Data Structures and Algorithms

Lecture 6 – String Algorithms

Miao Zhang

Introduction

- Numbers as key values: are data items of constant size, and can be compared in constant time.
- In real applications, text processing is more important than the processing of numbers
- We need different structures for strings than for numeric keys.

Applications

- Word processors.
- Virus scanning.
- Text information retrieval systems. (Lexis, Nexis)
- Digital libraries.
- Natural language processing.
- Specialized databases.
- Computational molecular biology.
- Web search engines.

Concepts

- STRING: a string is traditionally a sequence of characters
- The length of string
- char： basic unit of string
- The difference between a character array and a string：

the string is terminated with a special character ‘\0’.

Substrings

Substring

P[i..j] to denote the substring of P from index i to index j,

$$P[i..j] = P[i]P[i+1]\cdots P[j].$$

Main string

Ex，“eij” is the substring of “beijing” . “beijing” is the main string.

The position of the character is its sequence number in the string.

The position of substring is the position of its first character in the main string.

Ex，The position number of ‘n’ in the string "beijing" is 6 。

Ex， The position number of "eij" in the string "beijing" is 2。

The value of two strings are equal.

Ex，The string “bei jing” is not equal to the string “beijing”.

Blank character string ： "␣␣␣"

$\phi \neq$ "␣"

Operations

>>	Extracts characters from a stream and inserts them into a string. Characters are copied until a whitespace or the end of the string is encountered.
<<	Inserts a string into a stream.
=	Assigns the string on the right to the string object on the left.
+=	Appends a copy of the string on the right to the string object on the left.
+	Returns a string that is the concatenation of the two string operands.
[]	Implements array-subscript notation, as in *name[x]*. A reference to the character in the x position is returned.

The storage structure and Class definition

- Sequential storage
  - A special tail mark ‘\0’

- class String

The function of get substring:

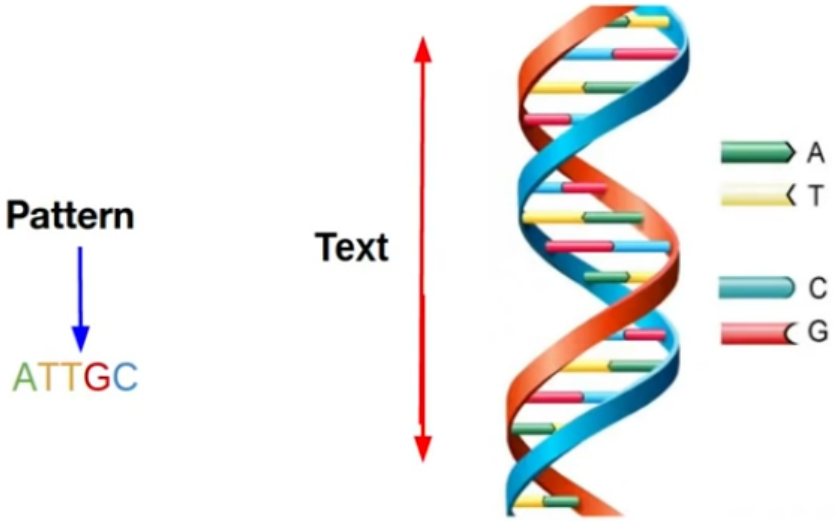
```
String  s1 = "value-";
s2 = s1.Substr(2,3);
```

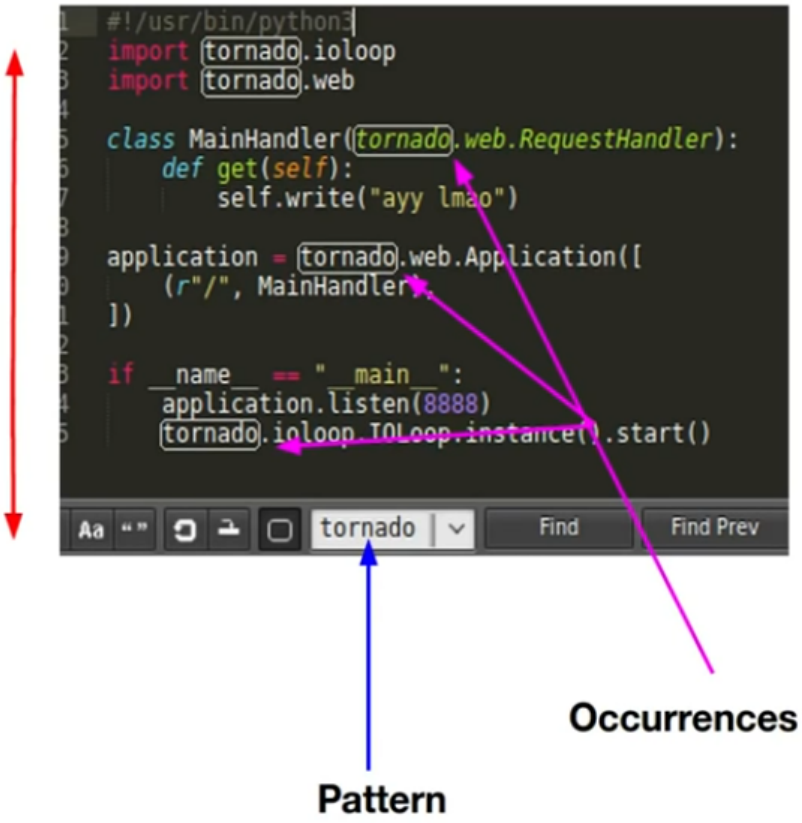
String Matching

What *is* String Matching?

Definition:

The problem of finding occurrence(s) of a **pattern** string within another string or body of **text**.





Pattern matching of String

- Find a substring in String T, which is equal to P. T is the Target，P is the Pattern.
- Finding the position of P in the string T is called Pattern matching.

Naïve Approach

Naive approach

- Using bruteforce
- Try every possible fit in the text
- We will use DNA as our text for examples.
- The alphabet of DNA is {A,C,G,T}

Naïve Approach

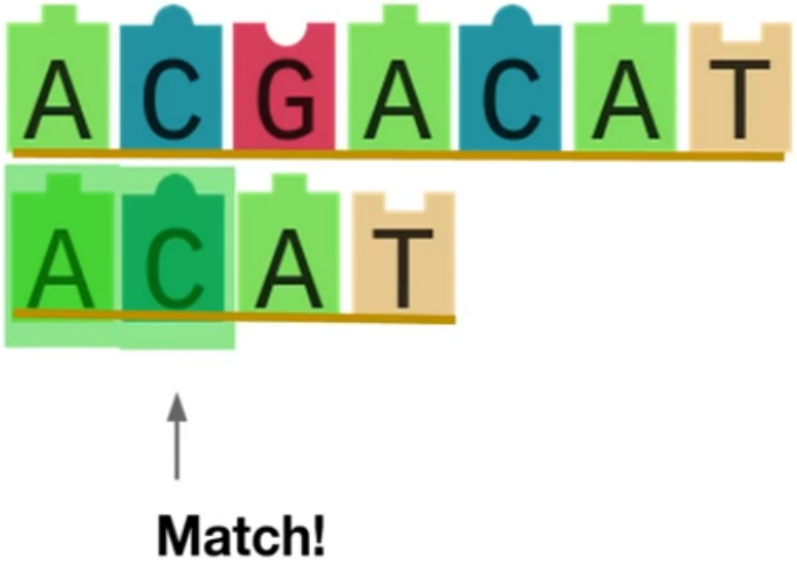
Using Naive

Check the first letter of pattern against the first letter of the text

Naïve Approach

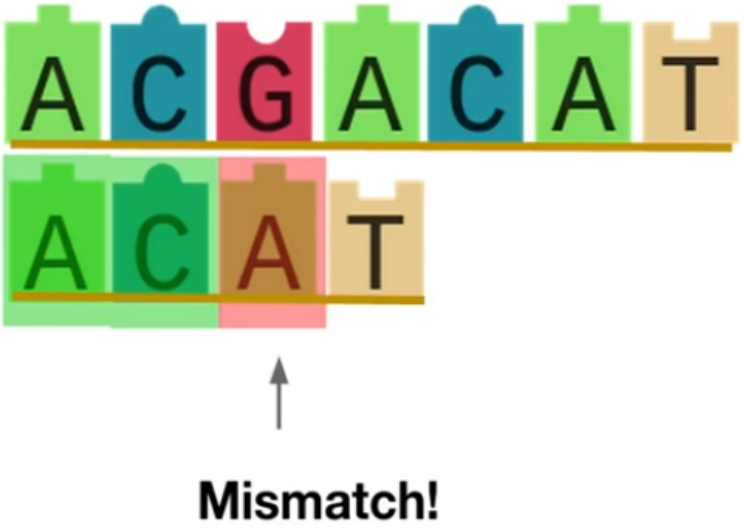
Now compare the next letter in pattern against next letter in the text...

Naïve Approach



Now continue checking...

Naïve Approach

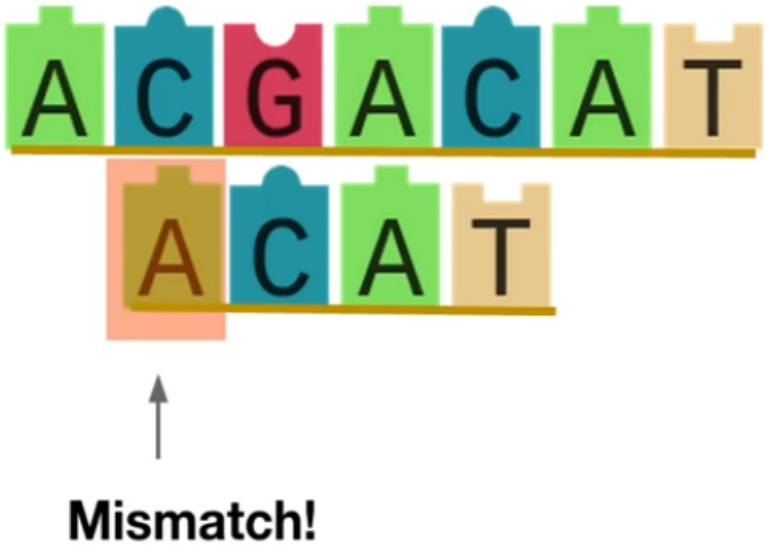


What do we do now?

Naïve Approach

Shift our **pattern** to the right by **one** letter

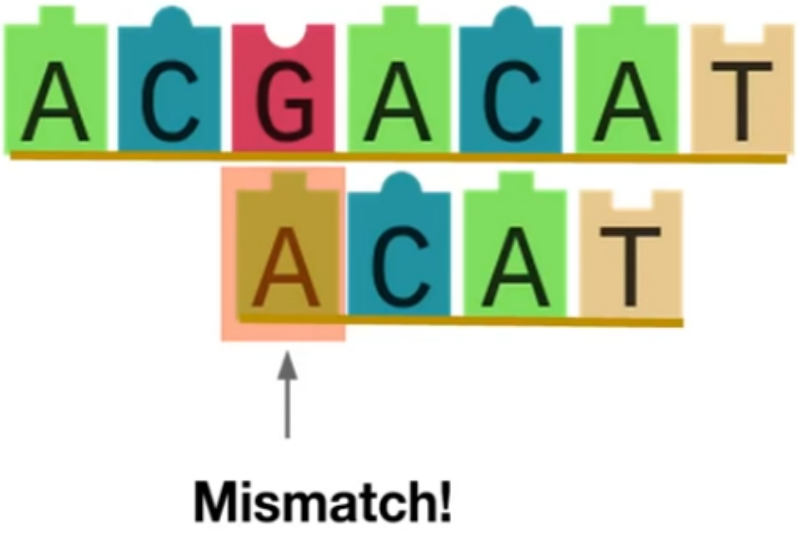
Naïve Approach



We will have to **shift** again!

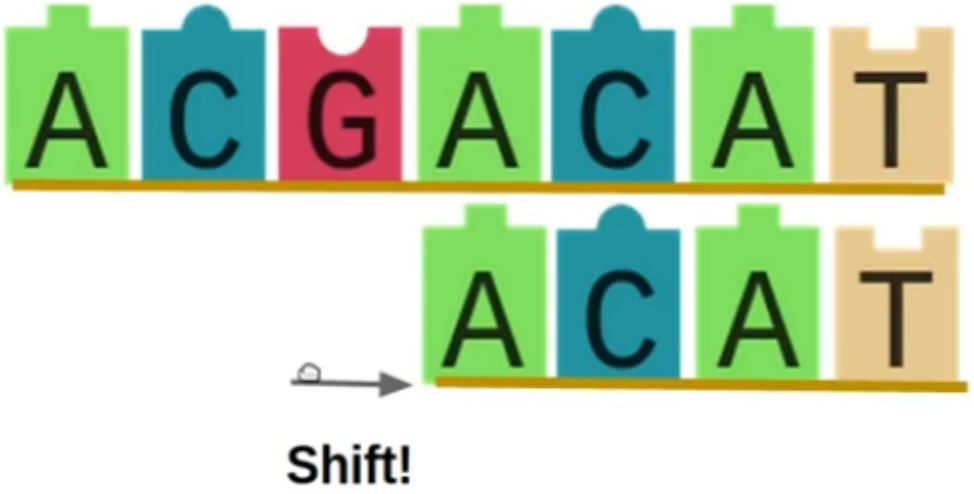
Naïve Approach

Naïve Approach

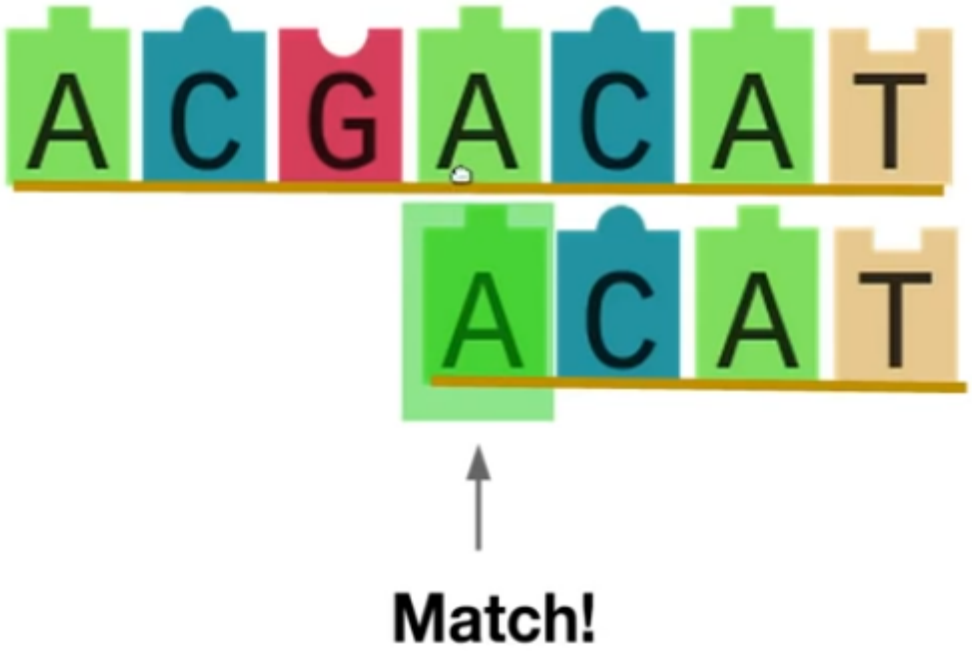


We do the same as before...

Naïve Approach



Naïve Approach



Naïve Approach

Pattern found!

We managed to find our pattern in the text!

Native pattern matching algorithm

```
int NaiveStrMatching(String* T, String* P) {
int p=0,t=0;
int plen =P.length(); int tlen=T.length();
if(tlen<plen) return -1;
while(p<plen && t<tlen){
    if(T[t]==P[p]) {p++; t++;}
    else {t=t-p+1;p=0;} }
 if(p>=plen) return  t-p; //Matching successfully
else  return -1;    }
```

Complexity of Naïve Approach

Complexity

The performance of a string matching algorithm is determined by how many **comparisons** are performed.

Let ***m*** be the ***length*** of the pattern and let ***n*** be the ***length*** of the searchable text.

The complexity is **O(nm)** for naive string matching.

Now the question is...
Can we do better ? ;)

Knuth-Morris-Pratt Algorithm

KMP

for string matching

KMP Algorithm

Knuth-Morris-Pratt Algorithm (KMP)

It uses a pre-generated table called a prefix table.

A prefix table will allow us to skip certain comparisons.

Let's not worry about how this is generated for now.

Example:

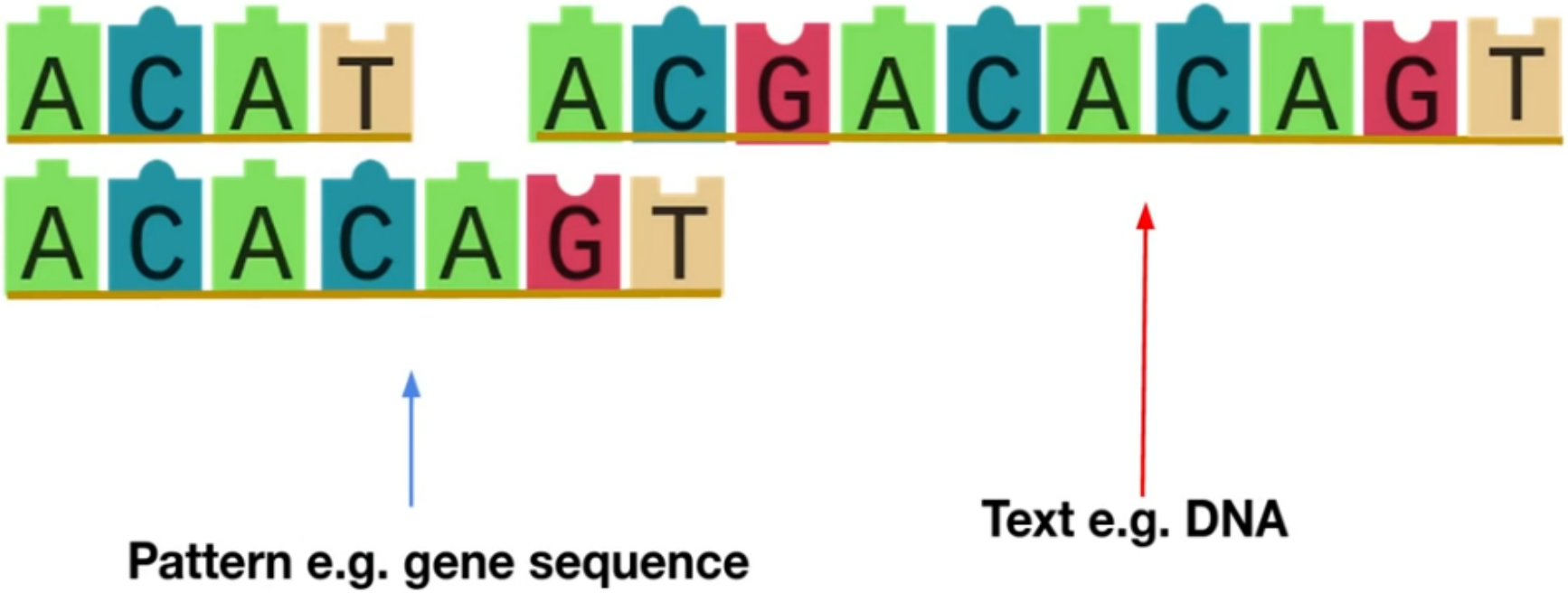
i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	0	0	1	2	3	0	0

Pattern: ACACAGT

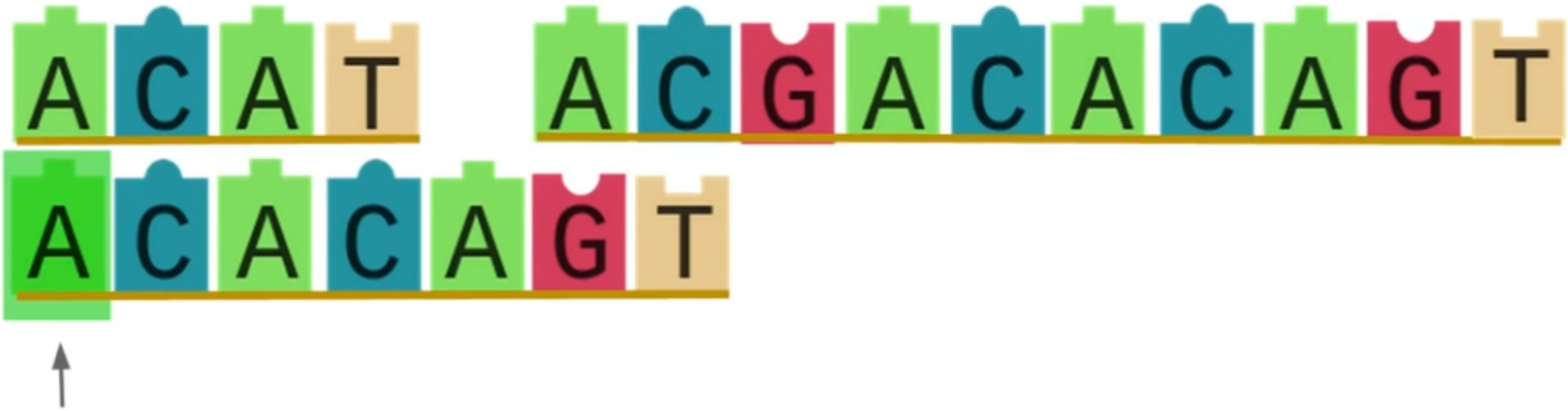
Text: ACAT ACGACACAGT

KMP Algorithm

Now using KMP...

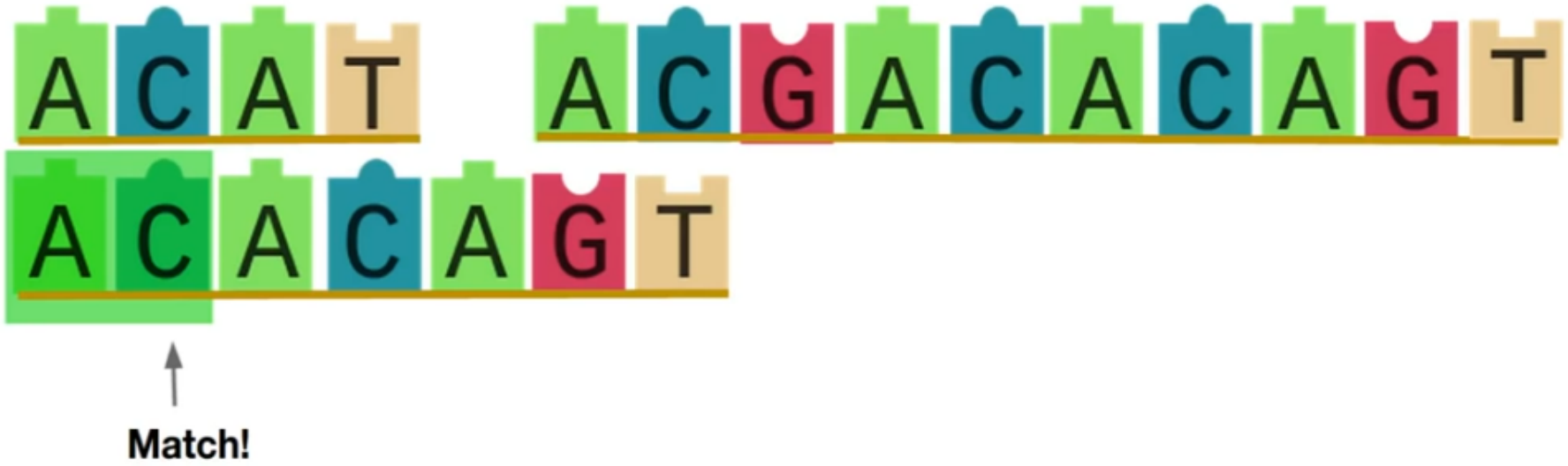


KMP Algorithm

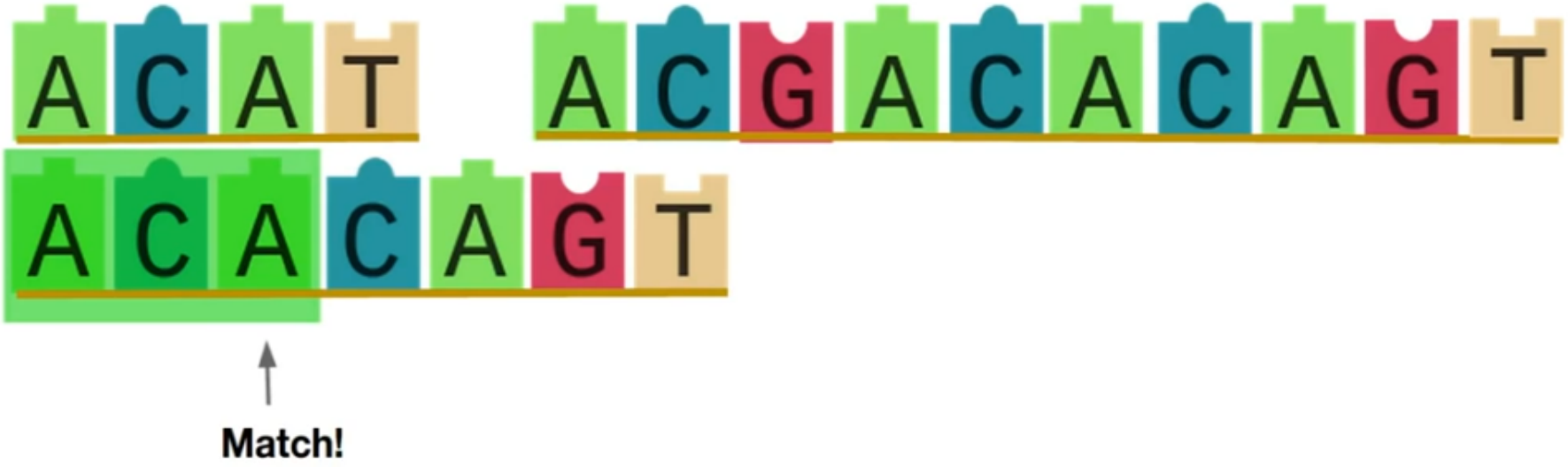


Check the first letter of pattern against the first letter of the text just like the Naive method and continue...

KMP Algorithm

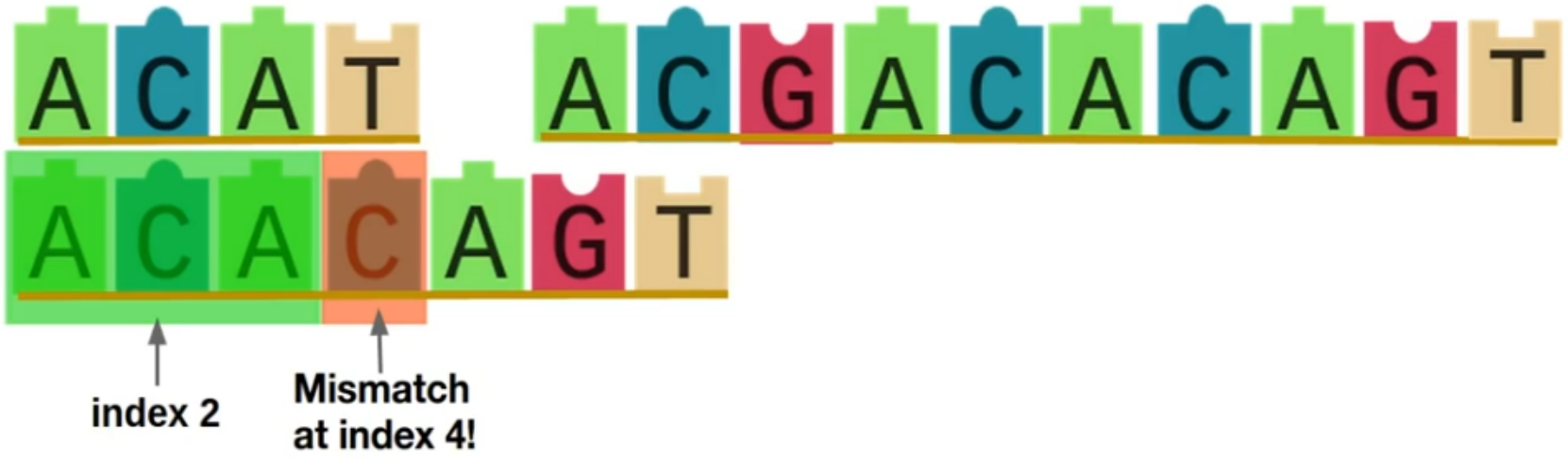


KMP Algorithm



KMP Algorithm

i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	0	0	1	2	3	0	0

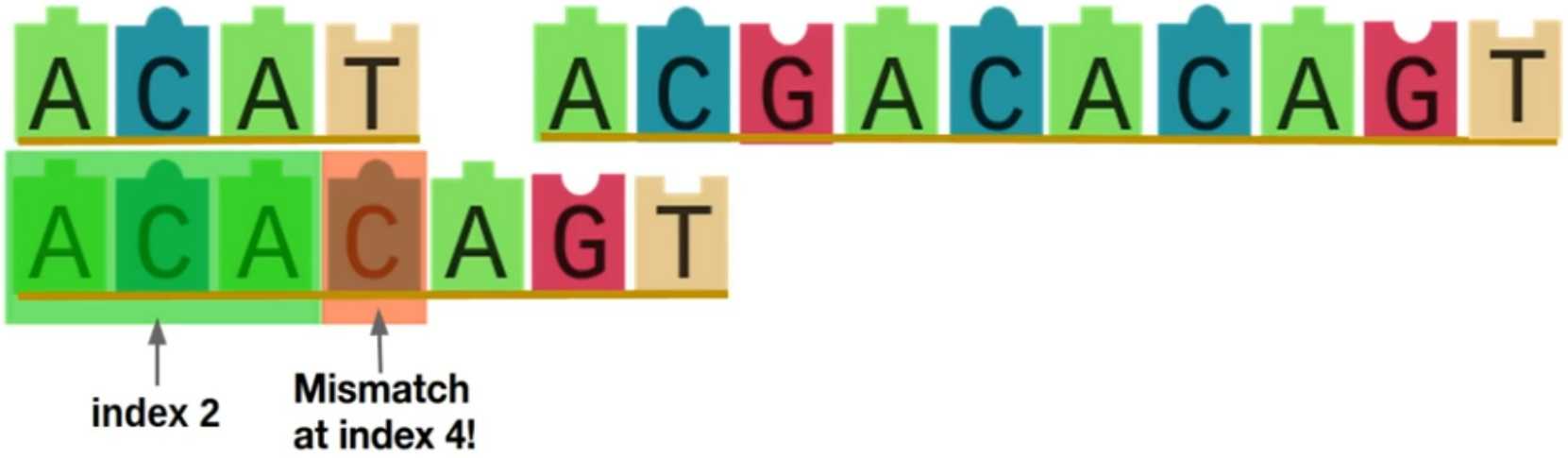


What do we do now?

Check prefix table for the **value** when i = 4, and align the pattern so that the new index is in the same position as the old. This is visualised by shifting right by 2.

KMP Algorithm

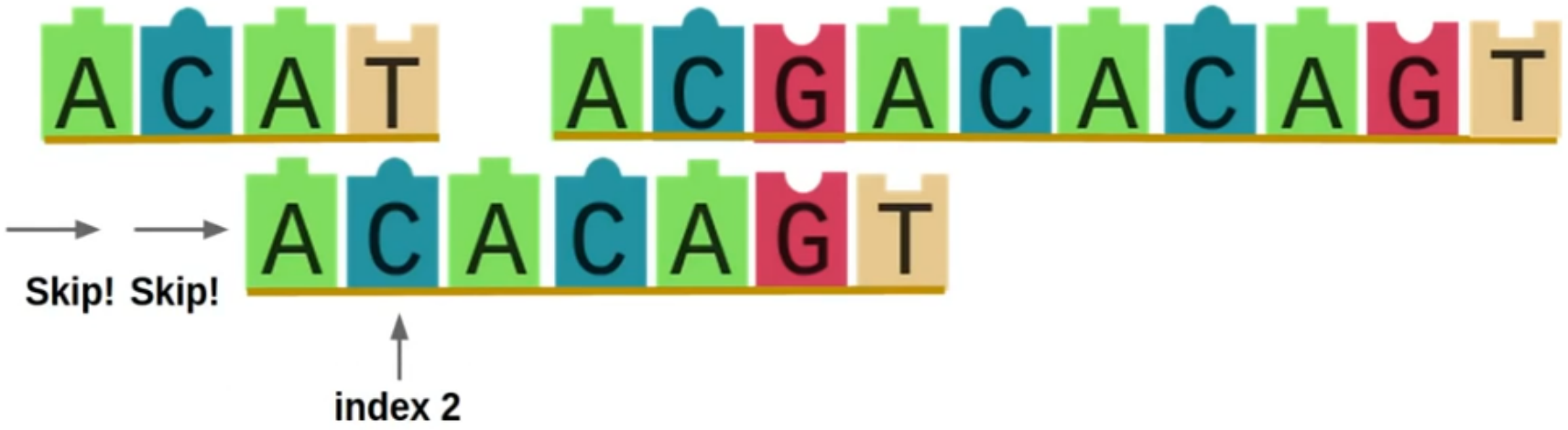
i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	0	0	1	2	3	0	0



What do we do now?

Check prefix table for the **value** when i = 4, and align the pattern so that the new index is in the same position as the old. This is visualised by shifting right by 2.

KMP Algorithm



Now we check index 2 against the text...

KMP Algorithm

i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	0	0	1	2	3	0	0

What do we do now?

Check prefix table for the **value** at pattern[2]. Since it is 0, we continue like we did for naive. We shift to the right.

KMP Algorithm

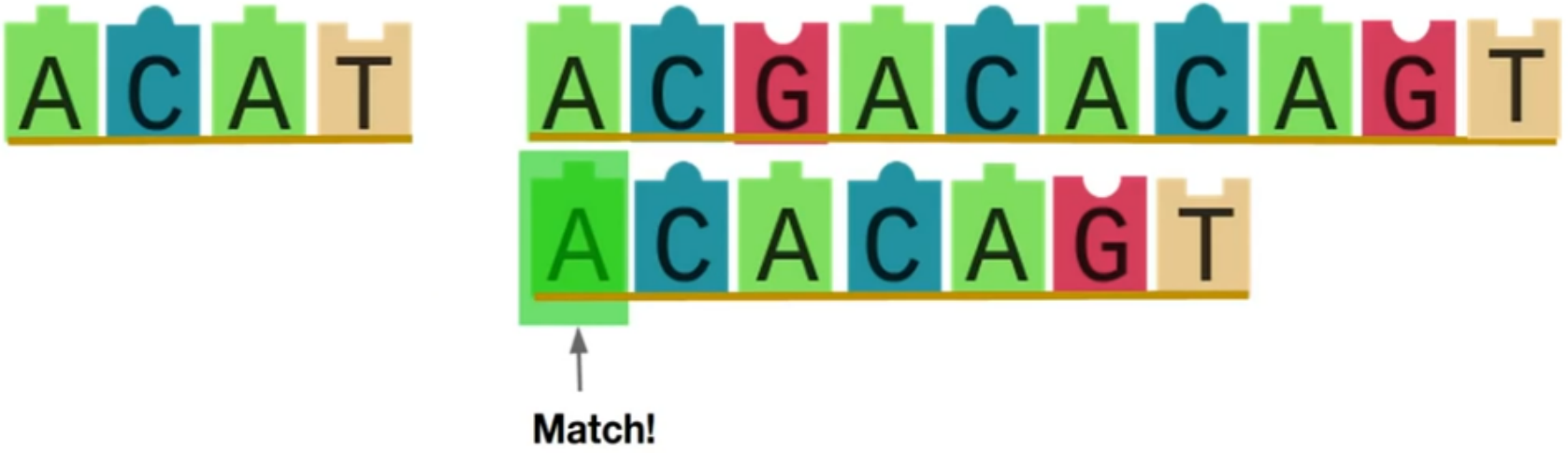
i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	0	0	1	2	3	0	0

What do we do now?

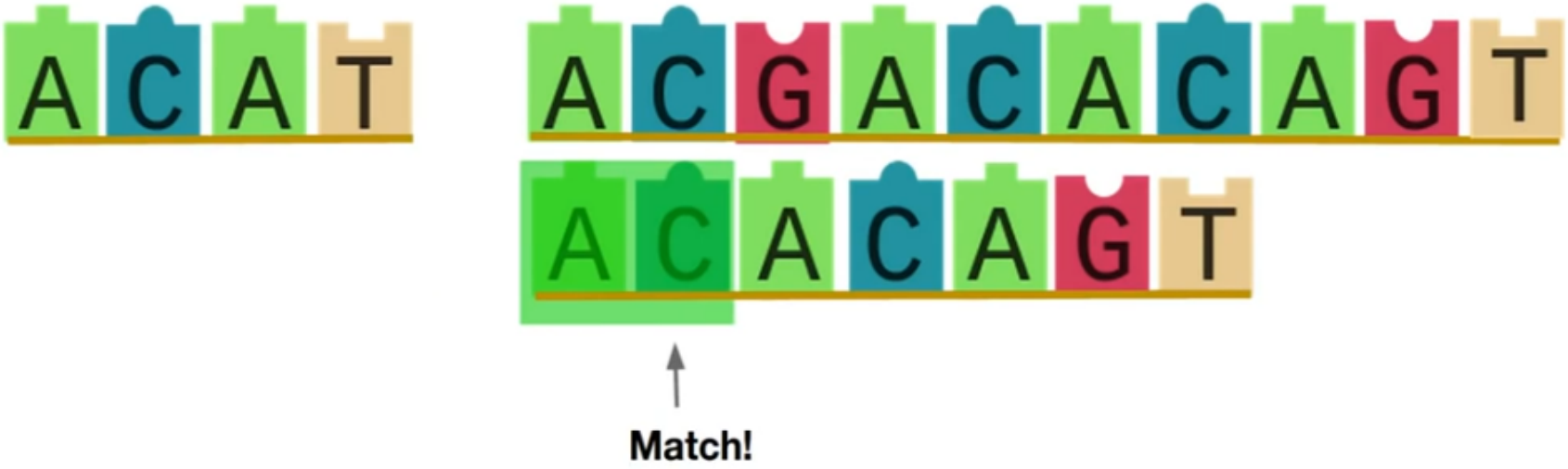
Check prefix table for the **value** at pattern[1] which is 0 again so we continue.

KMP Algorithm

KMP Algorithm



KMP Algorithm



KMP Algorithm

i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	0	0	1	2	3	0	0

What do we do now?

Check prefix table for the **value** at the pattern[3] and skip to that index in our pattern which is 1 in this case.

KMP Algorithm

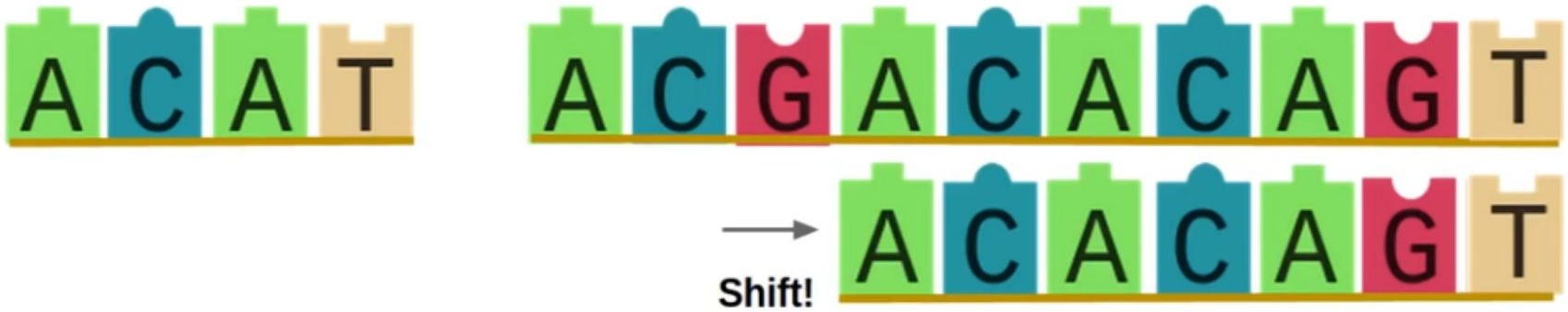
KMP Algorithm

i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	0	0	1	2	3	0	0

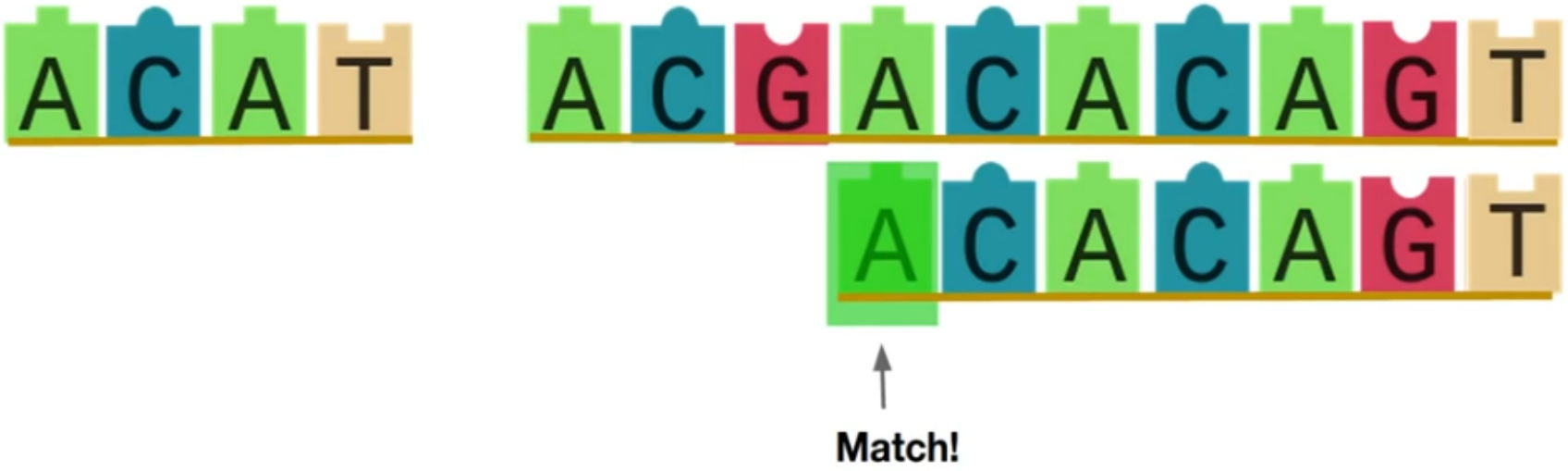
What do we do now?

Check prefix table for the **value** at pattern[1] which is 0 again so we continue.

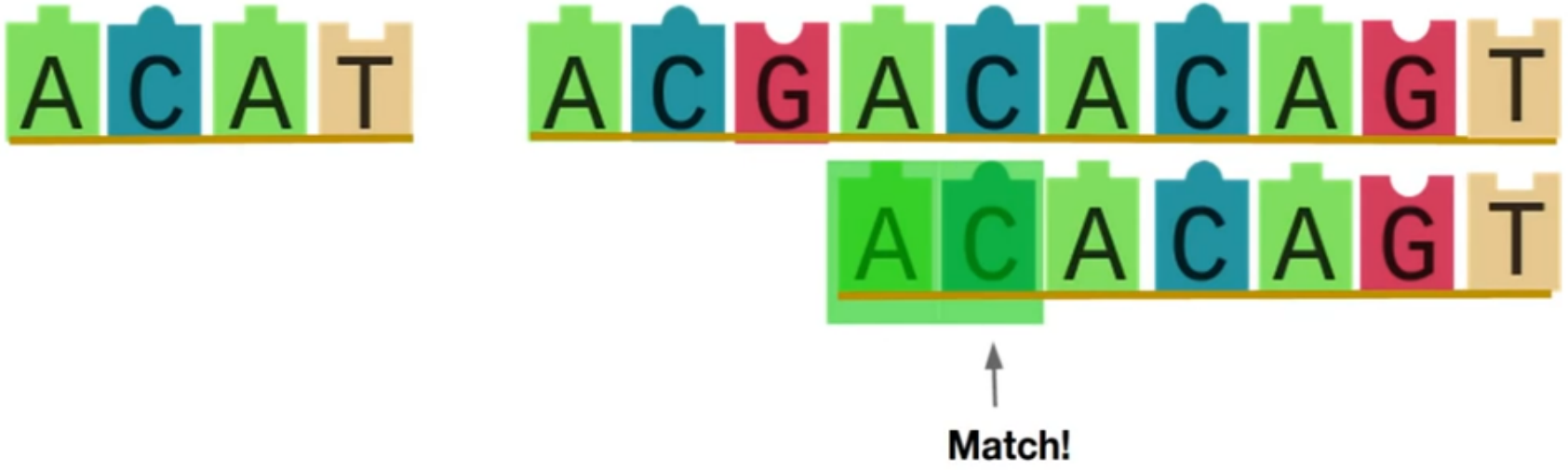
KMP Algorithm



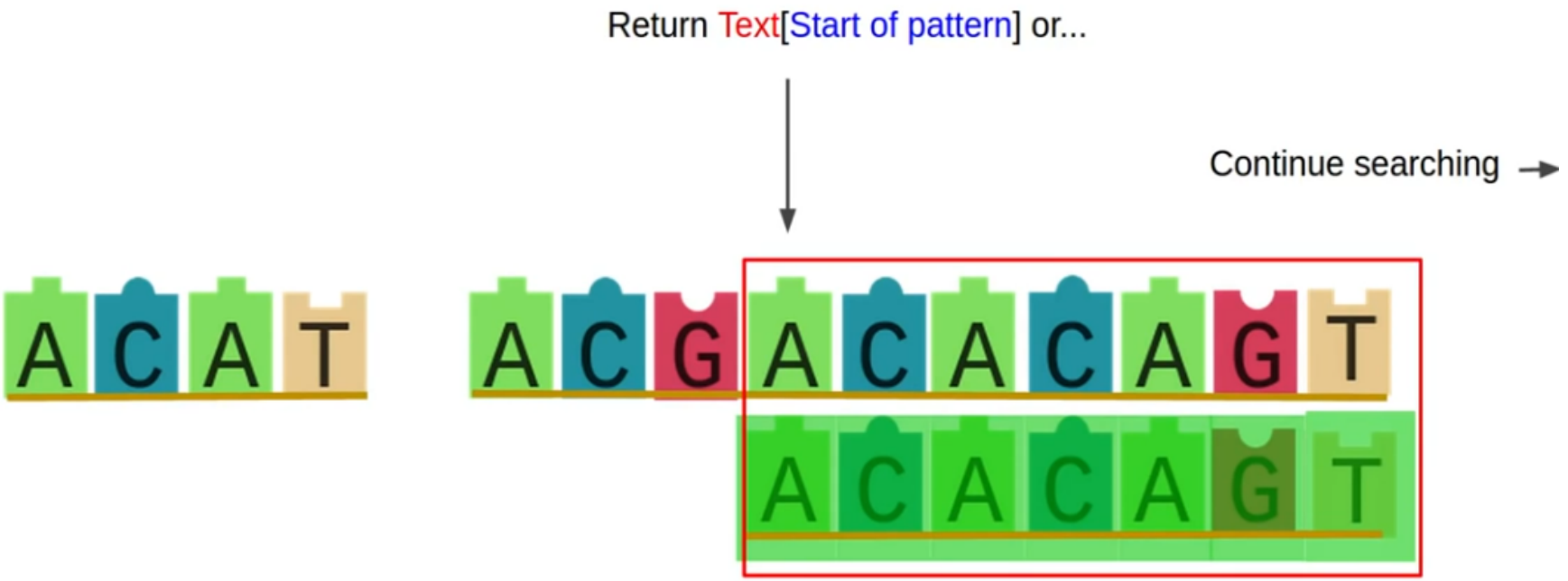
KMP Algorithm



KMP Algorithm



KMP Algorithm



Pattern found!

We managed to find our pattern in the text!

KMP Algorithm

Generation of the prefix table

Definitions we first need to know:

Proper prefix:

All the characters in a string with one or more cut off the end.
"A" "Ayy" "Ayyy" "Ayyyy" are the proper prefixes of the string "Ayyyyy".

Proper suffix:

All the characters in a string, with one or more cut off the beginning.
"maoo", "aoo", "oo", "o" are the proper suffixes of the string "Lmaoo".

KMP Algorithm

So what are the values in the prefix table exactly?

The values in the prefix table are:

The **length** of the longest proper **prefix** that matches a proper **suffix** in the **same subpattern**.

Don't worry if you don't understand!

We shall now illustrate how the values are calculated from the pattern...

KMP Algorithm

We initialise the prefix table.

i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]							

KMP Algorithm

i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]							

At the first letter:

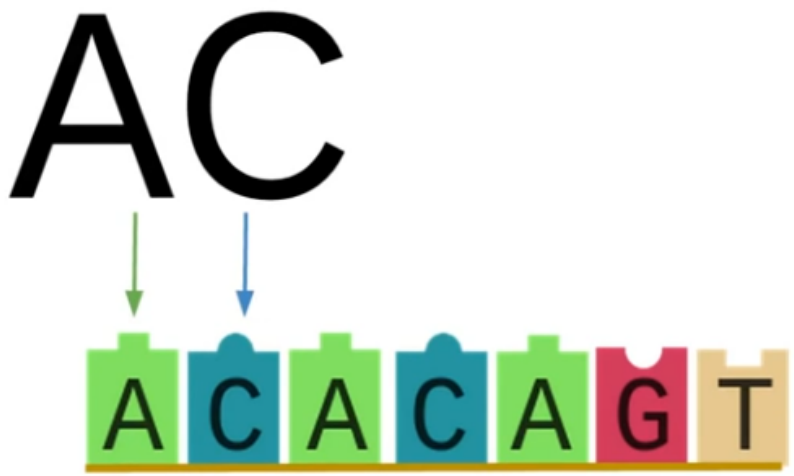
- ➔ we have have no possible prefixes
- ➔ we have no possible suffixes

KMP Algorithm

i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	**0**						

So at the first letter, we always have a value of 0 in our prefix table.

KMP Algorithm



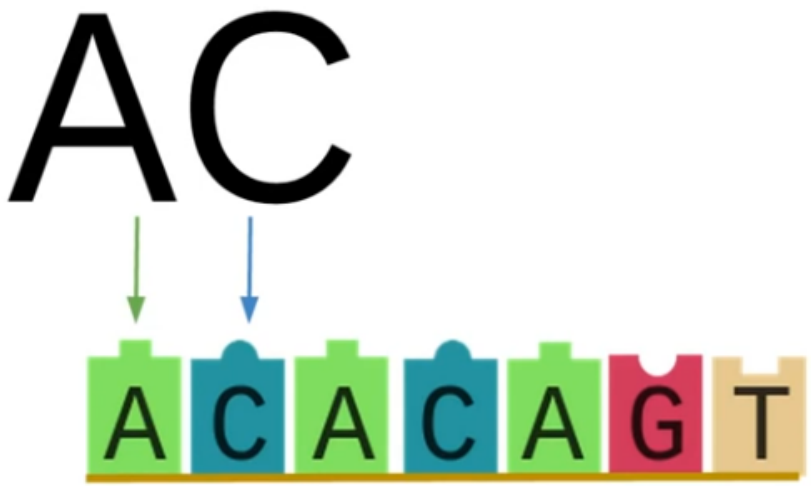
i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	0						

The only possible prefix is A.

The only possible suffix is C.

They do not match therefore...

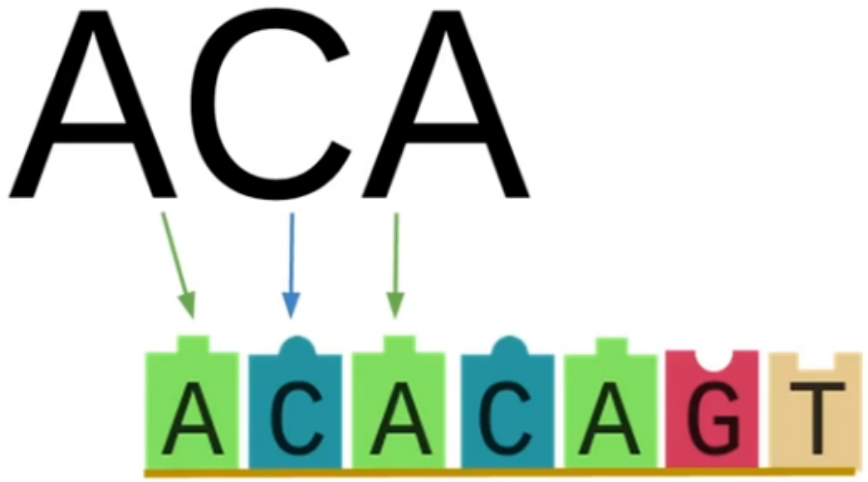
KMP Algorithm



i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	0	0					

The value in our prefix table at this point is 0.

KMP Algorithm



i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	**0**	**0**					

The possible prefixes are A, AC.

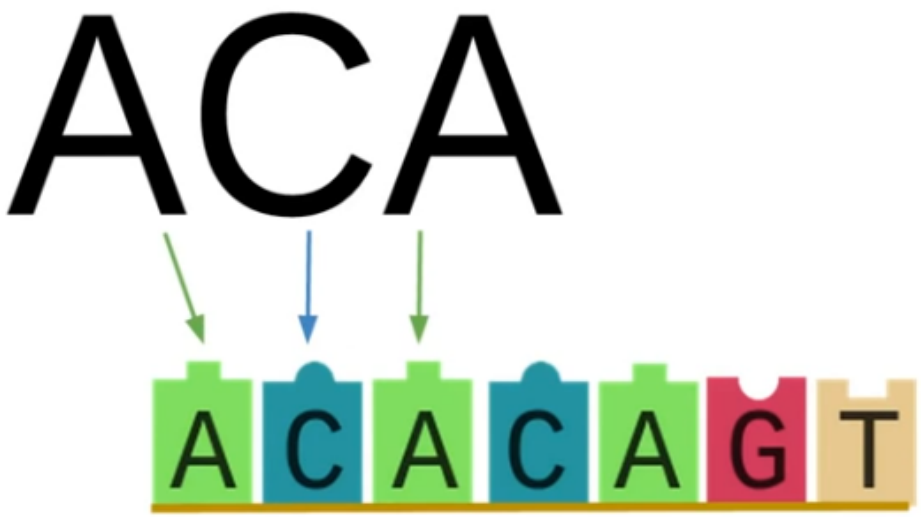
The possible suffixes are A, CA.

We notice that A is both a prefix and a suffix.

The length of A is 1.

So the value will be...

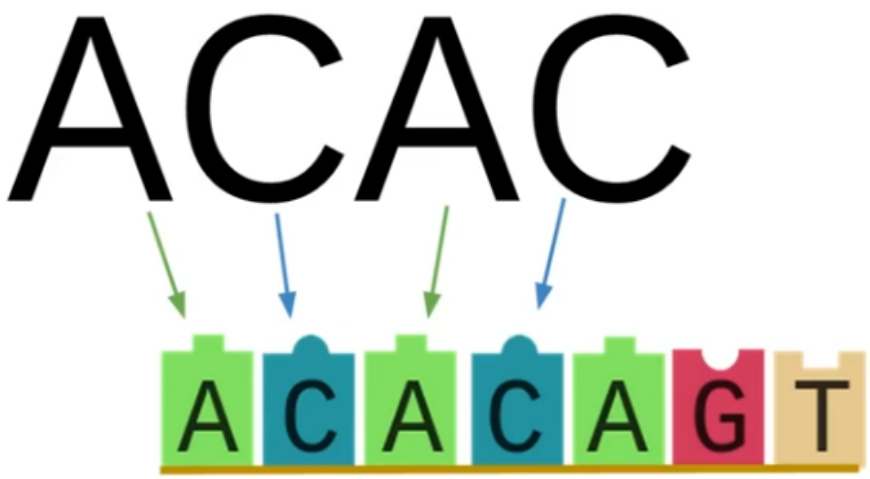
KMP Algorithm



i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	0	0	1				

The value in the prefix table will be 1!

KMP Algorithm



i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	0	0	1				

The possible prefixes are A, AC, ACA.

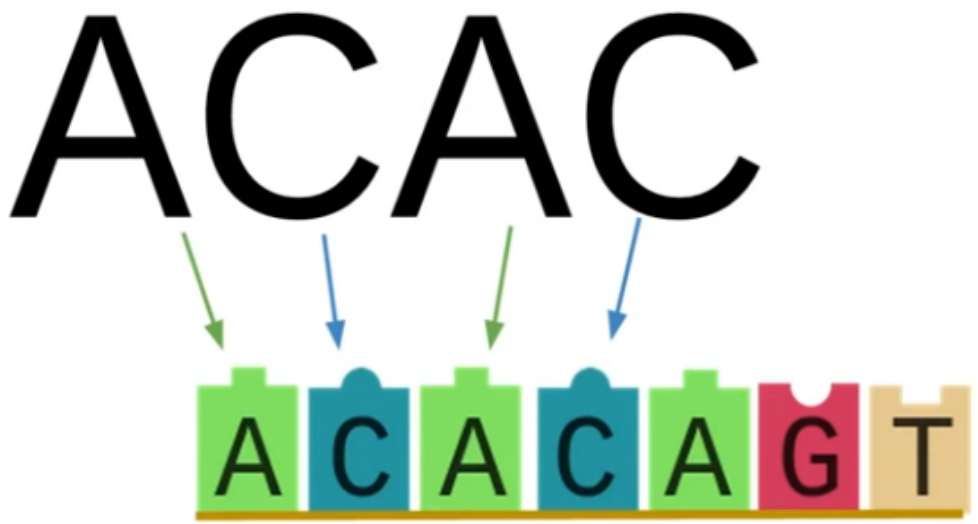
The possible suffixes are C, AC, CAC,

We notice that AC is both a prefix and a suffix.

The length of AC is 2.

So the value will be...

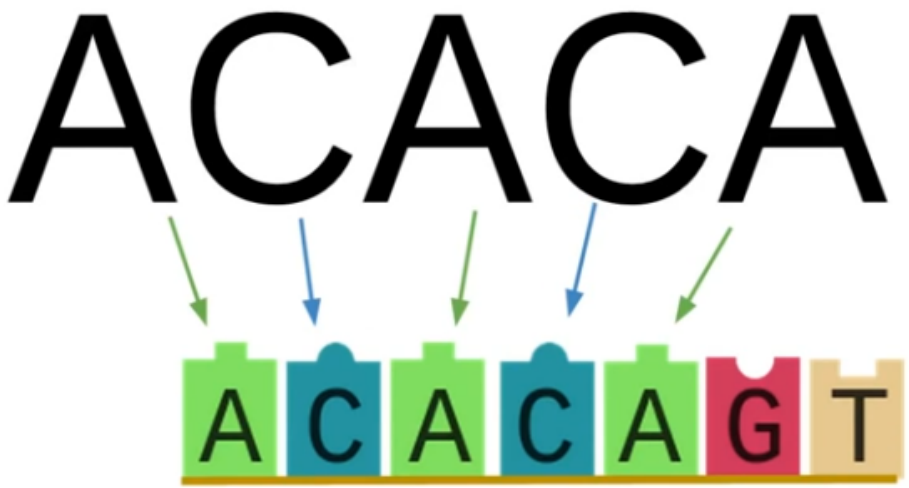
KMP Algorithm



i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	0	0	1	2			

The value at this point will be 2.

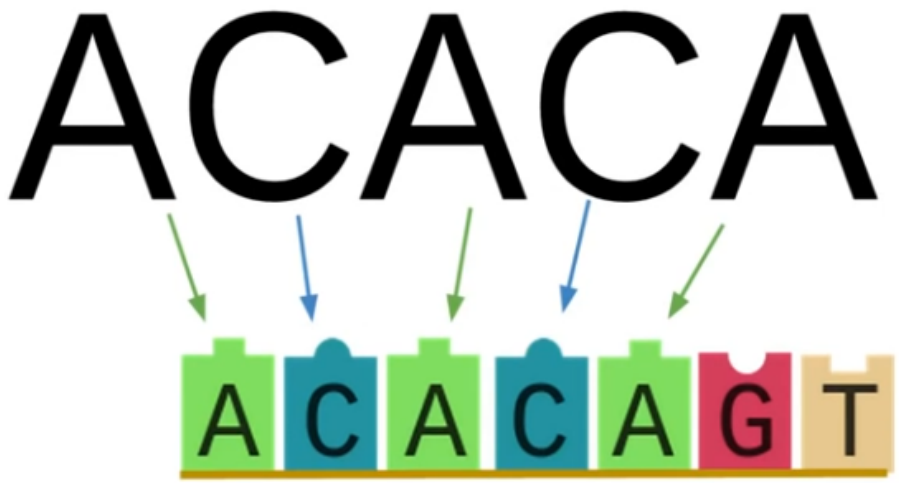
KMP Algorithm



i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	0	0	1	2			

It's your turn! ;)

KMP Algorithm



i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	**0**	**0**	**1**	**2**			

The possible prefixes are A, AC, ACA, ACAC.

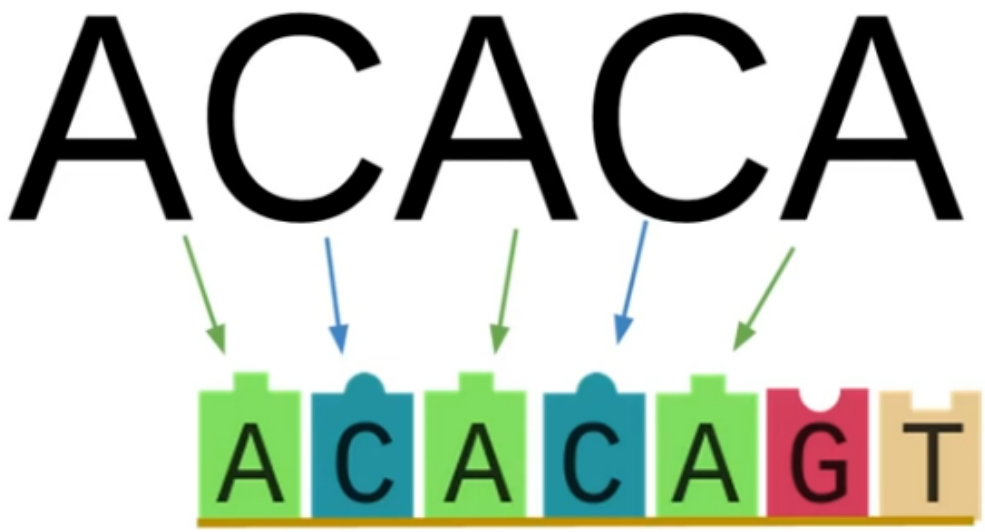
The possible suffixes are A, CA, ACA,CACA.

We notice that ACA is both a prefix and a suffix.

The length of ACA is 3.

So the value will be...

KMP Algorithm



i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	0	0	1	2	3		

The current prefix value will be 3.

KMP Algorithm

i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	**0**	**0**	**1**	**2**	**3**		

The possible prefixes are A, AC, ACA, ACACA.

The possible suffixes are G, AG, CAG, ACAG, CACAG.

We notice that the pattern does not have a prefix that is a suffix.

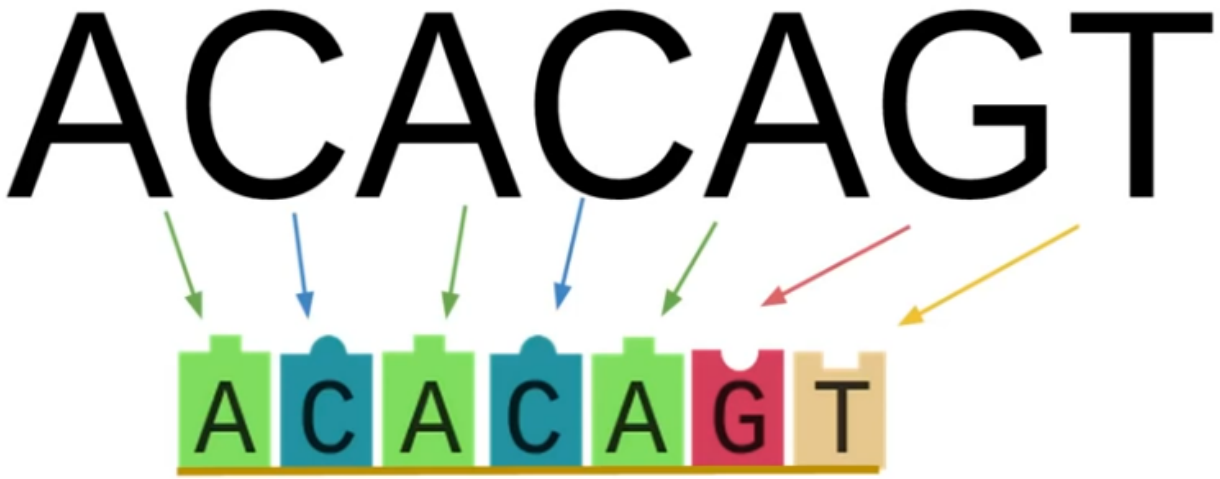
Therefore our value will be...

KMP Algorithm

i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	0	0	1	2	3	0	

The value in our prefix table at this point is 0.

KMP Algorithm



i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	**0**	**0**	**1**	**2**	**3**	**0**	

The possible prefixes are A, AC, ACA, ACACA, ACACAG.

The possible suffixes are T,GT,AGT,CAGT,ACAGT,CACAGT.

We notice that the pattern does not have a prefix that is a suffix.

Therefore our value will be..

KMP Algorithm

i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	0	0	1	2	3	0	0

The value in our prefix table at this point is 0.

KMP Algorithm

i	1	2	3	4	5	6	7
Pattern[i]	A	C	A	C	A	G	T
Prefix[i]	0	0	1	2	3	0	0

We have generated our prefix table and now we can use it in the KMP algorithm!

Exercise

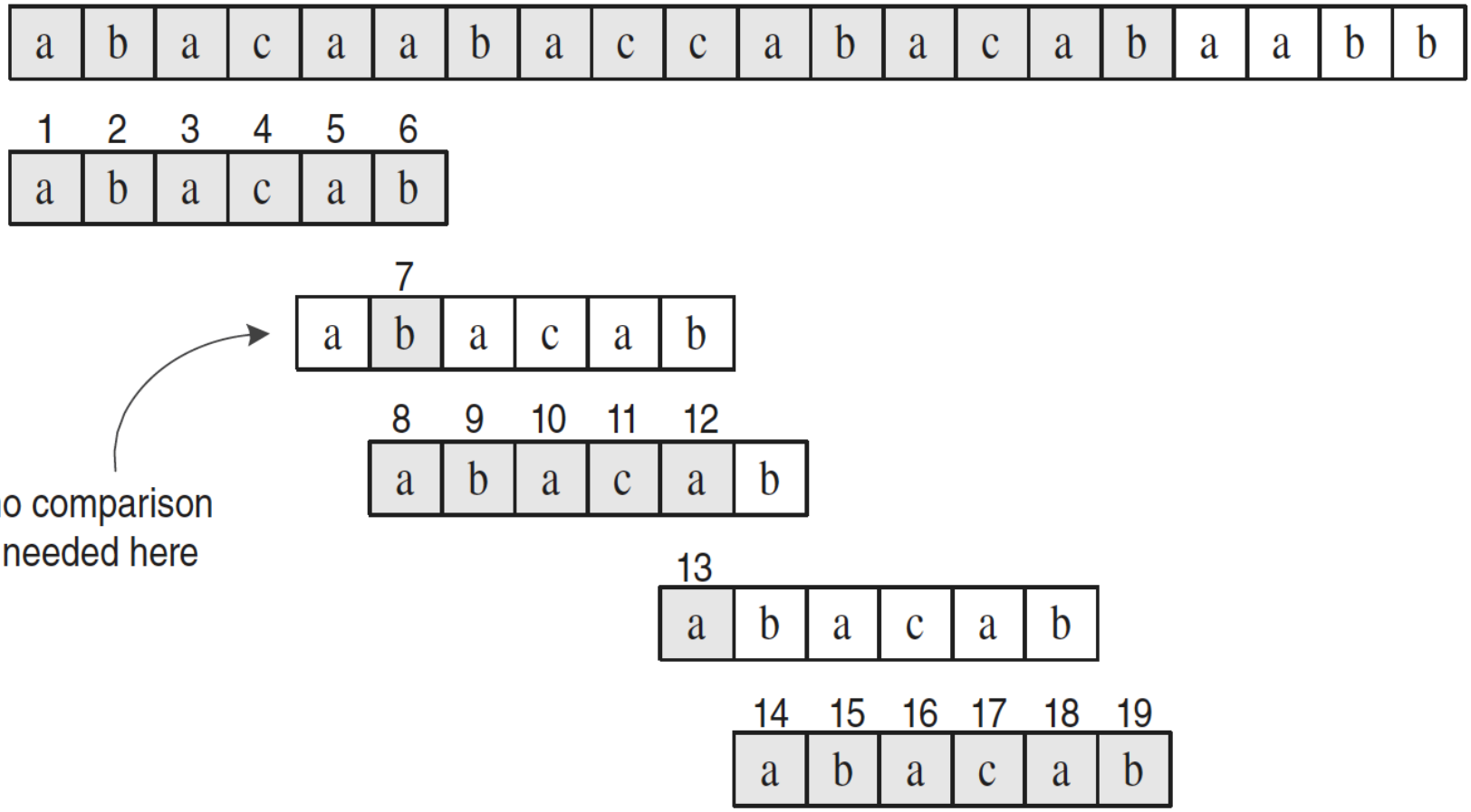
a	b	a	c	a	a	b	a	c	c	a	b	a	c	a	b	a	a	b	b

1	2	3	4	5	6
a	b	a	c	a	b

i	0	1	2	3	4	5
Pattern[i]	a	b	a	c	a	b
Prefix[i]	0	0	1	0	1	2

Exercise

i	0	1	2	3	4	5
Pattern[i]	a	b	a	c	a	b
Prefix[i]	0	0	1	0	1	2



Pseudocode for the prefix table generation

Prefix Table Pseudocode

Prefix table pseudocode

- ★ **m** is the **pattern** (P) length
- ★ **i** is the longest **prefix** that has been found in the **pattern**

  (**prefix** meaning prefix that is also a suffix at ***P[j]***. Referred to as longest prefix hereafter)
- ★ **j** is the current ***index*** of the **pattern** for which we are calculating the **π** value
- ★ **π** is the ***prefix table*** array

KMP-PREFIX(P)
begin
  $m \leftarrow |P|$
  $\Pi[1] \leftarrow 0$
  $i \leftarrow 0$
  for $j = 2$ **upto** m **step** 1 **do**
    while $i > 0$ **and** $P[i+1] \neq P[j]$ **do**
      $i \leftarrow \Pi[i]$
    if $P[i+1] = P[j]$ **then**
      $i \leftarrow i + 1$
    $\Pi[j] \leftarrow i$
  return Π
end

Prefix Table Pseudocode

P = ATAG

i	1	2	3	4
P[i]	A	T	A	G
π[i]				

- ★ **m** is the **pattern** (P) length
- ★ **i** is the longest **prefix** that has been found in **pattern**
- ★ **j** is the current ***index*** of the **pattern** for which we are calculating the π value
- ★ π is the ***prefix table*** array

KMP-PREFIX(P)
begin
  $m \leftarrow |P|$
  $\Pi[1] \leftarrow 0$
  $i \leftarrow 0$
  for $j = 2$ **upto** m **step** 1 **do**
    while $i > 0$ **and** $P[i+1] \neq P[j]$ **do**
      $i \leftarrow \Pi[i]$
    if $P[i+1] = P[j]$ **then**
      $i \leftarrow i + 1$
    $\Pi[j] \leftarrow i$
  return Π
end

Prefix Table Pseudocode

P = ATAG
m = |P| = 4

i	1	2	3	4
P[i]	A	T	A	G
π[i]				

- ★ **m** is the **pattern** (P) length
- ★ **i** is the longest **prefix** that has been found in **pattern**
- ★ **j** is the current ***index*** of the **pattern** for which we are calculating the π value
- ★ π is the ***prefix table*** array

```
KMP-PREFIX(P)
begin
  m ← |P|
  Π[1] ← 0
  i ← 0
  for j = 2 upto m step 1 do
    while i > 0 and P[i + 1] ≠ P[j] do
      i ← Π[i]
    if P[i + 1] = P[j] then
      i ← i + 1
    Π[j] ← i
  return Π
end
```

Prefix Table Pseudocode

P = ATAG
m = |P| = 4

i	1	2	3	4
P[i]	A	T	A	G
π[i]	0			

We know π[1] is always 0

- ★ **m** is the **pattern** (P) length
- ★ **i** is the longest **prefix** that has been found in **pattern**
- ★ **j** is the current ***index*** of the **pattern** for which we are calculating the π value
- ★ π is the ***prefix table*** array

```
KMP-PREFIX(P)
begin
  m ← |P|
  Π[1] ← 0
  i ← 0
  for j = 2 upto m step 1 do
    while i > 0 and P[i + 1] ≠ P[j] do
      i ← Π[i]
    if P[i + 1] = P[j] then
      i ← i + 1
    Π[j] ← i
  return Π
end
```

Prefix Table Pseudocode

P = ATAG

m = |P| = 4

i = 0 as we have not found the longest prefix yet

i	1	2	3	4
P[i]	A	T	A	G
π[i]	0			

- ★ **m** is the **pattern** (P) length
- ★ **i** is the longest **prefix** that has been found in **pattern**
- ★ **j** is the current ***index*** of the **pattern** for which we are calculating the π value
- ★ π is the ***prefix table*** array

```
KMP-PREFIX(P)
begin
  m ← |P|
  Π[1] ← 0
  i ← 0
  for j = 2 upto m step 1 do
    while i > 0 and P[i + 1] ≠ P[j] do
      i ← Π[i]
    if P[i + 1] = P[j] then
      i ← i + 1
    Π[j] ← i
  return Π
end
```

Prefix Table Pseudocode

P = ATAG

m = |P| = 4

i = 0 as we have not found the longest prefix yet

j = 2 as we already know π[1] is 0

i	1	2	3	4
P[i]	A	T	A	G
π[i]	0			

- ★ **m** is the **pattern** (P) length
- ★ **i** is the longest **prefix** that has been found in **pattern**
- ★ **j** is the current ***index*** of the **pattern** for which we are calculating the π value
- ★ π is the ***prefix table*** array

KMP-PREFIX(P)
begin
  $m \leftarrow |P|$
  $\Pi[1] \leftarrow 0$
  $i \leftarrow 0$
→ **for** $j = 2$ **upto** m **step** 1 **do**
    while $i > 0$ **and** $P[i+1] \neq P[j]$ **do**
      $i \leftarrow \Pi[i]$
    if $P[i+1] = P[j]$ **then**
      $i \leftarrow i + 1$
    $\Pi[j] \leftarrow i$
  return Π
end

Prefix Table Pseudocode

P = ATAG
m = |P| = 4
i = 0 as we have not found the longest prefix yet
j = 2 as we already know π[1] is 0

i	1	2	3	4
P[i]	A	T	A	G
π[i]	0			

- ★ **m** is the **pattern** (P) length
- ★ **i** is the longest **prefix** that has been found in **pattern**
- ★ **j** is the current ***index*** of the **pattern** for which we are calculating the π value
- ★ π is the ***prefix table*** array

KMP-PREFIX(P)
begin
  $m \leftarrow |P|$
  $\Pi[1] \leftarrow 0$
  $i \leftarrow 0$
  for $j = 2$ **upto** m **step** 1 **do**
    while $i > 0$ **and** $P[i+1] \neq P[j]$ **do**
      $i \leftarrow \Pi[i]$
    if $P[i+1] = P[j]$ **then**
      $i \leftarrow i + 1$
    $\Pi[j] \leftarrow i$
  return Π
end

Prefix Table Pseudocode

P = ATAG
m = |P| = 4
i = 0
j = 2

i	1	2	3	4
P[i]	A	T	A	G
π[i]	0			

- ★ **m** is the **pattern** (P) length
- ★ **i** is the longest **prefix** that has been found in **pattern**
- ★ **j** is the current ***index*** of the **pattern** for which we are calculating the π value
- ★ π is the ***prefix table*** array

If (A == T) which is **false** in this case so we…

The algorithm is trying to find a prefix here.

KMP-PREFIX(P)
begin
  $m \leftarrow |P|$
  $\Pi[1] \leftarrow 0$
  $i \leftarrow 0$
  for $j = 2$ **upto** m **step** 1 **do**
    while $i > 0$ **and** $P[i+1] \neq P[j]$ **do**
      $i \leftarrow \Pi[i]$
    if $P[i+1] = P[j]$ **then**
      $i \leftarrow i + 1$
    $\Pi[j] \leftarrow i$
  return Π
end

Prefix Table Pseudocode

P = ATAG
m = |P| = 4
i = 0
j = 2

i	1	2	3	4
P[i]	A	T	A	G
π[i]	0	0		

- ★ **m** is the **pattern** (P) length
- ★ **i** is the longest **prefix** that has been found in **pattern**
- ★ **j** is the current ***index*** of the **pattern** for which we are calculating the π value
- ★ π is the ***prefix table*** array

We add the value of i to our prefix table
π[2] = 0

KMP-PREFIX(P)
begin
  $m \leftarrow |P|$
  $\Pi[1] \leftarrow 0$
  $i \leftarrow 0$
  for $j = 2$ **upto** m **step** 1 **do**
    while $i > 0$ **and** $P[i+1] \neq P[j]$ **do**
      $i \leftarrow \Pi[i]$
    if $P[i+1] = P[j]$ **then**
      $i \leftarrow i + 1$
    $\Pi[j] \leftarrow i$
  return Π
end

Prefix Table Pseudocode

P = ATAG
m = |P| = 4
i = 0
j = 3

i	1	2	3	4
P[i]	A	T	A	G
π[i]	0	0		

- ★ **m** is the **pattern** (P) length
- ★ **i** is the longest **prefix** that has been found in **pattern**
- ★ **j** is the current ***index*** of the **pattern** for which we are calculating the π value
- ★ π is the ***prefix table*** array

We increment J to 3...

KMP-PREFIX(P)
begin
  $m \leftarrow |P|$
  $\Pi[1] \leftarrow 0$
  $i \leftarrow 0$
→ **for** $j = 2$ **upto** m **step** 1 **do**
    while $i > 0$ **and** $P[i+1] \neq P[j]$ **do**
      $i \leftarrow \Pi[i]$
    if $P[i+1] = P[j]$ **then**
      $i \leftarrow i + 1$
    $\Pi[j] \leftarrow i$
  return Π
end

Prefix Table Pseudocode

P = ATAG
m = |P| = 4
i = 0
j = 3

i	1	2	3	4
P[i]	A	T	A	G
π[i]	0	0		

- ★ **m** is the **pattern** (P) length
- ★ **i** is the longest **prefix** that has been found in **pattern**
- ★ **j** is the current ***index*** of the **pattern** for which we are calculating the π value
- ★ π is the ***prefix table*** array

If(A == A)
which is ***true***

KMP-PREFIX(P)
begin
  $m \leftarrow |P|$
  $\Pi[1] \leftarrow 0$
  $i \leftarrow 0$
  for $j = 2$ **upto** m **step** 1 **do**
    while $i > 0$ **and** $P[i+1] \neq P[j]$ **do**
      $i \leftarrow \Pi[i]$
    if $P[i+1] = P[j]$ **then**
      $i \leftarrow i + 1$
    $\Pi[j] \leftarrow i$
  return Π
end

Prefix Table Pseudocode

P = ATAG
m = |P| = 4
i = **1**
j = 3

i	1	2	3	4
P[i]	A	T	A	G
π[i]	0	0		

- ★ **m** is the **pattern** (P) length
- ★ **i** is the longest **prefix** that has been found in **pattern**
- ★ **j** is the current ***index*** of the **pattern** for which we are calculating the π value
- ★ π is the ***prefix table*** array

so we increment i by 1

```
KMP-PREFIX(P)
begin
  m ← |P|
  Π[1] ← 0
  i ← 0
  for j = 2 upto m step 1 do
    while i > 0 and P[i + 1] ≠ P[j] do
      i ← Π[i]
    if P[i + 1] = P[j] then
      i ← i + 1
    Π[j] ← i
  return Π
end
```

Prefix Table Pseudocode

P = ATAG
m = |P| = 4
i = 1
j = 3

i	1	2	3	4
P[i]	A	T	A	G
π[i]	0	0	1	

- ★ **m** is the **pattern** (P) length
- ★ **i** is the longest **prefix** that has been found in **pattern**
- ★ **j** is the current ***index*** of the **pattern** for which we are calculating the π value
- ★ π is the ***prefix table*** array

We add the value of i to our prefix table
π[3] = 1

```
KMP-PREFIX(P)
begin
  m ← |P|
  Π[1] ← 0
  i ← 0
  for j = 2 upto m step 1 do
    while i > 0 and P[i + 1] ≠ P[j] do
      i ← Π[i]
    if P[i + 1] = P[j] then
      i ← i + 1
    Π[j] ← i
  return Π
end
```

Prefix Table Pseudocode

P = **ATAG**
m = |P| = 4
i = 1
j = **4**

i	1	2	3	4
P[i]	A	T	A	G
π[i]	0	0	1	

- ★ **m** is the **pattern** (P) length
- ★ **i** is the longest **prefix** that has been found in **pattern**
- ★ **j** is the current ***index*** of the **pattern** for which we are calculating the π value
- ★ π is the ***prefix table*** array

We increment J to 4...

KMP-PREFIX(P)
begin
  $m \leftarrow |P|$
  $\Pi[1] \leftarrow 0$
  $i \leftarrow 0$
→ **for** $j = 2$ **upto** m **step** 1 **do**
    while $i > 0$ **and** $P[i+1] \neq P[j]$ **do**
      $i \leftarrow \Pi[i]$
    if $P[i+1] = P[j]$ **then**
      $i \leftarrow i + 1$
    $\Pi[j] \leftarrow i$
  return Π
end

Prefix Table Pseudocode

P = **ATAG**
m = |P| = 4
i = **1**
j = 4

i	1	2	3	4
P[i]	A	T	A	G
π[i]	0	0	1	

We check if **i** is greater than 0 (this ensures we have found a prefix)

...and the *letter after* **i** is **not** equal to the **j**th character...

- ★ **m** is the **pattern** (P) length
- ★ **i** is the longest **prefix** that has been found in **pattern**
- ★ **j** is the current ***index*** of the **pattern** for which we are calculating the π value
- ★ π is the ***prefix table*** array

KMP-PREFIX(P)
begin
  $m \leftarrow |P|$
  $\Pi[1] \leftarrow 0$
  $i \leftarrow 0$
  for $j = 2$ **upto** m **step** 1 **do**
→ **while** $i > 0$ **and** $P[i+1] \neq P[j]$ **do**
    $i \leftarrow \Pi[i]$
  if $P[i+1] = P[j]$ **then**
    $i \leftarrow i + 1$
  $\Pi[j] \leftarrow i$
  return Π
end

Prefix Table Pseudocode

P = ATAG
m = |P| = 4
i = **0**
j = 4

i	1	2	3	4
P[i]	A	T	A	G
π[i]	0	0	1	

- ★ **m** is the **pattern** (P) length
- ★ **i** is the longest **prefix** that has been found in **pattern**
- ★ **j** is the current ***index*** of the **pattern** for which we are calculating the π value
- ★ π is the ***prefix table*** array

Which is **true** so we **do**
i = π[i]
which makes **i = 0**

This makes our **current** longest *prefix* equal to our **previously** generated *prefix value*

KMP-PREFIX(P)
begin
  $m \leftarrow |P|$
  $\Pi[1] \leftarrow 0$
  $i \leftarrow 0$
  for $j = 2$ **upto** m **step** 1 **do**
    while $i > 0$ **and** $P[i+1] \neq P[j]$ **do**
      $i \leftarrow \Pi[i]$
    if $P[i+1] = P[j]$ **then**
      $i \leftarrow i + 1$
    $\Pi[j] \leftarrow i$
  return Π
end

Prefix Table Pseudocode

P = ATAG
m = |P| = 4
i = 0
j = 4

i	1	2	3	4
P[i]	A	T	A	G
π[i]	0	0	1	

- ★ **m** is the **pattern** (P) length
- ★ **i** is the longest **prefix** that has been found in **pattern**
- ★ **j** is the current ***index*** of the **pattern** for which we are calculating the π value
- ★ π is the ***prefix table*** array

We check if P[i+1] which is **A** is equal to P[j] which is **G**... which is not **true**.

The algorithm is trying to find another prefix.

KMP-PREFIX(P)
begin
  $m \leftarrow |P|$
  $\Pi[1] \leftarrow 0$
  $i \leftarrow 0$
  for $j = 2$ **upto** m **step** 1 **do**
    while $i > 0$ **and** $P[i+1] \neq P[j]$ **do**
      $i \leftarrow \Pi[i]$
    if $P[i+1] = P[j]$ **then**
      $i \leftarrow i + 1$
    $\Pi[j] \leftarrow i$
  return Π
end

Prefix Table Pseudocode

P = ATAG
m = |P| = 4
i = 0
j = 4

i	1	2	3	4
P[i]	A	T	A	G
π[i]	0	0	1	**0**

- ★ **m** is the **pattern** (P) length
- ★ **i** is the longest **prefix** that has been found in **pattern**
- ★ **j** is the current ***index*** of the **pattern** for which we are calculating the **π** value
- ★ **π** is the ***prefix table*** array

We add the value of i to our prefix table
π[4] = 0

KMP-PREFIX(P)
begin
  $m \leftarrow |P|$
  $\Pi[1] \leftarrow 0$
  $i \leftarrow 0$
  for $j = 2$ **upto** m **step** 1 **do**
    while $i > 0$ **and** $P[i+1] \neq P[j]$ **do**
      $i \leftarrow \Pi[i]$
    if $P[i+1] = P[j]$ **then**
      $i \leftarrow i + 1$
  → $\Pi[j] \leftarrow i$
  return Π
end

Prefix Table Pseudocode

P = ATAG
m = |P| = 4
i = 0
j = 4

i	1	2	3	4
P[i]	A	T	A	G
π[i]	0	0	1	0

- ★ **m** is the **pattern** (P) length
- ★ **i** is the longest **prefix** that has been found in **pattern**
- ★ **j** is the current ***index*** of the **pattern** for which we are calculating the π value
- ★ π is the ***prefix table*** array

We **return** our prefix table because the for loop terminates and we have reached the end of our pattern.

KMP-PREFIX(P)
begin
  $m \leftarrow |P|$
  $\Pi[1] \leftarrow 0$
  $i \leftarrow 0$
  for $j = 2$ **upto** m **step** 1 **do**
    while $i > 0$ **and** $P[i+1] \neq P[j]$ **do**
      $i \leftarrow \Pi[i]$
    if $P[i+1] = P[j]$ **then**
      $i \leftarrow i + 1$
    $\Pi[j] \leftarrow i$
  return Π
end

Now the KMP pseudocode...

KMP Pseudocode

KMP pseudocode

The pseudocode is ***almost*** identical to the prefix table's pseudocode!

Differences:

➔ **for** loop runs for the ***length*** of the **text** rather than the **pattern**

★ **n** is the **text** (T) ***length***
★ **m** is the **pattern** (P) ***length***
★ **i** points to the most **matches** in the **pattern** so far
★ **j** is the pointer to the **text** where the pattern is currently aligned to.
★ π is the ***prefix table*** array

```
KMP-MATCHER(T, P)
begin
  n ← |T|
  m ← |P|
  Π ← KMP-PREFIX(P)
  i ← 0
  for j = 1 upto n step 1 do
    while i > 0 and P[i + 1] ≠ T[j] do
      i ← Π[i]          ▷ Skip using prefix table
    if P[i + 1] = T[j] then
      i ← i + 1         ▷ Next character matches
    if i = m then
      OUTPUT(j − m) ▷ Pattern at shift j − m
      i ← Π[i]          ▷ Look for next match
end
```

KMP Pseudocode

KMP pseudocode

The pseudocode is ***almost*** identical to the prefix table's pseudocode!

Differences:

- ➔ **for** loop runs for the ***length*** of the **text** rather than the **pattern**
- ➔ **Comparisons** are performed against the **text**

- ★ **n** is the **text** (T) ***length***
- ★ **m** is the **pattern** (P) ***length***
- ★ **i** points to the most **matches** in the **pattern** so far
- ★ **j** is the pointer to the **text** where the pattern is currently aligned to.
- ★ π is the ***prefix table*** array

```
KMP-MATCHER(T, P)
begin
  n ← |T|
  m ← |P|
  Π ← KMP-PREFIX(P)
  i ← 0
  for j = 1 upto n step 1 do
    while i > 0 and P[i + 1] ≠ T[j] do
      i ← Π[i]          ▷ Skip using prefix table
    if P[i + 1] = T[j] then
      i ← i + 1         ▷ Next character matches
    if i = m then
      OUTPUT(j − m) ▷ Pattern at shift j − m
      i ← Π[i]          ▷ Look for next match
end
```

KMP Pseudocode

KMP pseudocode

The pseudocode is ***almost*** identical to the prefix table's pseudocode!

Differences:

- ➔ **for** loop runs for the ***length*** of the **text** rather than the **pattern**
- ➔ **Comparisons** are performed against the **text**
- ➔ It performs an extra **if** that checks whether the longest **match** is **equal** to the **pattern length** or not to see if it has *found* the pattern or not.
  - ➔ If **true**,
    It outputs the result as the first character of the pattern found in the **text**.
  - ➔ *Otherwise,*
    the algorithm *loops* to find all *occurrences* until the end of the **text** is reached.

- ★ **n** is the **text** (T) ***length***
- ★ **m** is the **pattern** (P) ***length***
- ★ **i** points to the most **matches** in the **pattern** so far
- ★ **j** is the pointer to the **text** where the pattern is currently aligned to.
- ★ π is the ***prefix table*** array

```
KMP-MATCHER(T, P)
begin
  n ← |T|
  m ← |P|
  Π ← KMP-PREFIX(P)
  i ← 0
  for j = 1 upto n step 1 do
    while i > 0 and P[i + 1] ≠ T[j] do
      i ← Π[i]          ▷ Skip using prefix table
    if P[i + 1] = T[j] then
      i ← i + 1         ▷ Next character matches
    if i = m then
      OUTPUT(j − m) ▷ Pattern at shift j − m
      i ← Π[i]          ▷ Look for next match
end
```

Worst Case of KMP

i	1	2	3	4
Pattern[i]	A	G	T	C
Prefix[i]	0	0	0	0

- ➔ We never find a suffix that is also a prefix
- ➔ All the values in our prefix table will be 0.
- ➔ So we will never skip any comparisons
- ➔ This is means KMP perform exactly like the Naive method
- ➔ This is the worst case of KMP.

Time Complexity of KMP

Time Complexi

Our worry is that the inner loop
d up to m times per
tion, leading to the
plexities O(mn) .

nnot happen.

You always keep track of "where you are" within the search pattern, but you never ever have to backtrack as you move through the text. You just keep going, and that's why the time complexity is just O of the length of the text.

Time Complexity = **O(n + m)**

m is the complexity of the precomputation of the prefix table

n is the complexity of the matching

Because n is usually significantly greater than m, we can say that the complexity is **O(n).**

Space Complexity of KMP

Space complexity = O(m)

where m is the length of the prefix table.

Homework

- Please refer to Icourse and Huawei Cloud.

- Due date for quiz: 23:30 April 5, 2021
- Due date for homework: 23:30 April 10, 2021
- Due date for lab assignment: 23:30 April 10, 2021