i is the right-most endpoint of the Z-boxes that begin at or before i ;
 - i.e, r_i is the largest value of $j+Z_j(S)-1$ for all $1 < j \leq i$ such that $Z_j(S) > 0$

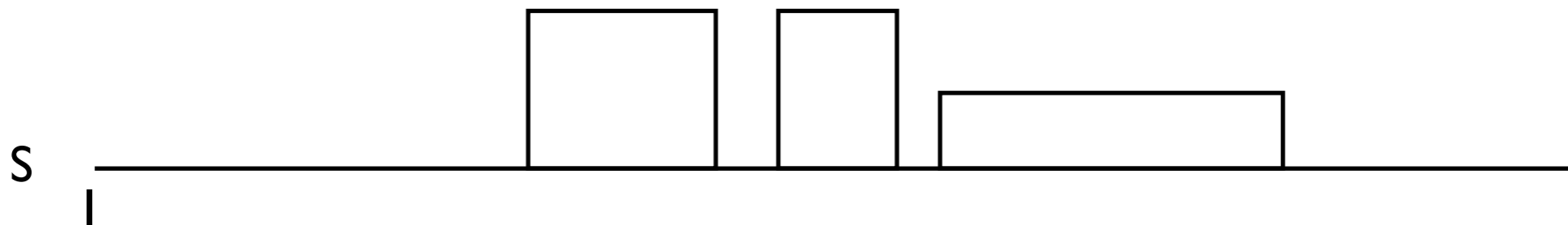


- The task: Compute $Z_i(S)$ in linear time, i.e. $O(|S|)$
- The notion of a **Z-box**
 - For every $i > 1$ with $Z_i(S) > 0$, define a Z-box to be the substring from i until $i+Z_i(S)-1$, i.e. $S[i\dots i+Z_i(S)-1]$
 - For every $i > 1$, r_i is the right-most endpoint of the Z-boxes that begin at or before i ;
 - i.e, r_i is the largest value of $j+Z_j(S)-1$ for all $1 < j \leq i$ such that $Z_j(S) > 0$



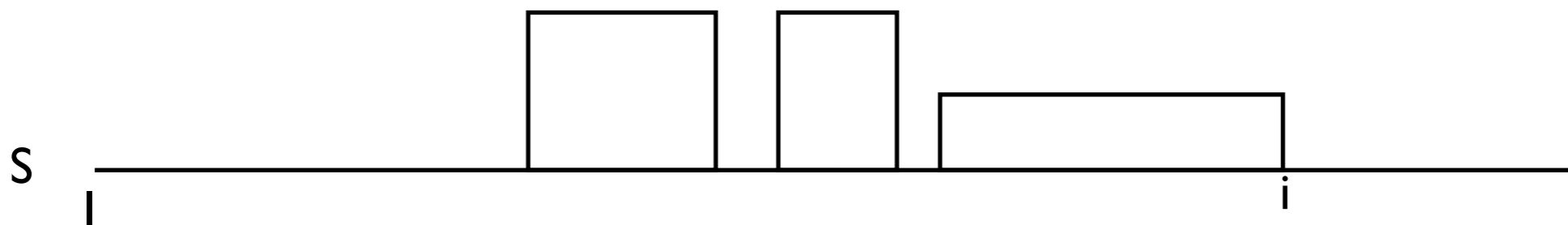


- The task: Compute $Z_i(S)$ in linear time, i.e. $O(|S|)$
- The notion of a **Z-box**
 - For every $i > 1$ with $Z_i(S) > 0$, define a Z-box to be the substring from i until $i+Z_i(S)-1$, i.e. $S[i\dots i+Z_i(S)-1]$
 - For every $i > 1$, r_i is the right-most endpoint of the Z-boxes that begin at or before i ;
 - i.e, r_i is the largest value of $j+Z_j(S)-1$ for all $1 < j \leq i$ such that $Z_j(S) > 0$



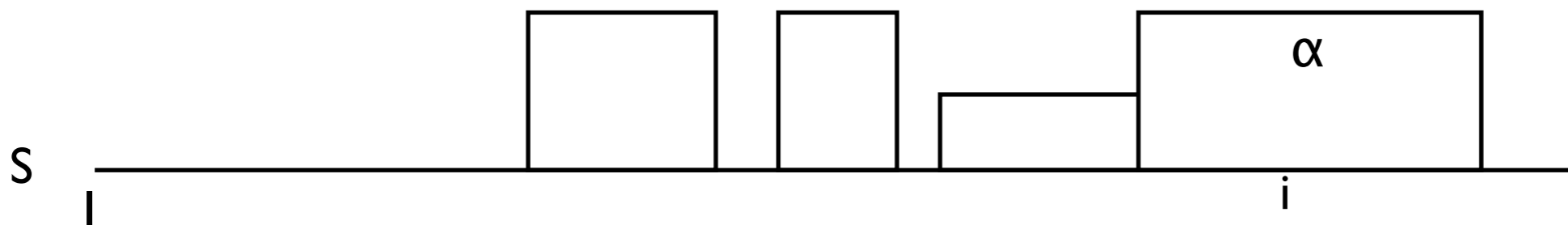


- The task: Compute $Z_i(S)$ in linear time, i.e. $O(|S|)$
- The notion of a **Z-box**
 - For every $i > 1$ with $Z_i(S) > 0$, define a Z-box to be the substring from i until $i+Z_i(S)-1$, i.e. $S[i\dots i+Z_i(S)-1]$
 - For every $i > 1$, r_i is the right-most endpoint of the Z-boxes that begin at or before i ;
 - i.e, r_i is the largest value of $j+Z_j(S)-1$ for all $1 < j \leq i$ such that $Z_j(S) > 0$



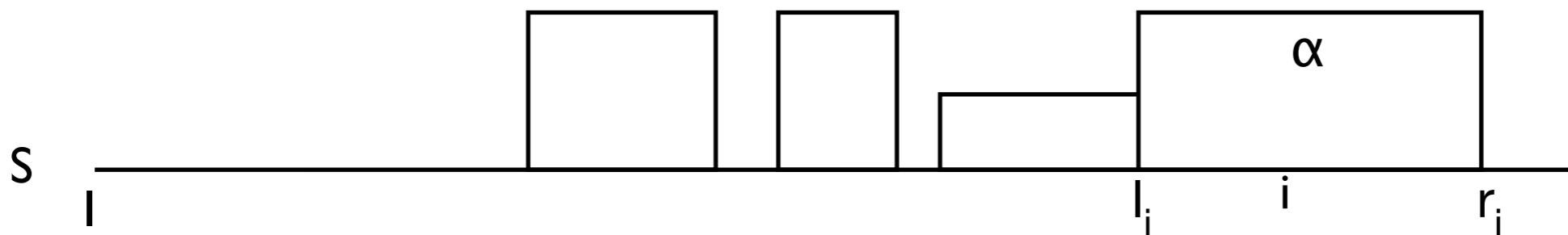


- The task: Compute $Z_i(S)$ in linear time, i.e. $O(|S|)$
- The notion of a **Z-box**
 - For every $i > 1$ with $Z_i(S) > 0$, define a Z-box to be the substring from i until $i+Z_i(S)-1$, i.e. $S[i\dots i+Z_i(S)-1]$
 - For every $i > 1$, r_i is the right-most endpoint of the Z-boxes that begin at or before i ;
 - i.e, r_i is the largest value of $j+Z_j(S)-1$ for all $1 < j \leq i$ such that $Z_j(S) > 0$



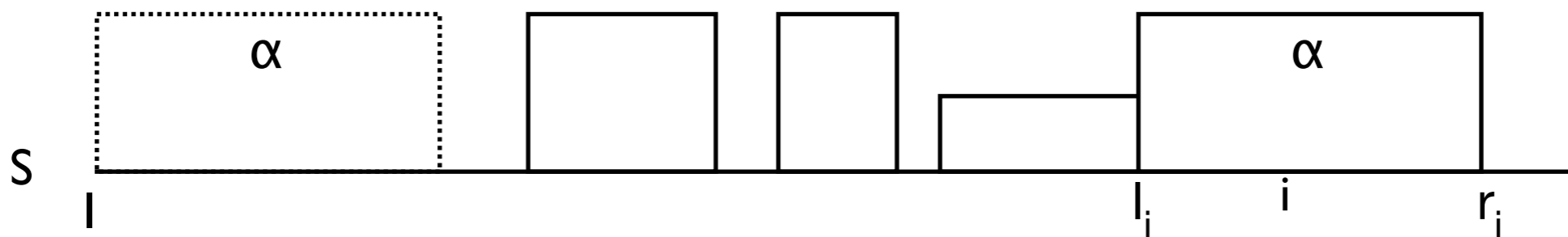


- The task: Compute $Z_i(S)$ in linear time, i.e. $O(|S|)$
- The notion of a **Z-box**
 - For every $i > 1$ with $Z_i(S) > 0$, define a Z-box to be the substring from i until $i+Z_i(S)-1$, i.e. $S[i\dots i+Z_i(S)-1]$
 - For every $i > 1$, r_i is the right-most endpoint of the Z-boxes that begin at or before i ;
 - i.e, r_i is the largest value of $j+Z_j(S)-1$ for all $1 < j \leq i$ such that $Z_j(S) > 0$



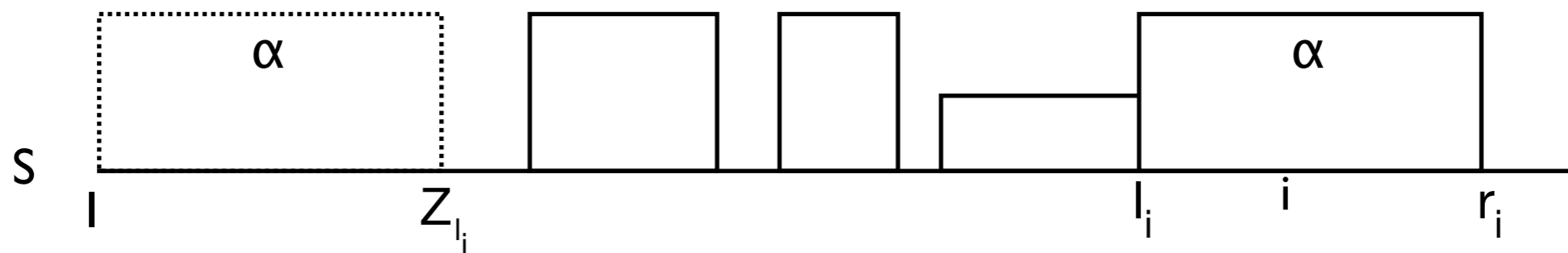


- The task: Compute $Z_i(S)$ in linear time, i.e. $O(|S|)$
- The notion of a **Z-box**
 - For every $i > 1$ with $Z_i(S) > 0$, define a Z-box to be the substring from i until $i+Z_i(S)-1$, i.e. $S[i\dots i+Z_i(S)-1]$
 - For every $i > 1$, r_i is the right-most endpoint of the Z-boxes that begin at or before i ;
 - i.e, r_i is the largest value of $j+Z_j(S)-1$ for all $1 < j \leq i$ such that $Z_j(S) > 0$





- The task: Compute $Z_i(S)$ in linear time, i.e. $O(|S|)$
- The notion of a **Z-box**
 - For every $i > 1$ with $Z_i(S) > 0$, define a Z-box to be the substring from i until $i+Z_i(S)-1$, i.e. $S[i\dots i+Z_i(S)-1]$
 - For every $i > 1$, r_i is the right-most endpoint of the Z-boxes that begin at or before i ;
 - i.e, r_i is the largest value of $j+Z_j(S)-1$ for all $1 < j \leq i$ such that $Z_j(S) > 0$





- We need to compute $Z_i(S)$, r_i and l_i for every $i > 2$
- In any iteration i , we only need r_j and l_j for $j=i-1$; i.e just r , l
- If we discover a new Z-box at i , set r to the end of that Z-box, which is the right-most position of any Z-box discovered so far
- Step 0
Find $Z_2(S)$ by comparing left to right $S[2..|S|]$ and $S[1..|S|]$ until a mismatch is found; $Z_2(S)$ is the length of that string. If $Z_2(S) > 0$ then set $r=r_2$ to $Z_2(S)+1$ and $l=l_2$, else $r=l=0$
- Induction hypothesis: we have correct $Z_i(S)$ for i up to $k-1 > 1$, r , l
 - Next, compute $Z_i(S)$ from the already computed Z values



- Simplest case: inclusion
- E.g. for $k=121$, we have $Z_2(S)\dots Z_{120}(S)$, and $r_{120}=130$, $l_{120}=100$
 - Thus: a substring of length 31 starting at position 100, matching $S[1..31]$
 - And: the substring of length 10 starting at 121 must match $S[22..31]$, so Z_{22} could help!
 - For example, if Z_{22} is 3, then Z_{121} must also be 3

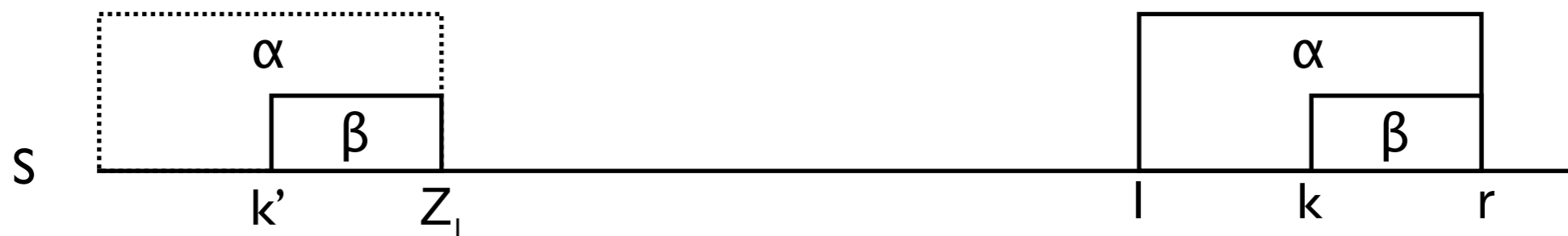


- Given $Z_i(S)$ for all $1 < i \leq k-1$, and the current values of $Z_k(S)$, r , and l ; compute the updated r and l
- Step 1:
 - if $k > r$, then find $Z_k(S)$ by comparing the characters starting at k to the characters starting at position 1 in S , until a mismatch is found. The length of the match is $Z_k(S)$. If $Z_k(S) > 0$, set $r = k + Z_k(S) - 1$, and $l = k$.



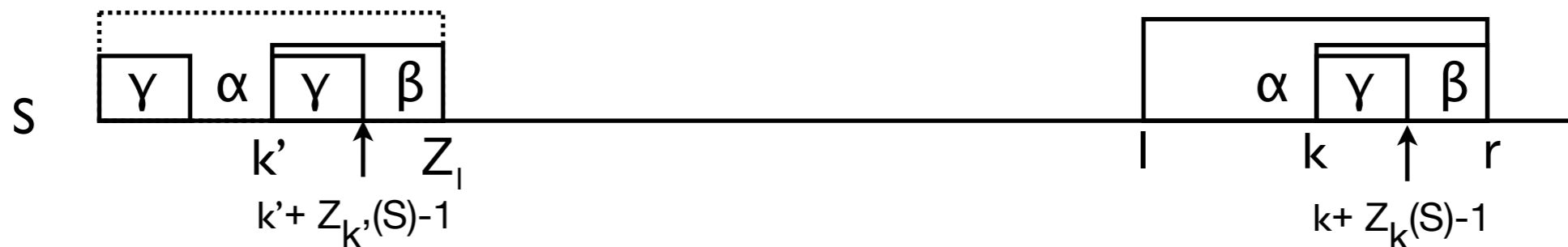
- Step 2

- If $k \leq r$, then position k is contained in a Z-box, and hence $S(k)$ is contained in a substring $S[l..r]$ (call it α) such that $l > 1$ and α matches a prefix of S .
- Therefore, character $S(k)$ also appears in position $k' = k - l + 1$ of S .
- By the same reasoning, the substring $S[k..r]$ (call it β) must match substring $S[k'..Z_l(S)]$. (Remember the example with $Z_{22}(S)$, $r=121!$)
- Hence, the substring at position k must match a prefix of S of length at least the *minimum* of $Z_{k'}(S)$ and $|\beta|$ (which is $r - k + 1$).



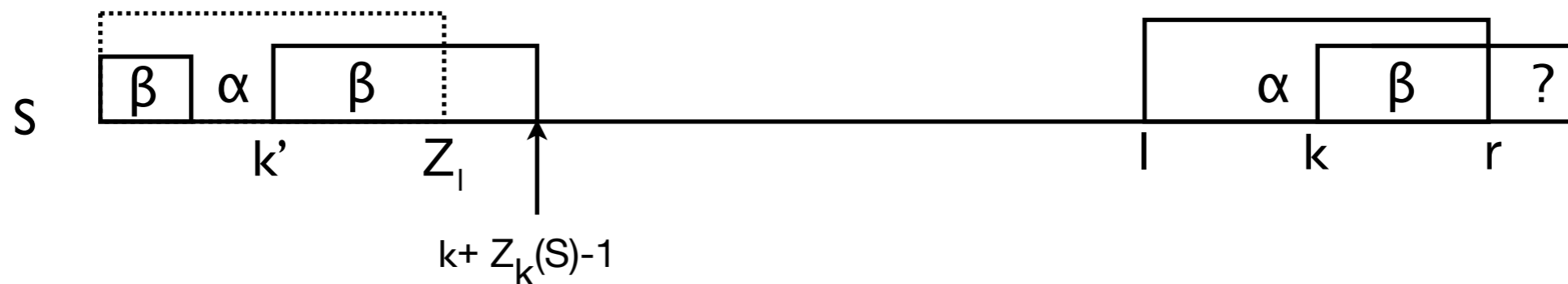


- Case 1: If $Z_{k'}(S) < |\beta|$
 - then position k is a Z-box (call it γ) contained within a larger Z-box
 - set $Z_k(S) = Z_{k'}(S)$ and leave r and l as they are





- Case 2: If $Z_{k'}(S) \geq |\beta|$
 - then the entire substring $S[k..r]$ must be a prefix of S and $Z_k(S) \geq |\beta| = r - k + 1$
 - However, $Z_k(S)$ may be strictly larger, so compare characters starting at $r + 1$ of S to the characters starting at $|\beta| + 1$ of S until a mismatch occurs (*Remember the second smart improvement over the naive method!*)
 - Say the mismatch is at $q \geq r + 1$. Then $Z_k(S) = q - k$, $r = q - 1$, and $l = k$





- “The algorithm computes all the $Z_i(S)$ values in $O(|S|)$ time”

The time is proportional to the number of iterations, $|S|$, plus the number of character comparisons. Each comparison is either a match or a mismatch. Each iteration that performs any character comparisons at all ends the first time it finds a mismatch; hence there are at most $|S|$ mismatches during the entire algorithm. To bound the number of mismatches, note first that $r_k \geq r_{k-1}$ for every iteration k . Now, let k be an iteration where $q > 0$ matches occur. Then r_k is set to $r_{k-1} + q$ at least. Finally, $r_k \leq |S|$ so the total number of matches that can occur during any execution of the algorithm is at most $|S|$.

- “Computing $Z_i(S)$ on $S=P\$T$ finds matches of P in T in $O(|T|)$ ”



- Like the naive method
 - Align P with T, check whether characters in P and T match
 - After the check is complete, P is shifted rightwards relative to T
- Smarter shifting
 - For an alignment, check whether P occurs in T scanning right-to-left in P
 - The **bad character** shift: shift right beyond the bad character
 - The **good suffix** shift: shift right using the match of the good suffix of P



- For any alignment of P against T, check P right-to-left

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T =	X	P	B	C	T	B	X	A	B	P	Q	X	C	T	B	P	Q
P =			T	P	A	B	X	A	B								
			1	2	3	4	5	6	7								

- For example,
 - $P(7)=T(9)$... but $P(3) \neq T(5)$
 - Upon a mismatch, shift P right relative to T
- The linear nature of the algorithm is in the shifts
 - Scanning right-to-left still yields an algorithm running in $O(nm)$ time



- For any alignment of P against T, check P right-to-left

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T =	X	P	B	C	T	B	X	A	B	P	Q	X	C	T	B	P	Q
P =			T	P	A	B	X	A	B								
			1	2	3	4	5	6	7								

✓

- For example,
 - $P(7)=T(9)$... but $P(3) \neq T(5)$
 - Upon a mismatch, shift P right relative to T
- The linear nature of the algorithm is in the shifts
 - Scanning right-to-left still yields an algorithm running in $O(nm)$ time



- For any alignment of P against T, check P right-to-left

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T =	X	P	B	C	T	B	X	A	B	P	Q	X	C	T	B	P	Q
P =			T	P	A	B	X	A	B								
			1	2	3	4	5	6	7								
								✓	✓								

- For example,
 - $P(7)=T(9)$... but $P(3) \neq T(5)$
 - Upon a mismatch, shift P right relative to T
- The linear nature of the algorithm is in the shifts
 - Scanning right-to-left still yields an algorithm running in $O(nm)$ time



- For any alignment of P against T, check P right-to-left

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T =	X	P	B	C	T	B	X	A	B	P	Q	X	C	T	B	P	Q
P =			T	P	A	B	X	A	B								
			1	2	3	4	5	6	7								
							✓	✓	✓								

- For example,
 - $P(7)=T(9)$... but $P(3) \neq T(5)$
 - Upon a mismatch, shift P right relative to T
- The linear nature of the algorithm is in the shifts
 - Scanning right-to-left still yields an algorithm running in $O(nm)$ time



- For any alignment of P against T, check P right-to-left

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T =	X	P	B	C	T	B	X	A	B	P	Q	X	C	T	B	P	Q
P =			T	P	A	B	X	A	B								
			1	2	3	4	5	6	7								
						✓	✓	✓	✓								

- For example,
 - $P(7)=T(9)$... but $P(3) \neq T(5)$
 - Upon a mismatch, shift P right relative to T
- The linear nature of the algorithm is in the shifts
 - Scanning right-to-left still yields an algorithm running in $O(nm)$ time



- The basic idea
 - Suppose the rightmost character in P is y , aligned to x in T with $x \neq y$
 - If x is in P , then we can shift P so that the rightmost x is below x in T
 - If x is not in P , then we can shift P completely beyond the x in T
- Possibly sublinear matching: not all characters in T may need to be compared



- Store the right-most position of each character

For each character x in the alphabet, let $R(x)$ be the rightmost position of x in P . $R(x)$ is defined to be 0 if x is not in P .

- The **bad character shift rule** makes use of R

Suppose for an alignment of P against T , the rightmost $n-i$ characters of P match against T , but the character at $P(i)$ is a mismatch with the character $T(k)$. Now, we can shift P right by $\max[1, i - R(T(k))]$ places; i.e. if the right-most occurrence in P of the character $T(k)$ is in position $j < i$ (possibly with $j=0$), then shift P so that the character j of P is below character k of T . Else, shift P by 1.



- Store the right-most position of each character

For each character x in the alphabet, let $R(x)$ be the rightmost position of x in P . $R(x)$ is defined to be 0 if x is not in P .

- The **bad character shift rule** makes use of R

Suppose for an alignment of P against T , the rightmost $n-i$ characters of P match against T , but the character at $P(i)$ is a mismatch with the character $T(k)$. Now, we can shift P right by $\max[1, i-R(T(k))]$ places; i.e. if the right-most occurrence in P of the character $T(k)$ is in position $j < i$ (possibly with $j=0$), then shift P so that the character j of P is below character k of T . Else, shift P by 1.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T =	X	P	B	C	T	B	X	A	B	P	Q	X	C	T	B	P	Q
P =			T	P	A	B	X	A	B								
			1	2	3	4	5	6	7								



- Store the right-most position of each character

For each character x in the alphabet, let $R(x)$ be the rightmost position of x in P . $R(x)$ is defined to be 0 if x is not in P .

- The **bad character shift rule** makes use of R

Suppose for an alignment of P against T , the rightmost $n-i$ characters of P match against T , but the character at $P(i)$ is a mismatch with the character $T(k)$. Now, we can shift P right by $\max[1, i - R(T(k))]$ places; i.e. if the right-most occurrence in P of the character $T(k)$ is in position $j < i$ (possibly with $j=0$), then shift P so that the character j of P is below character k of T . Else, shift P by 1.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T =	X	P	B	C	T	B	X	A	B	P	Q	X	C	T	B	P	Q
P =			T	P	A	B	X	A	B								
			1	2	3	4	5	6	7								
					×	✓	✓	✓	✓								



- Store the right-most position of each character

For each character x in the alphabet, let $R(x)$ be the rightmost position of x in P . $R(x)$ is defined to be 0 if x is not in P .

- The **bad character shift rule** makes use of R

Suppose for an alignment of P against T , the rightmost $n-i$ characters of P match against T , but the character at $P(i)$ is a mismatch with the character $T(k)$. Now, we can shift P right by $\max[1, i-R(T(k))]$ places; i.e. if the right-most occurrence in P of the character $T(k)$ is in position $j < i$ (possibly with $j=0$), then shift P so that the character j of P is below character k of T . Else, shift P by 1.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
T =	X	P	B	C	T	B	X	A	B	P	Q	X	C	T	B	P	Q	
P =			T	P	A	B	X	A	B									
			1	2	3	4	5	6	7									
					×	✓	✓	✓	✓									

$R(\text{"T"}) = 1$



- Store the right-most position of each character

For each character x in the alphabet, let $R(x)$ be the rightmost position of x in P . $R(x)$ is defined to be 0 if x is not in P .

- The **bad character shift rule** makes use of R

Suppose for an alignment of P against T , the rightmost $n-i$ characters of P match against T , but the character at $P(i)$ is a mismatch with the character $T(k)$. Now, we can shift P right by $\max[1, i - R(T(k))]$ places; i.e. if the right-most occurrence in P of the character $T(k)$ is in position $j < i$ (possibly with $j=0$), then shift P so that the character j of P is below character k of T . Else, shift P by 1.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T =	X	P	B	C	T	B	X	A	B	P	Q	X	C	T	B	P	Q
P =			T	P	A	B	X	A	B								
			1	2	3	4	5	6	7								
					×	✓	✓	✓	✓								
					T	P	A	B	X	A	B						

$R(\text{"T"}) = 1$



- The basic idea:
 - Given the character $T(k)$ against which P mismatches,
 - Take the good suffix t of P , i.e. the part that matched against T
 - Look in P for the left-most copy t' of t , such that the character k' to the immediate left of t' differs from $T(k)$; *else the shift would yield the same mismatch!*
 - Then, shift P to the right such that t' is below the matching t in T .



- The basic idea:

- Given the character $T(k)$ against which P mismatches,
- Take the good suffix t of P , i.e. the part that matched against T
- Look in P for the left-most copy t' of t , such that the character k' to the immediate left of t' differs from $T(k)$; *else the shift would yield the same mismatch!*
- Then, shift P to the right such that t' is below the matching t in T .

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T =	P	R	S	T	A	B	S	T	U	B	A	B	V	Q	X	R	S	T
P =	Q	C	A	B	D	A	B	D	A	B								
	1	2	3	4	5	6	7	8	9	10								



- The basic idea:

- Given the character $T(k)$ against which P mismatches,
- Take the good suffix t of P , i.e. the part that matched against T
- Look in P for the left-most copy t' of t , such that the character k' to the immediate left of t' differs from $T(k)$; *else the shift would yield the same mismatch!*
- Then, shift P to the right such that t' is below the matching t in T .

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T =	P	R	S	T	A	B	S	T	U	B	A	B	V	Q	X	R	S	T
P =	Q	C	A	B	D	A	B	D	A	B								
	1	2	3	4	5	6	7	8	9	10								
								×	✓	✓								



- The basic idea:

- Given the character $T(k)$ against which P mismatches,
- Take the good suffix t of P , i.e. the part that matched against T
- Look in P for the left-most copy t' of t , such that the character k' to the immediate left of t' differs from $T(k)$; *else the shift would yield the same mismatch!*
- Then, shift P to the right such that t' is below the matching t in T .

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T =	P	R	S	T	A	B	S	T	U	B	A	B	V	Q	X	R	S	T
P =	Q	C	A	B	D	A	B	D	A	B								
	1	2	3	4	5	6	7	8	9	10								
								x	✓	✓								



- The basic idea:

- Given the character $T(k)$ against which P mismatches,
- Take the good suffix t of P , i.e. the part that matched against T
- Look in P for the left-most copy t' of t , such that the character k' to the immediate left of t' differs from $T(k)$; *else the shift would yield the same mismatch!*
- Then, shift P to the right such that t' is below the matching t in T .

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T =	P	R	S	T	A	B	S	T	U	B	A	B	V	Q	X	R	S	T
P =	Q	C	A	B	D	A	B	D	A	B								
	1	2	3	4	5	6	7	8	9	10								
								x	✓	✓								



- The basic idea:

- Given the character $T(k)$ against which P mismatches,
- Take the good suffix t of P , i.e. the part that matched against T
- Look in P for the left-most copy t' of t , such that the character k' to the immediate left of t' differs from $T(k)$; *else the shift would yield the same mismatch!*
- Then, shift P to the right such that t' is below the matching t in T .

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T =	P	R	S	T	A	B	S	T	U	B	A	B	V	Q	X	R	S	T
P =	Q	C	A	B	D	A	B	D	A	B								
	1	2	3	4	5	6	7	8	9	10								
								x	✓	✓								



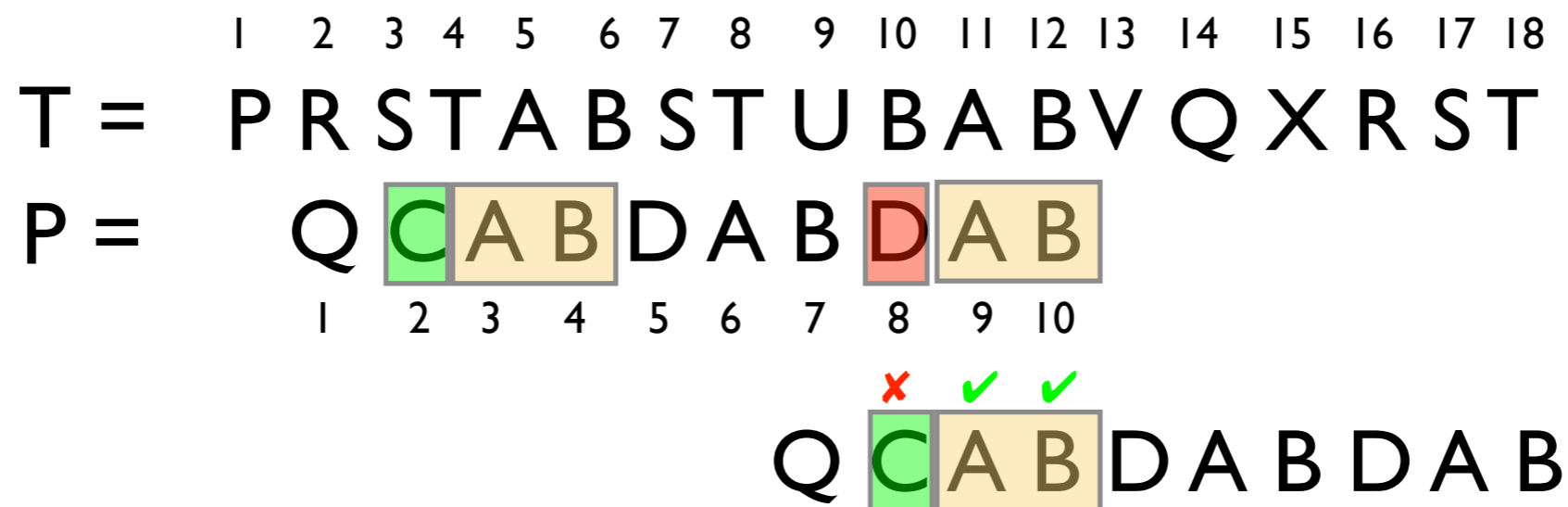
- The basic idea:

- Given the character $T(k)$ against which P mismatches,
- Take the good suffix t of P , i.e. the part that matched against T
- Look in P for the left-most copy t' of t , such that the character k' to the immediate left of t' differs from $T(k)$; *else the shift would yield the same mismatch!*
- Then, shift P to the right such that t' is below the matching t in T .

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T =	P	R	S	T	A	B	S	T	U	B	A	B	V	Q	X	R	S	T
P =	Q	C	A	B	D	A	B	D	A	B								
	1	2	3	4	5	6	7	8	9	10								
								x	✓	✓								

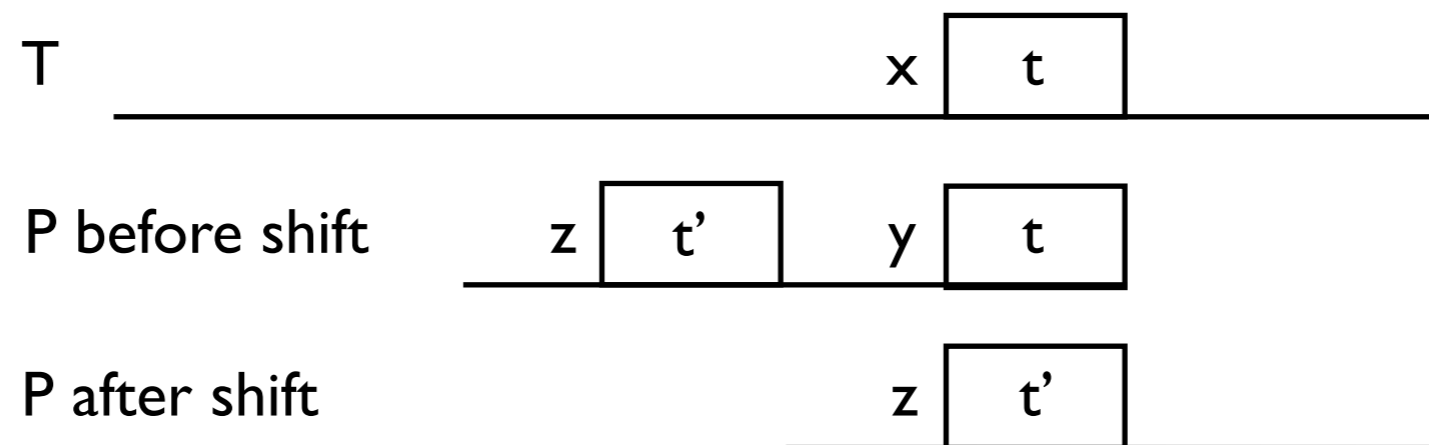


- The basic idea:
 - Given the character $T(k)$ against which P mismatches,
 - Take the good suffix t of P , i.e. the part that matched against T
 - Look in P for the left-most copy t' of t , such that the character k' to the immediate left of t' differs from $T(k)$; *else the shift would yield the same mismatch!*
 - Then, shift P to the right such that t' is below the matching t in T .





Suppose for a given alignment of P and T , a substring t of T matches a suffix of P , but a mismatch occurs at the next comparison to the left. Then find, if it exists, the right-most copy t' of t in P such that t' is not a suffix of P and the character to the left of t' in P differs from the character to the left of t in P . Shift P to the right so that the substring t' in P is below substring t in T . If t' does not exist, then shift the left end of P past the left end of t in T by the least amount so that a prefix of the shifted pattern matches a suffix of t in T . If no such shift is possible, then shift P by n places to the right. If an occurrence of P is found, then shift P by the least amount so that a proper prefix of the shifted P matches a suffix of the occurrence of P in T . If no such shift is possible, then shift P by n places, past t in T .

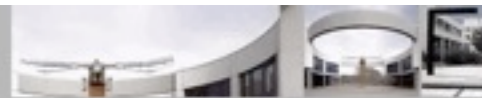




- We need some preprocessing for the good suffix rule
 - We need to compute the positions of copies of suffixes of P
 - whereby a copy differs from the suffix in its immediate left character

- Definition

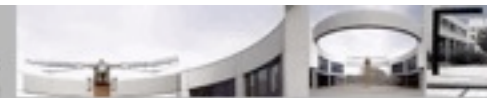
For each i , $L(i)$ is the largest position less than n such that string $P[i..n]$ matches a suffix of $P[1..L(i)]$. Let $L(i)$ be zero if there is no position satisfying the conditions. For each i , $L'(i)$ is the largest position less than n such that string $P[i..n]$ matches a suffix of $P[1..L'(i)]$ and such that the character preceding the suffix is not equal to $P(i-1)$. Let $L'(i)$ be 0 if there is no position satisfying the conditions.



- We need some preprocessing for the good suffix rule
 - We need to compute the positions of copies of suffixes of P
 - whereby a copy differs from the suffix in its immediate left character
- Definition

For each i , $L(i)$ is the largest position less than n such that string $P[i..n]$ matches a suffix of $P[1..L(i)]$. Let $L(i)$ be zero if there is no position satisfying the conditions. For each i , $L'(i)$ is the largest position less than n such that string $P[i..n]$ matches a suffix of $P[1..L'(i)]$ and such that the character preceding the suffix is not equal to $P(i-1)$. Let $L'(i)$ be 0 if there is no position satisfying the conditions.

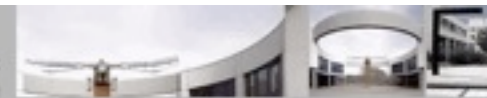
$P =$ **C A B D A B D A B**
 1 2 3 4 5 6 7 8 9



- We need some preprocessing for the good suffix rule
 - We need to compute the positions of copies of suffixes of P
 - whereby a copy differs from the suffix in its immediate left character
- Definition

For each i , $L(i)$ is the largest position less than n such that string $P[i..n]$ matches a suffix of $P[1..L(i)]$. Let $L(i)$ be zero if there is no position satisfying the conditions. For each i , $L'(i)$ is the largest position less than n such that string $P[i..n]$ matches a suffix of $P[1..L'(i)]$ and such that the character preceding the suffix is not equal to $P(i-1)$. Let $L'(i)$ be 0 if there is no position satisfying the conditions.

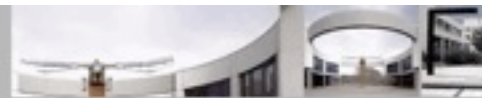
$P =$ **C A B D A B D A B** $L(8)=6$
 1 2 3 4 5 6 7 8 9



- We need some preprocessing for the good suffix rule
 - We need to compute the positions of copies of suffixes of P
 - whereby a copy differs from the suffix in its immediate left character
- Definition

For each i , $L(i)$ is the largest position less than n such that string $P[i..n]$ matches a suffix of $P[1..L(i)]$. Let $L(i)$ be zero if there is no position satisfying the conditions. For each i , $L'(i)$ is the largest position less than n such that string $P[i..n]$ matches a suffix of $P[1..L'(i)]$ and such that the character preceding the suffix is not equal to $P(i-1)$. Let $L'(i)$ be 0 if there is no position satisfying the conditions.

$P =$ **C A B D A B D A B** $L(8)=6$ $L'(8)=3$
 1 2 3 4 5 6 7 8 9



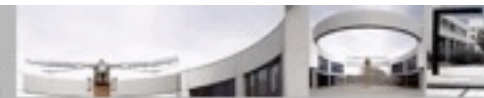
- Computing $L'(i)$

- For string P , $N_j(P)$ is the length of the longest suffix of the substring $P[1...j]$ that is also a suffix of the full string P .

$P =$ **C A B D A B D A B**
 1 2 3 4 5 6 7 8 9

- We can compute $N_i(S)$ from $Z_i(S)$

- Recall that $Z_i(S)$ is the length of the longest substring of S that starts at i and is a *prefix* of S
- $N_i(S)$ is the reverse of Z : if P^r is the reverse of P , then $N_j(P) = Z_{n-j+1}(P^r)$
- Hence we can obtain the values for N using the linear algorithm for Z



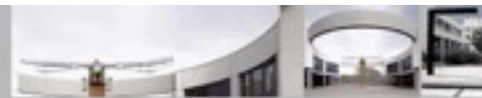
- Computing $L'(i)$

- For string P , $N_j(P)$ is the length of the longest suffix of the substring $P[1...j]$ that is also a suffix of the full string P .

$P =$ **C A B D A B D A B** $N_3(P)=2$
 1 2 3 4 5 6 7 8 9

- We can compute $N_i(S)$ from $Z_i(S)$

- Recall that $Z_i(S)$ is the length of the longest substring of S that starts at i and is a *prefix* of S
- $N_i(S)$ is the reverse of Z : if P^r is the reverse of P , then $N_j(P)=Z_{n-j+1}(P^r)$
- Hence we can obtain the values for N using the linear algorithm for Z



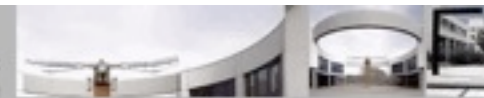
- Computing $L'(i)$

- For string P , $N_j(P)$ is the length of the longest suffix of the substring $P[1...j]$ that is also a suffix of the full string P .

$P =$ C A B D A B D A B $N_3(P)=2$
 1 2 3 4 5 6 7 8 9

- We can compute $N_i(S)$ from $Z_i(S)$

- Recall that $Z_i(S)$ is the length of the longest substring of S that starts at i and is a *prefix* of S
- $N_i(S)$ is the reverse of Z : if P^r is the reverse of P , then $N_j(P) = Z_{n-j+1}(P^r)$
- Hence we can obtain the values for N using the linear algorithm for Z



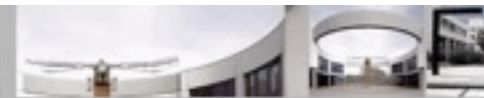
- Computing $L'(i)$

- For string P , $N_j(P)$ is the length of the longest suffix of the substring $P[1...j]$ that is also a suffix of the full string P .

$P =$ **C A B D A B D A B** $N_3(P)=2$
 1 2 3 4 5 6 7 8 9

- We can compute $N_i(S)$ from $Z_i(S)$

- Recall that $Z_i(S)$ is the length of the longest substring of S that starts at i and is a *prefix* of S
- $N_i(S)$ is the reverse of Z : if P^r is the reverse of P , then $N_j(P)=Z_{n-j+1}(P^r)$
- Hence we can obtain the values for N using the linear algorithm for Z



- Computing $L'(i)$

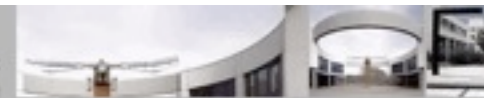
- For string P , $N_j(P)$ is the length of the longest suffix of the substring $P[1...j]$ that is also a suffix of the full string P .

$P =$ **C A B D A B D A B**
 1 2 3 4 5 6 7 8 9

$N_3(P)=2$ $N_6(P)=5$

- We can compute $N_i(S)$ from $Z_i(S)$

- Recall that $Z_i(S)$ is the length of the longest substring of S that starts at i and is a *prefix* of S
- $N_i(S)$ is the reverse of Z : if P^r is the reverse of P , then $N_j(P)=Z_{n-j+1}(P^r)$
- Hence we can obtain the values for N using the linear algorithm for Z



- Computing $L'(i)$

- For string P , $N_j(P)$ is the length of the longest suffix of the substring $P[1...j]$ that is also a suffix of the full string P .

$P =$ C **A B D A B** D A B $N_3(P)=2$ $N_6(P)=5$
 1 2 3 4 5 6 7 8 9

- We can compute $N_i(S)$ from $Z_i(S)$

- Recall that $Z_i(S)$ is the length of the longest substring of S that starts at i and is a *prefix* of S
- $N_i(S)$ is the reverse of Z : if P^r is the reverse of P , then $N_j(P) = Z_{n-j+1}(P^r)$
- Hence we can obtain the values for N using the linear algorithm for Z



- Z-based Boyer-Moore for obtaining $L'(i)$ from $N_i(P)$

for $i := 1$ to n do $L'(i) := 0$

for $j := 1$ to $n-1$ do

begin

$i := n - N_j(P) + 1$

$L'(i) := j$

end

- Intuition

- We have computed the lengths of the longest suffixes as $N_j(P)$
- Cycle over P right-to-left, looking at where the longest suffixes start
- Assign to $L'(i)$ the largest index j such that $N_j(P) = |P[i..n]| = (n-i+1)$
- Those $L'(i)$ for which there is no such index have been initialized to 0.

The final preprocessing detail ...





- Let $l'(i)$ denote the longest suffix of $P[i..n]$ that is also a prefix of P , if one exists. If none exists, let $l'(i)$ be zero.



- Let $l'(i)$ denote the longest suffix of $P[i..n]$ that is also a prefix of P , if one exists. If none exists, let $l'(i)$ be zero.
- Once more, all the preprocessing and rules:
 - Bad character rule: given a mismatch on x in T , shift P right to align with an x in P (if any)
 - Compute $R(x)$, the right-most occurrence of x in P
 - Good suffix rule: shift P right to a copy of the matching suffix but with a different character to its immediate left
 - Use $Z_j(P)$ to compute $N_j(P)$, the length of the longest suffix of $P[1..j]$ that is a suffix of P
 - Use $N_j(P)$ to compute $L'(i)$, the largest position less than n s.t. $P[i..n]$ matches a suffix of $P[1..L'(i)]$
 - Compute $l'(i)$, to deal with the case when we have $L'(i) = 0$ or when an occurrence of P is found



[Preprocessing stage]

Given the pattern P

Compute $L'(i)$ and $I'(i)$ for each position i of P

and compute $R(x)$ for each character $x \in \Sigma$

[Search stage]

$k := n$

while $k \leq m$ do

$i := n$

$h := k$

 while $i > 0$ and $P(i) = T(h)$ do

$i := i - 1$

$h := h - 1$

 if $i = 0$ then

 report an occurrence of P in T ending at position k

$k := k + n - I'(2)$

 else

 shift P (increase k) by the maximum amount determined by the bad character rule and the good suffix rule



- Improve search by making smart “jumps” upon mismatches
- Jumps based on computation of the Z-function
 - At each position i in a String, the score $Z_i(S)$ indicates the size of the Z-box at that position,
 - i.e. the length of the string starting at i which is a prefix of S (of that size)
- Boyer-Moore algorithm
 - Bad character, good suffix rules



- Knuth-Morris-Pratt algorithm
 - Left-to-right matching of P against T
 - Smart jumps: look at suffix of $P[1..i]$ which is also a prefix, jump beyond the prefix (like the good suffix role in B-M)



- Basic idea

- Shift smarter than the naive method does

1 2 3 4 5 6 7 8 9
P= A B C X A B C D E

- A mismatch with P(8) means we can shift 4 places
- Deduction on P alone: no need to know T, or how P and T are aligned

- Complexity of the algorithm

- The algorithm is linear, not -possibly- sublinear like Boyer-Moore
- Extension: the Aho-Corasick algorithm for matching sets of patterns



- Basic idea

- Shift smarter than the naive method does

1 2 3 4 5 6 7 8 9
P= A B C X A B C D E

- A mismatch with P(8) means we can shift 4 places
 - Deduction on P alone: no need to know T, or how P and T are aligned

- Complexity of the algorithm

- The algorithm is linear, not -possibly- sublinear like Boyer-Moore
 - Extension: the Aho-Corasick algorithm for matching sets of patterns



- Basic idea

- Shift smarter than the naive method does

1 2 3 4 5 6 7 8 9
P= A B C X A B C D E
↑

- A mismatch with P(8) means we can shift 4 places
- Deduction on P alone: no need to know T, or how P and T are aligned

- Complexity of the algorithm

- The algorithm is linear, not -possibly- sublinear like Boyer-Moore
- Extension: the Aho-Corasick algorithm for matching sets of patterns



- Basic idea

- Shift smarter than the naive method does

1 2 3 4 5 6 7 8 9
P= A B C X A B C D E

- A mismatch with P(8) means we can shift 4 places
- Deduction on P alone: no need to know T, or how P and T are aligned

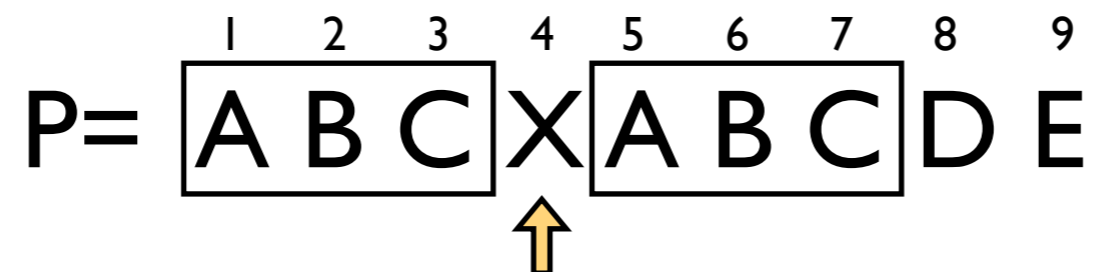
- Complexity of the algorithm

- The algorithm is linear, not -possibly- sublinear like Boyer-Moore
- Extension: the Aho-Corasick algorithm for matching sets of patterns



- Basic idea

- Shift smarter than the naive method does



- A mismatch with P(8) means we can shift 4 places
- Deduction on P alone: no need to know T, or how P and T are aligned

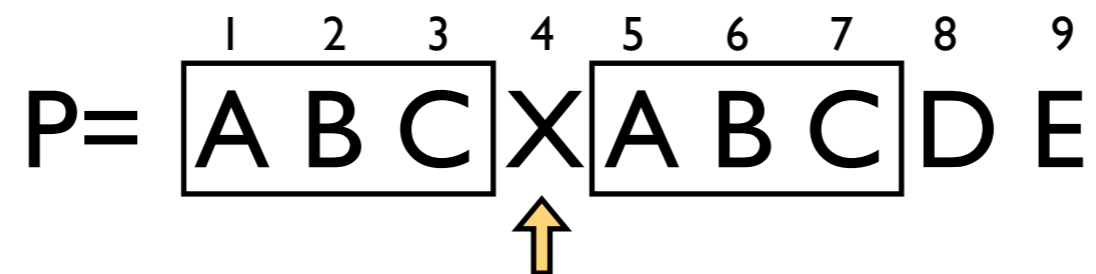
- Complexity of the algorithm

- The algorithm is linear, not -possibly- sublinear like Boyer-Moore
- Extension: the Aho-Corasick algorithm for matching sets of patterns



- Basic idea

- Shift smarter than the naive method does



- A mismatch with P(8) means we can shift 4 places (*like good suffix rule!*)
- Deduction on P alone: no need to know T, or how P and T are aligned

- Complexity of the algorithm

- The algorithm is linear, not -possibly- sublinear like Boyer-Moore
- Extension: the Aho-Corasick algorithm for matching sets of patterns



- Definition

For each position i in P , define $sp_i(P)$ to be the length of the longest proper suffix of $P[1\dots i]$ that matches a prefix of P .

1 2 3 4 5 6 7 8 9 10 11
P = A B C A E A B C A B D

- Optimization

For each position i in P , define $sp'_i(P)$ to be the length of the longest proper suffix of $P[1\dots i]$ that matches a prefix of P , with the added condition that characters $P(i+1)$ and $P(sp'_i+1)$ are unequal.



- Definition

For each position i in P , define $sp_i(P)$ to be the length of the longest proper suffix of $P[1\dots i]$ that matches a prefix of P .

	1	2	3	4	5	6	7	8	9	10	11
P=	A	B	C	A	E	A	B	C	A	B	D
sp	0	0	0	1	0	1	2	3	4	2	0

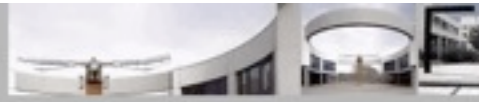
- Optimization

For each position i in P , define $sp'_i(P)$ to be the length of the longest proper suffix of $P[1\dots i]$ that matches a prefix of P , with the added condition that characters $P(i+1)$ and $P(sp'_i+1)$ are unequal.



- Alignment of P and T, left-to-right matching
- The shift rule:

For any alignment of P and T, if the first mismatch (comparing from left to right) occurs in position $i+1$ of P and position k of T, then shift P to the right (relative to T) so that $P[1..sp_i']$ aligns with $T[k-sp_i'..k-1]$. In other words, shift P exactly $i+1-(sp_i'+1)=i-sp_i'$ places to the right, so that character $sp_i'+1$ of P will align with character k in T. In the case that an occurrence of P has been found (no mismatch), shift P by $n-sp_i'$ places.



- Preprocessing using the Z values

Position $j > 1$ maps to i if $i = j + Z_j(P) - 1$. That is, j maps to i if i is the right end of a Z-box starting at j .

- Z-based Knuth-Morris-Pratt

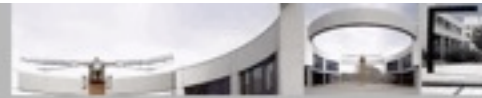
```
for i := 1 to n do
```

```
     $sp_i' := 0;$ 
```

```
for j := n downto 2 do
```

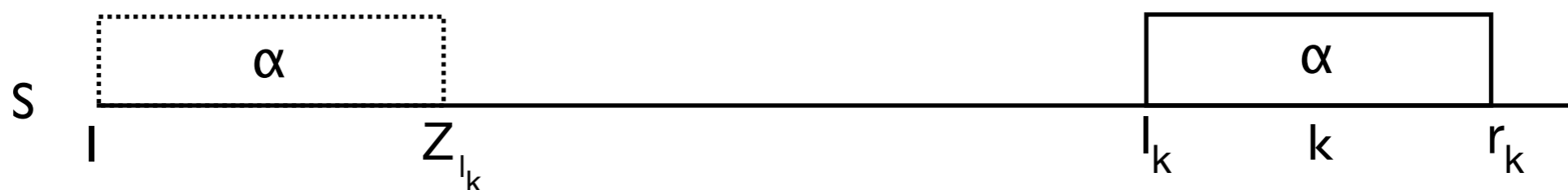
```
     $i := j + Z_j(P) - 1;$ 
```

```
     $sp_i' := Z_j$ 
```



- Preprocessing using the Z values

Position $j > 1$ maps to i if $i = j + Z_j(P) - 1$. That is, j maps to i if i is the right end of a Z-box starting at j .



- Z-based Knuth-Morris-Pratt

for $i := 1$ to n do

$sp_i' := 0;$

for $j := n$ downto 2 do

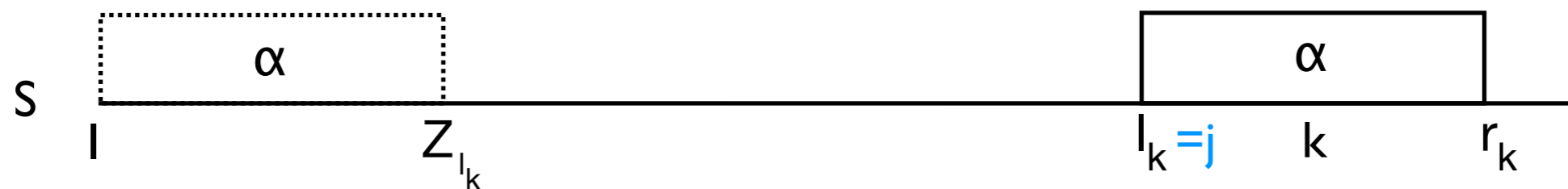
$i := j + Z_j(P) - 1;$

$sp_i' := Z_j$



- Preprocessing using the Z values

Position $j > 1$ maps to i if $i = j + Z_j(P) - 1$. That is, j maps to i if i is the right end of a Z-box starting at j .



- Z-based Knuth-Morris-Pratt

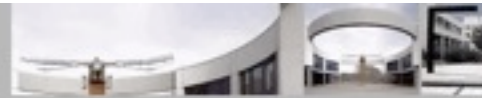
```
for i := 1 to n do
```

```
     $sp_i' := 0;$ 
```

```
for j := n downto 2 do
```

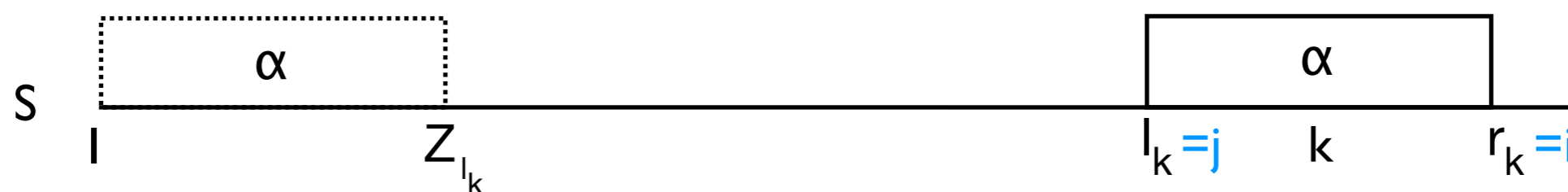
```
     $i := j + Z_j(P) - 1;$ 
```

```
     $sp_i' := Z_j$ 
```



- Preprocessing using the Z values

Position $j > 1$ maps to i if $i = j + Z_j(P) - 1$. That is, j maps to i if i is the right end of a Z-box starting at j .



- Z-based Knuth-Morris-Pratt

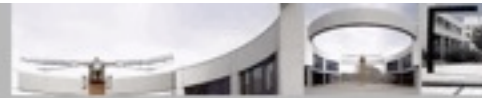
```
for i := 1 to n do
```

```
     $sp_i' := 0;$ 
```

```
for j := n downto 2 do
```

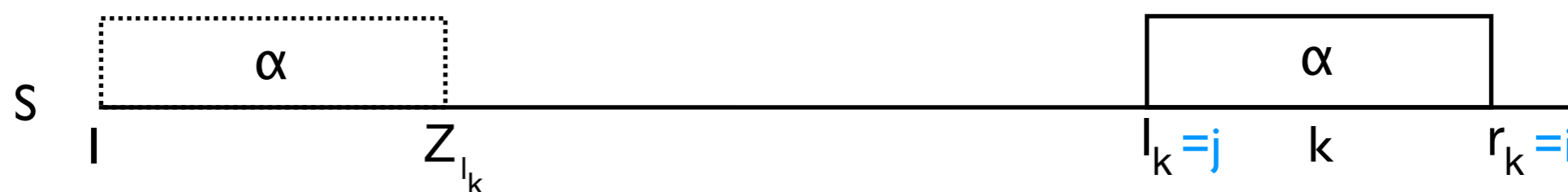
```
     $i := j + Z_j(P) - 1;$ 
```

```
     $sp_i' := Z_j$ 
```



- Preprocessing using the Z values

Position $j > 1$ maps to i if $i = j + Z_j(P) - 1$. That is, j maps to i if i is the right end of a Z-box starting at j .



- Z-based Knuth-Morris-Pratt

```
for i := 1 to n do
```

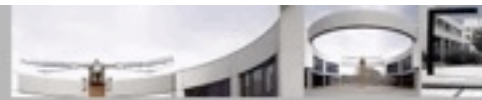
```
     $sp_i' := 0;$ 
```

```
for j := n downto 2 do
```

```
     $i := j + Z_j(P) - 1;$ 
```

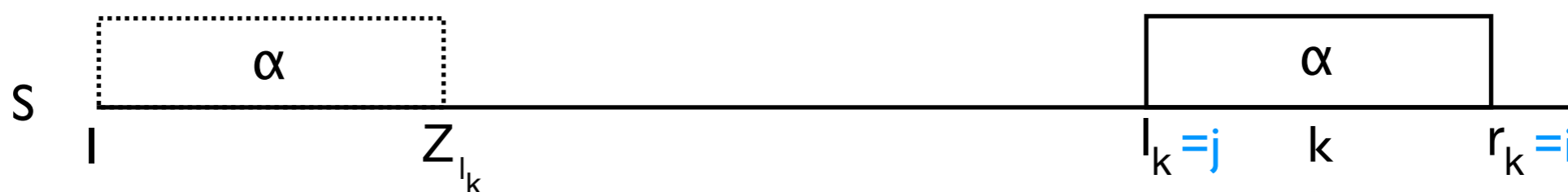
```
     $sp_i' := Z_j$ 
```

$sp_i'(P)$ is the length of the longest proper suffix of $P[1...i]$ i.e. the length of the Z-box that starts at j (the suffix)



- Preprocessing using the Z values

Position $j > 1$ maps to i if $i = j + Z_j(P) - 1$. That is, j maps to i if i is the right end of a Z-box starting at j .



- Z-based Knuth-Morris-Pratt

```
for i := 1 to n do
```

```
     $sp_i' := 0;$ 
```

```
for j := n downto 2 do
```

```
     $i := j + Z_j(P) - 1;$ 
```

```
     $sp_i' := Z_j$ 
```

$sp_i'(P)$ is the length of the longest proper suffix of $P[1...i]$ i.e. the length of the Z-box that starts at j (the suffix)



- Preliminaries

- Shifts through pointers: p points into P , c points into T
- For each position i from 1 to $n+1$, define the failure function $F'(i)$ to be $sp'_{i-1} + 1$ (and define $F(i) = sp_{i-1} + 1$); let sp'_0 and sp_0 be 0.

- The algorithm

preprocess P to find $F'(k) = sp'_{k-1} + 1$ for k from 1 to $n+1$

$c := 1$;

$p := 1$;

while $c + (n-p) \leq m$ do

 while $P(p) = T(c)$ and $p \leq n$

$p := p+1$;

$c := c+1$;

 if $p = n+1$ then

 report an occurrence of P starting at position $c-n$ of T

 if $p = 1$ then $c := c+1$

$p := F'(p)$



- Exact matching problem
 - Naive method compares character by character, single shift of P against T
 - Optimization through smarter shifting; base information for smarter shifting is provided by Z-boxes, computable in linear time
 - Boyer-Moore algorithm, Knuth-Morris-Pratt algorithm
- Exercises
 - a. Show how to compute Z_i stepwise for $i > 1$ (using the notion of Z-boxes) for the following strings:
 - i. AABCAABXAAZ
 - ii. ABCDXABCYABDXY
 - b. Apply the Knuth-Morris-Pratt algorithm to find occurrences of ABXYABXZ in XABXYABXYABXZABXZABXYABXZA