



Institut Pasteur

Pattern matching 1/2

Exact Pattern Matching

- Nicolas Maillet
- 23/11/08



Part 1

Exact Pattern matching



Pattern matching?

What for?

- Are you here?

Pattern matching?

What for?

- Are you here?
- Where are you?

Pattern matching?

What for?

- Are you here?
- Where are you?

Why?

Pattern matching?

What for?

- Are you here?
- Where are you?

Why?

- CTRL-F

Pattern matching?

What for?

- Are you here?
- Where are you?

Why?

- CTRL-F
- Search engines (50 billion web pages)

Pattern matching?

What for?

- Are you here?
- Where are you?

Why?

- CTRL-F
- Search engines (50 billion web pages)
- DB requests

Pattern matching?

What for?

- Are you here?
- Where are you?

Why?

- CTRL-F
- Search engines (50 billion web pages)
- DB requests
- Music (plagiarism, licenses youtube, shazam)

Pattern matching?

What for?

- Are you here?
- Where are you?

Why?

- CTRL-F
- Search engines (50 billion web pages)
- DB requests
- Music (plagiarism, licenses youtube, shazam)
- Spell checking

Pattern matching?

What for?

- Are you here?
- Where are you?

Why?

- CTRL-F
- Search engines (50 billion web pages)
- DB requests
- Music (plagiarism, licenses youtube, shazam)
- Spell checking
- Spam filtering

Pattern matching?

What for?

- Are you here?
- Where are you?

Why?

- CTRL-F
- Search engines (50 billion web pages)
- DB requests
- Music (plagiarism, licenses youtube, shazam)
- Spell checking
- Spam filtering
- Zettabytes (10^{21}) of data in biology

Application in biology

Is this gene/allele/chromosome/... is in this genome/data/...?

Application in biology

Is this gene/allele/chromosome/... is in this genome/data/...?

Many different use case... Recombination, integron, crossover, etc

Different pattern matching

Exact: Search ACTG in ACGCTAACGGACGCA

Different pattern matching

Exact: Search ACTG in ACGCTAACGGACGCA

Approximate: Search ACTG in ACGCTAACGGACGCA with at most x substitutions, insertions and deletions

Different pattern matching

Exact: Search ACTG in ACGCTAACGGACGCA

Approximate: Search ACTG in ACGCTAACGGACGCA with at most x substitutions, insertions and deletions

First, let see **Exact pattern matching**

What we saw

- Array/list/hash table/suffix array

What we saw

- Array/list/hash table/suffix array
- Suffix array: exact pattern matching with binary search! And more than just exact pattern matching!

What we saw

- Array/list/hash table/suffix array
- Suffix array: exact pattern matching with binary search! And more than just exact pattern matching!
- We need to index the **text**... 13GB for a human genome...

What we saw

- Array/list/hash table/suffix array
- Suffix array: exact pattern matching with binary search! And more than just exact pattern matching!
- We need to index the **text**... 13GB for a human genome...
- Can we do exact pattern matching without indexing?



Part 2

Naïve algorithm



Naive algorithm

Given a text T we want to find all the occurrences of a pattern P .

$T = \text{"ACGTACGATATATAGCTATAGCATGCATGCTA"}$

$P = \text{"TATAG"}$

How would you do it?

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P .

$T = \text{"ACGTACGATATATAGCTATAGCATGCATGCTA"}$

$P = \text{"TATAG"}$

The idea is to compare character after character on the main text to characters of pattern.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P .

$T = \text{"ACGTACGATATATAGCTATAGCATGCATGCTA"}$

$P = \text{"TATAG"}$

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P .

$T = \text{"ACGTACGATATATAGCTATAGCATGCATGCTA"}$

$P = \text{"TATAG"}$

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P .

$T = \text{"ACG}\color{green}{T}\text{ACGATATATAGCTATAGCATGCATGCTA"}$

$P = \text{"}\color{green}{T}\text{ATAG"}$

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P .

$T = \text{"ACGTACGATATATAGCTATAGCATGCATGCTA"}$

$P = \text{"TATAG"}$

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P .

$T = \text{"ACGTACGATATATAGCTATAGCATGCATGCTA"}$

$P = \text{"TATAG"}$

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

If one character is different, the search is reseted to the first character of the pattern and the next character of the text after the previous first match character.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P .

$T = \text{"ACGTACGATATATAGCTATAGCATGCATGCTA"}$

$P = \text{"TATAG"}$

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

If one character is different, the search is reseted to the first character of the pattern and the next character of the text after the previous first match character.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P.

T = "ACGTACGATATATAGCTATAGCATGCATGCTA"

P = "TATAG"

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

If one character is different, the search is reseted to the first character of the pattern and the next character of the text after the previous first match character.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P.

T = "ACGTACGATATATAGCTATAGCATGCATGCTA"

P = "TATAG"

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

If one character is different, the search is reseted to the first character of the pattern and the next character of the text after the previous first match character.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P.

T = "ACGTACGATATATAGCTATAGCATGCATGCTA"

P = "TATAG"

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

If one character is different, the search is reseted to the first character of the pattern and the next character of the text after the previous first match character.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P.

T = "ACGTACGA**T**ATATAGCTATAGCATGCATGCTA"

P = "**T**ATAG"

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

If one character is different, the search is reseted to the first character of the pattern and the next character of the text after the previous first match character.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P .

$T = \text{"ACGTACGATATATAGCTATAGCATGCATGCTA"}$

$P = \text{"TAG"}$

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

If one character is different, the search is reseted to the first character of the pattern and the next character of the text after the previous first match character.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P .

$T = \text{"ACGTACGATA\color{teal}TATAGCTATAGCATGCATGCTA"}$

$P = \text{"TA\color{teal}TAG"}$

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

If one character is different, the search is reseted to the first character of the pattern and the next character of the text after the previous first match character.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P .

$T = \text{"ACGTACGATATATAGCTATAGCATGCATGCTA"}$

$P = \text{"TATAG"}$

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

If one character is different, the search is reseted to the first character of the pattern and the next character of the text after the previous first match character.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P.

T = "ACGTACGATATA**T**AGCTATAGCATGCATGCTA"

P = "TATA**G**"

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

If one character is different, the search is reseted to the first character of the pattern and the next character of the text after the previous first match character.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P .

$T = \text{"ACGTACGATATATAGCTATAGCATGCATGCTA"}$

$P = \text{"TATAG"}$

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

If one character is different, the search is reseted to the first character of the pattern and the next character of the text after the previous first match character.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P.

T = "ACGTACGATA**T**ATAGCTATAGCATGCATGCTA"

P = "**T**ATAG"

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

If one character is different, the search is reseted to the first character of the pattern and the next character of the text after the previous first match character.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P .

$T = \text{"ACGTACGATATATAGCTATAGCATGCATGCTA"}$

$P = \text{"TAG"}$

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

If one character is different, the search is reseted to the first character of the pattern and the next character of the text after the previous first match character.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P .

$T = \text{"ACGTACGATATA\color{teal}TAGCTATAGCATGCATGCTA"}$

$P = \text{"TA\color{teal}TAG"}$

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

If one character is different, the search is reseted to the first character of the pattern and the next character of the text after the previous first match character.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P .

$T = \text{"ACGTACGATATATAGCTATAGCATGCATGCTA"}$

$P = \text{"TATAG"}$

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

If one character is different, the search is reseted to the first character of the pattern and the next character of the text after the previous first match character.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P.

T = "ACGTACGATATATAGCTATAGCATGCATGCTA"

P = "TATAG"

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

If one character is different, the search is reseted to the first character of the pattern and the next character of the text after the previous first match character.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P.

T = "ACGTACGATAT^ATAGCTATAGCATGCATGCTA"

P = "^TATAG"

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

If one character is different, the search is reseted to the first character of the pattern and the next character of the text after the previous first match character.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P .

$T = \text{"ACGTACGATATATAGCTATAGCATGCATGCTA"}$

$P = \text{"TAGATAG"}$

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

If one character is different, the search is reseted to the first character of the pattern and the next character of the text after the previous first match character.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P .

$T = \text{"ACGTACGATATATAGCTATAGCATGCATGCTA"}$

$P = \text{"TAG"}$

The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

If one character is different, the search is reseted to the first character of the pattern and the next character of the text after the previous first match character.

Naive algorithm

Given a text T we want to find all the occurrences of a pattern P .

$T = \text{"ACGTACGATATATAGCTATAGCATGCATGCTA"}$

$P = \text{"TATAG"}$


The idea is to compare character after character on the main text to characters of pattern.

While characters of the text are different than the first of the pattern, the algorithm moves on.

When the first character of pattern is identified, other characters of pattern are compared to the text, as long as they are identical.

If one character is different, the search is reseted to the first character of the pattern and the next character of the text after the previous first match character.

Time complexity: $\mathcal{O}(|T| \times |P|)$

A background image showing two scientists in a laboratory. A man in a white lab coat is leaning over a woman, also in a white lab coat, who is wearing glasses and looking down at something. They appear to be working together on a task.

Part 3

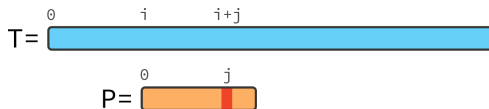
Knuth–Morris–Pratt

Idea of the algorithm



The naive algorithm is very slow. Suppose that we find the beginning of a match starting at position i on the text. Either:

Idea of the algorithm

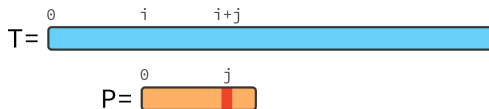


The naive algorithm is very slow. Suppose that we find the beginning of a match starting at position i on the text. Either:

We don't find a match. Then there is at least one index in which the text is not equal to the pattern. Let $i + j$ be the smallest:

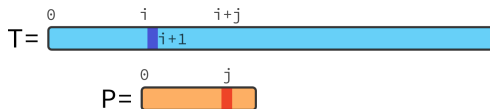
$$T[i \dots i + j - 1] = P[0 \dots j - 1] \text{ and } T[i + j] \neq P[j]$$

Idea of the algorithm



The naive algorithm is very slow. Suppose that we find the beginning of a match starting at position i on the text. Either:
We don't find a match. Then there is at least one index in which the text is not equal to the pattern. Let $i + j$ be the smallest:
 $T[i \dots i + j - 1] = P[0 \dots j - 1]$ and $T[i + j] \neq P[j]$
Since there is no match at i , we should start finding a match from another position: **where?**

Idea of the algorithm



The naive algorithm is very slow. Suppose that we find the beginning of a match starting at position i on the text. Either:

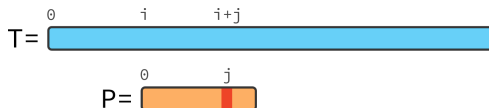
We don't find a match. Then there is at least one index in which the text is not equal to the pattern. Let $i + j$ be the smallest:

$$T[i \dots i + j - 1] = P[0 \dots j - 1] \text{ and } T[i + j] \neq P[j]$$

Since there is no match at i , we should start finding a match from another position: **where?**

Naive algorithm selects $i + 1$. This position might end up finding a mismatch.

Idea of the algorithm



The naive algorithm is very slow. Suppose that we find the beginning of a match starting at position i on the text. Either:

We don't find a match. Then there is at least one index in which the text is not equal to the pattern. Let $i + j$ be the smallest:

$$T[i \dots i + j - 1] = P[0 \dots j - 1] \text{ and } T[i + j] \neq P[j]$$

Since there is no match at i , we should start finding a match from another position: **where?**

Naive algorithm selects $i + 1$. This position might end up finding a mismatch.

We want to start from a position that guarantees that the strings matches until $i + j - 1$.

Idea of the algorithm



The naive algorithm is very slow. Suppose that we find the beginning of a match starting at position i on the text. Either:

We don't find a match. Then there is at least one index in which the text is not equal to the pattern. Let $i + j$ be the smallest:

$$T[i \dots i + j - 1] = P[0 \dots j - 1] \text{ and } T[i + j] \neq P[j]$$

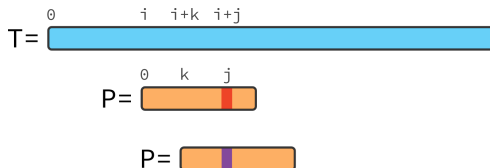
Since there is no match at i , we should start finding a match from another position: **where?**

Naive algorithm selects $i + 1$. This position might end up finding a mismatch.

We want to start from a position that guarantees that the strings matches until $i + j - 1$.

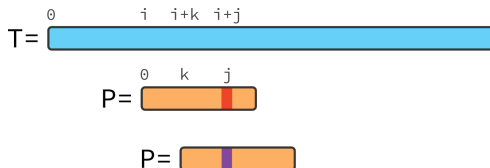
Therefore, we should find a match starting from the smallest $i + k$ such that $T[i + k \dots i + j - 1]$ matches with the **prefix** of P. We find a mismatch at j , so we start from the smallest k such that $P[k \dots j - 1]$ is a **prefix** of P.

Idea of the algorithm



The naive algorithm is very slow. Suppose that we find the beginning of a match starting at position i on the text. Either:
We don't find a match....

Idea of the algorithm



The naive algorithm is very slow. Suppose that we find the beginning of a match starting at position i on the text. Either:

We don't find a match....

Since we start from the smallest k such that $P[k...j-1]$ is a **prefix** of P , we don't need to check again $P[k...j-1]$. We can immediately check j position.

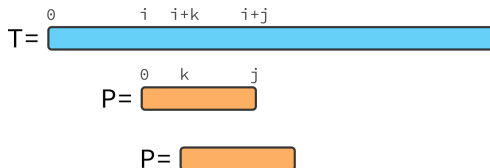
There is no backtracking in T .

Idea of the algorithm



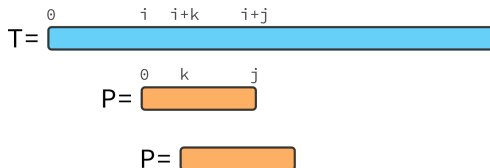
The naive algorithm is very slow. Suppose that we find the beginning of a match starting at position i on the text. Either:

Idea of the algorithm



The naive algorithm is very slow. Suppose that we find the beginning of a match starting at position i on the text. Either:
We don't find a match....

Idea of the algorithm



The naive algorithm is very slow. Suppose that we find the beginning of a match starting at position i on the text. Either:

We don't find a match....

We find a match. Using the same argument, it's easy to see that we have to start finding for the next match from the smallest k such that $P[k...j]$ is a **prefix** of P .

Steps of KMP

Knuth-Morris-Pratt algorithm (KMP) first analyze P to deduce informations allowing to then compare P to T by comparing only one time each characters. KMP can be seen as two different steps:

Steps of KMP

Knuth-Morris-Pratt algorithm (KMP) first analyze P to deduce informations allowing to then compare P to T by comparing only one time each characters. KMP can be seen as two different steps:

- First step: construction of an array indicating for each position in P an "offset", i.e. the next position where there is a potential occurrence of P.

Steps of KMP

Knuth-Morris-Pratt algorithm (KMP) first analyze P to deduce informations allowing to then compare P to T by comparing only one time each characters. KMP can be seen as two different steps:

- First step: construction of an array indicating for each position in P an "offset", i.e. the next position where there is a potential occurrence of P.
- Second step: compare each characters, one by one. When there is a mismatch, the previous array is used to directly jump to the next potentially interesting position.

Building the array

$P = \text{ACGTACA}$

j	i						
A	C	G	T	A	C	A	
0	1	2	3	4	5	6	
0							

Start with $j = 0$ et $i = 1$ and first slot is always 0.

Building the array

$P = \text{ACGTACA}$

j	i						
A	C	G	T	A	C	A	
0	1	2	3	4	5	6	
0							

Compare $P[i]$ and $P[j]$:
-mismatch between A and C.

Building the array

$P = \text{ACGTACA}$

j	i						
A	C	G	T	A	C	A	
0	1	2	3	4	5	6	
0	0						

Compare $P[i]$ and $P[j]$:

-mismatch between A and C.

The value at index i is the value of j

Building the array

$P = \text{ACGTACA}$

		j			i			
		A	C	G	T	A	C	A
		0	1	2	3	4	5	6
		0	0					

Compare $P[i]$ and $P[j]$:

-mismatch between A and C.

The value at index i is the value of j and we increment i

Building the array

$P = \text{ACGTACA}$

		j		i				
		A	C	G	T	A	C	A
		0	1	2	3	4	5	6
0	0							

Compare $P[i]$ and $P[j]$:
-mismatch between A and G.

Building the array

$P = \text{ACGTACA}$

		j		i				
		A	C	G	T	A	C	A
		0	1	2	3	4	5	6
0		0	0	0				

Compare $P[i]$ and $P[j]$:

-mismatch between A and G.

The value at index i is the value of j

Building the array

$P = \text{ACGTACA}$

j			i			
A	C	G	T	A	C	A
0	1	2	3	4	5	6
0	0	0				

Compare $P[i]$ and $P[j]$:

-mismatch between A and G.

The value at index i is the value of j and we increment i

Building the array

$P = \text{ACGTACA}$

j			i			
A	C	G	T	A	C	A
0	1	2	3	4	5	6
0	0	0				

Compare $P[i]$ and $P[j]$:
-mismatch between A and T.

Building the array

$P = \text{ACGTACA}$

j			i			
A	C	G	T	A	C	A
0	1	2	3	4	5	6
0	0	0	0			

Compare $P[i]$ and $P[j]$:

-mismatch between A and T.

The value at index i is the value of j

Building the array

$P = \text{ACGTACA}$

j				i		
A	C	G	T	A	C	A
0	1	2	3	4	5	6
0	0	0	0			

Compare $P[i]$ and $P[j]$:

-mismatch between A and T.

The value at index i is the value of j and we increment i

Building the array

$P = \text{ACGTACA}$

j				i		
A	C	G	T	A	C	A
0	1	2	3	4	5	6
0	0	0	0			

Compare $P[i]$ and $P[j]$:
-match between A and A.

Building the array

$P = \text{ACGTACA}$

j				i		
A	C	G	T	A	C	A
0	1	2	3	4	5	6
0	0	0	0	1		

Compare $P[i]$ and $P[j]$:

-match between A and A.

The value at index i is the value of j plus 1

Building the array

$P = \text{ACGTACA}$

j				i		
A	C	G	T	A	C	A
0	1	2	3	4	5	6
0	0	0	0	1		

Compare $P[i]$ and $P[j]$:

-match between A and A.

The value at index i is the value of j plus 1 and we increment both i and j

Building the array

$P = \text{ACGTACA}$

	j				i	
A	C	G	T	A	C	A
0	1	2	3	4	5	6
0	0	0	0	1		

Compare $P[i]$ and $P[j]$:
-match between C and C.

Building the array

$P = \text{ACGTACA}$

j				i		
A	C	G	T	A	C	A
0	1	2	3	4	5	6
0	0	0	0	1	2	

Compare $P[i]$ and $P[j]$:

-match between C and C.

The value at index i is the value of j plus 1

Building the array

$P = \text{ACGTACA}$

		j				i
A	C	G	T	A	C	A
0	1	2	3	4	5	6
0	0	0	0	1	2	

Compare $P[i]$ and $P[j]$:

-match between C and C.

The value at index i is the value of j plus 1 and we increment both i and j

Building the array

$P = \text{ACGTACA}$

		j				i
A	C	G	T	A	C	A
0	1	2	3	4	5	6
0	0	0	0	1	2	

Compare $P[i]$ and $P[j]$:
-mismatch between G and A.

Building the array

$P = \text{ACGTACA}$

		j			i		
A	C	G	T	A	C	A	
0	1	2	3	4	5	6	
0	0	0	0	1	2		

Compare $P[i]$ and $P[j]$:

-mismatch between G and A.

j will go at the **index** of the **value of the previous position** ($C = 0$).

Building the array

$P = \text{ACGTACA}$

j						i
A	C	G	T	A	C	A
0	1	2	3	4	5	6
0	0	0	0	1	2	

Compare $P[i]$ and $P[j]$:

-mismatch between G and A.

j will go at the **index** of the **value of the previous position** ($C = 0$).

Building the array

$P = \text{ACGTACA}$

j						i
A	C	G	T	A	C	A
0	1	2	3	4	5	6
0	0	0	0	1	2	

Compare $P[i]$ and $P[j]$:

Then, compare again $P[i]$ and $P[j]$.

Building the array

$P = \text{ACGTACA}$

j						i
A	C	G	T	A	C	A
0	1	2	3	4	5	6
0	0	0	0	1	2	1

Compare $P[i]$ and $P[j]$:

Then, compare again $P[i]$ and $P[j]$.

It is a match, so the value at index i is the value of j plus 1.

Building the array, more complicated example

P = AACAAACAAA

j	i								
A	A	C	A	A	C	A	A	A	
0	1	2	3	4	5	6	7	8	
0									

Start with $j = 0$ et $i = 1$ and first slot is always 0.

Building the array, more complicated example

$P = \text{AACAAACAAA}$

j	i								
A	A	C	A	A	C	A	A	A	
0	1	2	3	4	5	6	7	8	
0									

Compare $P[i]$ and $P[j]$:
-match between A and A.

Building the array, more complicated example

$P = \text{AACAAACAAA}$

j	i								
A	A	C	A	A	C	A	A	A	
0	1	2	3	4	5	6	7	8	
0	1								

Compare $P[i]$ and $P[j]$:

-match between A and A.

The value at index i is the value of j plus 1

Building the array, more complicated example

$P = \text{AACAAACAAA}$

	j	i							
A	A	C	A	A	C	A	A	A	
0	1	2	3	4	5	6	7	8	
0	1								

Compare $P[i]$ and $P[j]$:

-match between A and A.

The value at index i is the value of j plus 1 and we increment both i and j

Building the array, more complicated example

$P = \text{ACAACAAA}$

	j i								
A	A	C	A	A	C	A	A	A	
0	1	2	3	4	5	6	7	8	
0	1								

Compare $P[i]$ and $P[j]$:
-mismatch between A and C.

Building the array, more complicated example

$P = \text{AACAAACAAA}$

	j	i							
A	A	C	A	A	C	A	A	A	
0	1	2	3	4	5	6	7	8	
0	1								

Compare $P[i]$ and $P[j]$:

-mismatch between A and C.

j will go at the **index** of the **value of the previous position** ($A = 0$).

Building the array, more complicated example

$P = \text{AACAAACAAA}$

		j				i			
A	A	C	A	A	C	A	A	A	
0	1	2	3	4	5	6	7	8	
0	1								

Compare $P[i]$ and $P[j]$:

-mismatch between A and C.

j will go at the **index** of the **value of the previous position** ($A = 0$).

Building the array, more complicated example

$P = \text{ACAACAAA}$

		j				i			
A	A	C	A	A	C	A	A	A	
0	1	2	3	4	5	6	7	8	
0	1								

Compare $P[i]$ and $P[j]$:

Then, compare again $P[i]$ and $P[j]$.

Building the array, more complicated example

$P = \text{AACAAACAAA}$

		j				i			
A	A	C	A	A	C	A	A	A	
0	1	2	3	4	5	6	7	8	
0	1	0							

Compare $P[i]$ and $P[j]$:

Then, compare again $P[i]$ and $P[j]$.

It is a mismatch, so the value at index i is the value of j

Building the array, more complicated example

$P = \text{ACAACAAA}$

j			i						
A	A	C	A	A	C	A	A	A	
0	1	2	3	4	5	6	7	8	
0	1	0							

Compare $P[i]$ and $P[j]$:

Then, compare again $P[i]$ and $P[j]$.

It is a mismatch, so the value at index i is the value of j and we increment i

Building the array, more complicated example

$P = \text{AACAAACAAA}$

j			i					
A	A	C	A	A	C	A	A	A
0	1	2	3	4	5	6	7	8
0	1	0						

Compare $P[i]$ and $P[j]$:
-match between A and A.

Building the array, more complicated example

$P = \text{ACAACAAA}$

j			i						
A	A	C	A	A	C	A	A	A	
0	1	2	3	4	5	6	7	8	
0	1	0	1						

Compare $P[i]$ and $P[j]$:

-match between A and A.

The value at index i is the value of j plus 1

Building the array, more complicated example

$P = \text{ACAACAAA}$

j				i				
A	A	C	A	A	C	A	A	A
0	1	2	3	4	5	6	7	8
0	1	0	1					

Compare $P[i]$ and $P[j]$:

-match between A and A.

The value at index i is the value of j plus 1 and we increment both i and j

Building the array, more complicated example

$P = \text{AACAAACAAA}$

	j				i			
A	A	C	A	A	C	A	A	A
0	1	2	3	4	5	6	7	8
0	1	0	1					

Compare $P[i]$ and $P[j]$:
-match between A and A.

Building the array, more complicated example

$P = \text{ACAACAAA}$

	j			i				
A	A	C	A	A	C	A	A	A
0	1	2	3	4	5	6	7	8
0	1	0	1	2				

Compare $P[i]$ and $P[j]$:

-match between A and A.

The value at index i is the value of j plus 1

Building the array, more complicated example

$P = \text{AACAAACAAA}$

		j			i			
A	A	C	A	A	C	A	A	A
0	1	2	3	4	5	6	7	8
0	1	0	1	2				

Compare $P[i]$ and $P[j]$:

-match between A and A.

The value at index i is the value of j plus 1 and we increment both i and j

Building the array, more complicated example

$P = \text{AACAAACAAA}$

		j			i			
A	A	C	A	A	C	A	A	A
0	1	2	3	4	5	6	7	8
0	1	0	1	2				

Compare $P[i]$ and $P[j]$:
-match between C and C.

Building the array, more complicated example

$P = \text{ACAACAAA}$

j			i					
A	A	C	A	A	C	A	A	A
0	1	2	3	4	5	6	7	8
0	1	0	1	2	3			

Compare $P[i]$ and $P[j]$:

-match between C and C.

The value at index i is the value of j plus 1

Building the array, more complicated example

$P = \text{ACAACAAA}$

			j			i		
A	A	C	A	A	C	A	A	A
0	1	2	3	4	5	6	7	8
0	1	0	1	2	3			

Compare $P[i]$ and $P[j]$:

-match between C and C.

The value at index i is the value of j plus 1 and we increment both i and j

Building the array, more complicated example

$P = \text{AACAAACAAA}$

			j				i		
A	A	C	A	A	C	A	A	A	
0	1	2	3	4	5	6	7	8	
0	1	0	1	2	3				

Compare $P[i]$ and $P[j]$:
-match between A and A.

Building the array, more complicated example

$P = \text{ACAACAAA}$

			j			i		
A	A	C	A	A	C	A	A	A
0	1	2	3	4	5	6	7	8
0	1	0	1	2	3	4		

Compare $P[i]$ and $P[j]$:

-match between A and A.

The value at index i is the value of j plus 1

Building the array, more complicated example

$P = \text{ACAACAAA}$

				j						i
A	A	C	A	A	C	A	A	A		
0	1	2	3	4	5	6	7	8		
0	1	0	1	2	3	4				

Compare $P[i]$ and $P[j]$:

-match between A and A.

The value at index i is the value of j plus 1 and we increment both i and j

Building the array, more complicated example

$P = \text{AACAAACAAA}$

				j			i	
A	A	C	A	A	C	A	A	A
0	1	2	3	4	5	6	7	8
0	1	0	1	2	3	4		

Compare $P[i]$ and $P[j]$:
-match between A and A.

Building the array, more complicated example

$P = \text{ACAACAAA}$

				j						i
A	A	C	A	A	C	A	A	A		
0	1	2	3	4	5	6	7	8		
0	1	0	1	2	3	4	5			

Compare $P[i]$ and $P[j]$:

-match between A and A.

The value at index i is the value of j plus 1

Building the array, more complicated example

$P = \text{AACAAACAAA}$

					j					i
A	A	C	A	A	C	A	A	A		
0	1	2	3	4	5	6	7	8		
0	1	0	1	2	3	4	5			

Compare $P[i]$ and $P[j]$:

-match between A and A.

The value at index i is the value of j plus 1 and we increment both i and j

Building the array, more complicated example

$P = \text{ACAACAAA}$

					j			i
A	A	C	A	A	C	A	A	A
0	1	2	3	4	5	6	7	8
0	1	0	1	2	3	4	5	

Compare $P[i]$ and $P[j]$:
-mismatch between C and A.

Building the array, more complicated example

$P = \text{AACAAACAAA}$

					j			i
A	A	C	A	A	C	A	A	A
0	1	2	3	4	5	6	7	8
0	1	0	1	2	3	4	5	

Compare $P[i]$ and $P[j]$:

-mismatch between C and A.

j will go at the **index** of the **value of the previous position** ($A = 2$).

Building the array, more complicated example

$P = \text{AACAAACAAA}$

j									i
A	A	C	A	A	C	A	A	A	
0	1	2	3	4	5	6	7	8	
0	1	0	1	2	3	4	5		

Compare $P[i]$ and $P[j]$:

-mismatch between C and A.

j will go at the **index** of the **value of the previous position** ($A = 2$).

Building the array, more complicated example

$P = \text{ACAACAAA}$

		j						i
A	A	C	A	A	C	A	A	A
0	1	2	3	4	5	6	7	8
0	1	0	1	2	3	4	5	

Compare $P[i]$ and $P[j]$:

Then, compare again $P[i]$ and $P[j]$.

Building the array, more complicated example

$P = \text{AACAAACAAA}$

		j						i
A	A	C	A	A	C	A	A	A
0	1	2	3	4	5	6	7	8
0	1	0	1	2	3	4	5	

Compare $P[i]$ and $P[j]$:

Then, compare again $P[i]$ and $P[j]$.

It is a mismatch, so j will go at the **index** of the **value of the previous position** ($A = 1$).

Building the array, more complicated example

$P = \text{AACAAACAAA}$

	j							i
A	A	C	A	A	C	A	A	A
0	1	2	3	4	5	6	7	8
0	1	0	1	2	3	4	5	

Compare $P[i]$ and $P[j]$:

Then, compare again $P[i]$ and $P[j]$.

It is a mismatch, so j will go at the **index** of the **value of the previous position** ($A = 1$).

Building the array, more complicated example

$P = \text{ACAACAAA}$

j								i
A	A	C	A	A	C	A	A	A
0	1	2	3	4	5	6	7	8
0	1	0	1	2	3	4	5	

Compare $P[i]$ and $P[j]$:

Then, compare again $P[i]$ and $P[j]$.

Building the array, more complicated example

$P = \text{AACAAACAAA}$

	j							i
A	A	C	A	A	C	A	A	A
0	1	2	3	4	5	6	7	8
0	1	0	1	2	3	4	5	

Compare $P[i]$ and $P[j]$:

Then, compare again $P[i]$ and $P[j]$.

It is a match, so the value at index i is the value of j plus 1.

Building the array, more complicated example

$P = \text{AACAAACAAA}$

	j							i
A	A	C	A	A	C	A	A	A
0	1	2	3	4	5	6	7	8
0	1	0	1	2	3	4	5	2

Compare $P[i]$ and $P[j]$:

Then, compare again $P[i]$ and $P[j]$.

It is a match, so the value at index i is the value of j plus 1.

Running KMP

T = ATTATCATCATG

A	T	C	A	T	G
0	1	2	3	4	5
0	0	0	1	2	0

Running KMP

T = **A**TTATCATCATG

A	T	C	A	T	G
0	1	2	3	4	5
0	0	0	1	2	0

It is a match

Running KMP

T = A**T**TATCATCATG

A	T	C	A	T	G
0	1	2	3	4	5
0	0	0	1	2	0

It is a match

Running KMP

T = ATT**T**ATCATCATG

A	T	C	A	T	G
0	1	2	3	4	5
0	0	0	1	2	0

It is a mismatch!

Running KMP

T = ATT**T**ATCATCATG

A	T	C	A	T	G
0	1	2	3	4	5
0	0	0	1	2	0

It is a mismatch!

Let's look where is the next point of comparison in P (previous index)

Running KMP

T = ATTATCATCATG

A	T	C	A	T	G
0	1	2	3	4	5
0	0	0	1	2	0

It is a mismatch!

Let's look where is the next point of comparison in P (previous index)

Running KMP

T = ATTATCATCATG

A	T	C	A	T	G
0	1	2	3	4	5
0	0	0	1	2	0

It is a mismatch!

Running KMP

T = ATT**T**ATCATCATG

A	T	C	A	T	G
0	1	2	3	4	5
0	0	0	1	2	0

It is a mismatch!

Move on

Running KMP

T = ATT**A**TCATCATG

A	T	C	A	T	G
0	1	2	3	4	5

0	0	0	1	2	0
---	---	---	---	---	---

It is a match

Running KMP

T = ATTATCATCATG

A	T	C	A	T	G
0	1	2	3	4	5
0	0	0	1	2	0

It is a match

Running KMP

T = ATTAT**C**ATCATG

A	T	C	A	T	G
0	1	2	3	4	5
0	0	0	1	2	0

It is a match

Running KMP

T = ATTATC**A**TCATG

A	T	C	A	T	G
0	1	2	3	4	5
0	0	0	1	2	0

It is a match

Running KMP

T = ATTATCAT**T**CATG

A	T	C	A	T	G
0	1	2	3	4	5
0	0	0	1	2	0

It is a match

Running KMP

T = ATTATCAT**C**ATG

A	T	C	A	T	G
0	1	2	3	4	5
0	0	0	1	2	0

It is a mismatch!

Running KMP

T = ATTATCAT**C**ATG

A	T	C	A	T	G
0	1	2	3	4	5
0	0	0	1	2	0

It is a mismatch!

Let's look where is the next point of comparison in P (previous index)

Running KMP

T = ATTATCAT**C**ATG

A	T	C	A	T	G
0	1	2	3	4	5
0	0	0	1	2	0

It is a mismatch!

Let's look where is the next point of comparison in P (previous index)

Running KMP

T = ATTATCAT**C**ATG

A	T	C	A	T	G
0	1	2	3	4	5
0	0	0	1	2	0

It is a match

Running KMP

T = ATTATCATC**A**TG

A	T	C	A	T	G
0	1	2	3	4	5
0	0	0	1	2	0

It is a match

Running KMP

T = ATTATCATCAT**T**G

A	T	C	A	T	G
0	1	2	3	4	5
0	0	0	1	2	0

It is a match

Running KMP

T = ATTATCATCAT**G**

A	T	C	A	T	G
0	1	2	3	4	5
0	0	0	1	2	0

It is a match

Running KMP

Naive algorithm:

Time complexity: $\mathcal{O}(|T| \times |P|)$

Running KMP

Naive algorithm:

Time complexity: $\mathcal{O}(|T| \times |P|)$

KMP:

Time complexity: $\mathcal{O}(|T| + |P|)$

No backtracking, each character is only processed one time!

Part 4

Rabin-Karp



Idea of the algorithm

The main problem of the naive algorithm is that comparing two string is time consuming.

Idea of the algorithm

The main problem of the naive algorithm is that comparing two string is time consuming.

If we can compare quickly two strings, then we can use our naive algorithm (iterate over all i , and check if there is a match)

Idea of the algorithm

The main problem of the naive algorithm is that comparing two string is time consuming.

If we can compare quickly two strings, then we can use our naive algorithm (iterate over all i , and check if there is a match)

Digits are easy and fast to compare. How can we change our string to integer values?

Idea of the algorithm

The main problem of the naive algorithm is that comparing two string is time consuming.

If we can compare quickly two strings, then we can use our naive algorithm (iterate over all i , and check if there is a match)

Digits are easy and fast to compare. How can we change our string to integer values?

Hash function!

Idea of the algorithm

The main problem of the naive algorithm is that comparing two string is time consuming.

If we can compare quickly two strings, then we can use our naive algorithm (iterate over all i , and check if there is a match)

Digits are easy and fast to compare. How can we change our string to integer values?

Hash function!

Compute the hash of the pattern (length = k). Then compute the hash for each substring of text of size k , and compare hash values.

Idea of the algorithm

The main problem of the naive algorithm is that comparing two string is time consuming.

If we can compare quickly two strings, then we can use our naive algorithm (iterate over all i , and check if there is a match)

Digits are easy and fast to compare. How can we change our string to integer values?

Hash function!

Compute the hash of the pattern (length= k). Then compute the hash for each substring of text of size k , and compare hash values.

Hashes are different: move on the next hash of T ,

Idea of the algorithm

The main problem of the naive algorithm is that comparing two string is time consuming.

If we can compare quickly two strings, then we can use our naive algorithm (iterate over all i , and check if there is a match)

Digits are easy and fast to compare. How can we change our string to integer values?

Hash function!

Compute the hash of the pattern (length = k). Then compute the hash for each substring of text of size k , and compare hash values.

Hashes are different: move on the next hash of T ,

Hashes are identical: compare the strings (false positives are possible).

Idea of the algorithm

T=AACGTT $k = 3$

Doing this we will compute k times (almost) each positions. This is not efficient.

Idea of the algorithm

AA**C**gtt $k = 3$

Doing this we will compute k times (almost) each positions. This is not efficient.

Idea of the algorithm

aACGtt $k = 3$

Doing this we will compute k times (almost) each positions. This is not efficient.

Idea of the algorithm

aaCGTt $k = 3$

Doing this we will compute k times (almost) each positions. This is not efficient.

Idea of the algorithm

T=AACGTT $k = 3$

Doing this we will compute k times (almost) each positions. This is not efficient.

Concept of **rolling hash**

Rolling hash

A rolling hash allows an algorithm to calculate a hash value without having to rehash the entire string.

Rolling hash

A rolling hash allows an algorithm to calculate a hash value without having to rehash the entire string.

When hashing a substring, rolling hash will do an operation on the hash of previous substring to get the new hash from the old hash.

Rolling hash

A rolling hash allows an algorithm to calculate a hash value without having to rehash the entire string.

When hashing a substring, rolling hash will do an operation on the hash of previous substring to get the new hash from the old hash.

We want to perform rolling hash on ACGGT with $k = 3$. We will compute the hash for ACG, CGG then GGT.

Rolling hash

A rolling hash allows an algorithm to calculate a hash value without having to rehash the entire string.

When hashing a substring, rolling hash will do an operation on the hash of previous substring to get the new hash from the old hash.

We want to perform rolling hash on ACGGT with $k = 3$. We will compute the hash for ACG, CGG then GGT.

The hash of CGG can be obtain by removing "A" from the previous hash (ACG) and adding the hash of "G" at the end. Likewise, GGT will be obtain from CGG.

Rolling hash

Definition: a k -mer is a substring of length k of a string.

Rolling hash

Definition: a k -mer is a substring of length k of a string.

A DNA Sequence S: ATTACCGTGTTAGG

A 5-mer of S: ATTAC

Rolling hash

Definition: a k -mer is a substring of length k of a string.

A DNA Sequence S: ATTACCGTGTTAGG

A 5-mer of S: ATTAC

A 3-mer of S: TGT

Rolling hash

Definition: a k -mer is a substring of length k of a string.

A DNA Sequence S: ATTACCGTGTAGG

A 5-mer of S: ATTAC

A 3-mer of S: TGT

All 4-mers of S: ATTA

Rolling hash

Definition: a k -mer is a substring of length k of a string.

A DNA Sequence S: ATTACCGTTAGG

A 5-mer of S: ATTAC

A 3-mer of S: TGT

All 4-mers of S: ATTA
TTAC

Rolling hash

Definition: a k -mer is a substring of length k of a string.

A DNA Sequence S: ATTACCGTTAGG

A 5-mer of S: ATTAC

A 3-mer of S: TGT

All 4-mers of S: ATTA
TTAC
TACC

Rolling hash

Definition: a k -mer is a substring of length k of a string.

A DNA Sequence S: ATTACCGTTAGG

A 5-mer of S: ATTAC

A 3-mer of S: TGT

All 4-mers of S: ATTA
TTAC
TACC
ACCG

Rolling hash

Definition: a k -mer is a substring of length k of a string.

A DNA Sequence S: ATTACCGTGTTAGG

A 5-mer of S: ATTAC

A 3-mer of S: TGT

All 4-mers of S: ATTA
TTAC
TACC
ACCG
CCGT

Rolling hash

Definition: a k -mer is a substring of length k of a string.

A DNA Sequence S: ATTACCGTGTTAGG

A 5-mer of S: ATTAC

A 3-mer of S: TGT

All 4-mers of S: ATTA
TTAC
TACC
ACCG
CCGT
CGTG

Rolling hash

Definition: a k -mer is a substring of length k of a string.

A DNA Sequence S: ATTACCGTGTTAGG

A 5-mer of S: ATTAC

A 3-mer of S: TGT

All 4-mers of S: ATTA
TTAC
TACC
ACCG
CCGT
CGTG
GTGT

Rolling hash

Definition: a k -mer is a substring of length k of a string.

A DNA Sequence S: ATTACCGTGTTAGG

A 5-mer of S: ATTAC

A 3-mer of S: TGT

All 4-mers of S: ATTA
TTAC
TACC
ACCG
CCGT
CGTG
GTGT
TGTT

Rolling hash

Definition: a k -mer is a substring of length k of a string.

A DNA Sequence S: ATTACCGTGTTAGG

A 5-mer of S: ATTAC

A 3-mer of S: TGT

All 4-mers of S: ATTA
TTAC
TACC
ACCG
CCGT
CGTG
GTGT
TGTT
GTTA

Rolling hash

Definition: a k -mer is a substring of length k of a string.

A DNA Sequence S: ATTACCGTGTTAGG

A 5-mer of S: ATTAC

A 3-mer of S: TGT

All 4-mers of S: ATTA
TTAC
TACC
ACCG
CCGT
CGTG
GTGT
TGTT
GTTA
TTAG

Rolling hash

Definition: a k -mer is a substring of length k of a string.

A DNA Sequence S: ATTACCGTGTTAGG

A 5-mer of S: ATTAC

A 3-mer of S: TGT

All 4-mers of S: ATTA
TTAC
TACC
ACCG
CCGT
CGTG
GTGT
TGTT
GTTA
TTAG
TAGG

Rolling hash

Definition: a k -mer is a substring of length k of a string.

Rolling hash will use all k -mers of the string.

Rolling hash

A possible hash function

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$|\Sigma| = a = 4$

$A = 0, C = 1, G = 2, T = 3$

$k = 4$

Rolling hash

A possible hash function

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$$|\Sigma| = a = 4$$

$$A = 0, C = 1, G = 2, T = 3$$

$$k = 4$$

$$\text{Hash function: } H = c_1 * a^{k-1} + c_2 * a^{k-2} + c_3 * a^{k-3} + c_4 * a^{k-4}$$

Rolling hash

A possible hash function

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$$|\Sigma| = a = 4$$

$$A = 0, C = 1, G = 2, T = 3$$

$$k = 4$$

$$\text{Hash function: } H = c_1 * a^{k-1} + c_2 * a^{k-2} + c_3 * a^{k-3} + c_4 * a^{k-4}$$

$$\text{Hash of ACCG: } H = 0 * 4^3 + 1 * 4^2 + 1 * 4^1 + 2 * 4^0 = 22$$

Rolling hash

A possible hash function

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$$|\Sigma| = a = 4$$

$$A = 0, C = 1, G = 2, T = 3$$

$$k = 4$$

$$\text{Hash function: } H = c_1 * a^{k-1} + c_2 * a^{k-2} + c_3 * a^{k-3} + c_4 * a^{k-4}$$

$$\text{Hash of ACCG: } H = 0 * 4^3 + 1 * 4^2 + 1 * 4^1 + 2 * 4^0 = 22$$

$$\text{Hash of CCGT: } H = 1 * 4^3 + 1 * 4^2 + 2 * 4^1 + 3 * 4^0 = 91$$

Rolling hash

A possible hash function

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$$|\Sigma| = a = 4$$

$$A = 0, C = 1, G = 2, T = 3$$

$$k = 4$$

$$\text{Hash function: } H = c_1 * a^{k-1} + c_2 * a^{k-2} + c_3 * a^{k-3} + c_4 * a^{k-4}$$

$$\text{Hash of ACCG: } H = 0 * 4^3 + 1 * 4^2 + 1 * 4^1 + 2 * 4^0$$

$$\text{Hash of CCGT: } H = 1 * 4^3 + 1 * 4^2 + 2 * 4^1 + 3 * 4^0$$

Rolling hash

A possible hash function

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$$|\Sigma| = a = 4$$

$$A = 0, C = 1, G = 2, T = 3$$

$$k = 4$$

$$\text{Hash function: } H = c_1 * a^{k-1} + c_2 * a^{k-2} + c_3 * a^{k-3} + c_4 * a^{k-4}$$

Rolling hash

A possible hash function

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$$|\Sigma| = a = 4$$

$$A = 0, C = 1, G = 2, T = 3$$

$$k = 4$$

$$\text{Hash function: } H = c_1 * a^{k-1} + c_2 * a^{k-2} + c_3 * a^{k-3} + c_4 * a^{k-4}$$

$$\text{Hash of ACCG: } H = 0 * 4^3 + 1 * 4^2 + 1 * 4^1 + 2 * 4^0 = 22$$

Rolling hash

A possible hash function

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$|\Sigma| = a = 4$

$A = 0, C = 1, G = 2, T = 3$

$k = 4$

Hash function: $H = c_1 * a^{k-1} + c_2 * a^{k-2} + c_3 * a^{k-3} + c_4 * a^{k-4}$

Hash of ACCG: $H = 0 * 4^3 + 1 * 4^2 + 1 * 4^1 + 2 * 4^0 = 22$

Remove left letter from hash value:

Rolling hash

A possible hash function

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$$|\Sigma| = a = 4$$

$$A = 0, C = 1, G = 2, T = 3$$

$$k = 4$$

$$\text{Hash function: } H = c_1 * a^{k-1} + c_2 * a^{k-2} + c_3 * a^{k-3} + c_4 * a^{k-4}$$

$$\text{Hash of ACCG: } H = 0 * 4^3 + 1 * 4^2 + 1 * 4^1 + 2 * 4^0 = 22$$

$$\text{Remove left letter from hash value: } H - = (\text{val}(A)) * 4^3)$$

Rolling hash

A possible hash function

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$$|\Sigma| = a = 4$$

$$A = 0, C = 1, G = 2, T = 3$$

$$k = 4$$

$$\text{Hash function: } H = c_1 * a^{k-1} + c_2 * a^{k-2} + c_3 * a^{k-3} + c_4 * a^{k-4}$$

$$\text{Hash of CCG: } H = 1 * 4^2 + 1 * 4^1 + 2 * 4^0 = 22$$

$$\text{Remove left letter from hash value: } H - = (\text{val}(A)) * 4^3)$$

Rolling hash

A possible hash function

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$$|\Sigma| = a = 4$$

$$A = 0, C = 1, G = 2, T = 3$$

$$k = 4$$

$$\text{Hash function: } H = c_1 * a^{k-1} + c_2 * a^{k-2} + c_3 * a^{k-3} + c_4 * a^{k-4}$$

$$\text{Hash of CCG: } H = 1 * 4^2 + 1 * 4^1 + 2 * 4^0 = 22$$

$$\text{Remove left letter from hash value: } H - = (\text{val}(A)) * 4^3$$

All k are increased:

Rolling hash

A possible hash function

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$$|\Sigma| = a = 4$$

$$A = 0, C = 1, G = 2, T = 3$$

$$k = 4$$

$$\text{Hash function: } H = c_1 * a^{k-1} + c_2 * a^{k-2} + c_3 * a^{k-3} + c_4 * a^{k-4}$$

$$\text{Hash of CCG: } H = 1 * 4^2 + 1 * 4^1 + 2 * 4^0 = 22$$

$$\text{Remove left letter from hash value: } H - = (\text{val}(A)) * 4^3)$$

$$\text{All } ^\wedge \text{ are increased: } H * = 4 \text{ (value of } a)$$

Rolling hash

A possible hash function

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$$|\Sigma| = a = 4$$

$$A = 0, C = 1, G = 2, T = 3$$

$$k = 4$$

$$\text{Hash function: } H = c_1 * a^{k-1} + c_2 * a^{k-2} + c_3 * a^{k-3} + c_4 * a^{k-4}$$

$$\text{Hash of CCGx: } H = 1 * 4^3 + 1 * 4^2 + 2 * 4^1 = 88$$

$$\text{Remove left letter from hash value: } H - = (\text{val}(A)) * 4^3)$$

$$\text{All } ^\wedge \text{ are increased: } H * = 4 \text{ (value of } a)$$

Rolling hash

A possible hash function

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$$|\Sigma| = a = 4$$

$$A = 0, C = 1, G = 2, T = 3$$

$$k = 4$$

$$\text{Hash function: } H = c_1 * a^{k-1} + c_2 * a^{k-2} + c_3 * a^{k-3} + c_4 * a^{k-4}$$

$$\text{Hash of CCGx: } H = 1 * 4^3 + 1 * 4^2 + 2 * 4^1 = 88$$

$$\text{Remove left letter from hash value: } H - = (\text{val}(A)) * 4^3)$$

$$\text{All } ^\wedge \text{ are increased: } H * = 4 \text{ (value of } a)$$

Add the new last character value:

Rolling hash

A possible hash function

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$$|\Sigma| = a = 4$$

$$A = 0, C = 1, G = 2, T = 3$$

$$k = 4$$

$$\text{Hash function: } H = c_1 * a^{k-1} + c_2 * a^{k-2} + c_3 * a^{k-3} + c_4 * a^{k-4}$$

$$\text{Hash of CCGx: } H = 1 * 4^3 + 1 * 4^2 + 2 * 4^1 = 88$$

$$\text{Remove left letter from hash value: } H - = (\text{val}(A)) * 4^3)$$

$$\text{All } ^\wedge \text{ are increased: } H * = 4 \text{ (value of } a)$$

$$\text{Add the new last character value: } H + = (\text{val}(T)) * 4^0)$$

Rolling hash

A possible hash function

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$$|\Sigma| = a = 4$$

$$A = 0, C = 1, G = 2, T = 3$$

$$k = 4$$

$$\text{Hash function: } H = c_1 * a^{k-1} + c_2 * a^{k-2} + c_3 * a^{k-3} + c_4 * a^{k-4}$$

$$\text{Hash of CCGT: } H = 1 * 4^3 + 1 * 4^2 + 2 * 4^1 + 3 * 4^0 = 91$$

$$\text{Remove left letter from hash value: } H - = (\text{val}(A)) * 4^3)$$

$$\text{All } ^\wedge \text{ are increased: } H * = 4 \text{ (value of } a)$$

$$\text{Add the new last character value: } H + = (\text{val}(T)) * 4^0)$$

Rolling hash

The binary way of seeing it

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$|\Sigma| = a = 4$

$A = 00, C = 01, G = 10, T = 11$

$k = 4$

Rolling hash

The binary way of seeing it

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$|\Sigma| = a = 4$

$A = 00, C = 01, G = 10, T = 11$

$k = 4$

Hash of ACCG: $H = 00\ 01\ 01\ 10$ (10110 is 22)

Rolling hash

The binary way of seeing it

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$$|\Sigma| = a = 4$$

$$A = 00, C = 01, G = 10, T = 11$$

$$k = 4$$

Hash of ACCG: $H = 00\ 01\ 01\ 10$ (10110 is 22)

Remove left letter from hash value:

$$H \& = 00111111 \rightarrow H = 00\ 01\ 01\ 10 (\mathcal{O}(1))$$

Rolling hash

The binary way of seeing it

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$$|\Sigma| = a = 4$$

$$A = 00, C = 01, G = 10, T = 11$$

$$k = 4$$

Hash of ACCG: $H = 00\ 01\ 01\ 10$ (10110 is 22)

Remove left letter from hash value:

$$H \& = 00111111 \rightarrow H = 00\ 01\ 01\ 10 (\mathcal{O}(1))$$

All ^are increased:

$$H \ll 2 \rightarrow H = 00\ 01\ 01\ 10\ 00 (\mathcal{O}(1))$$

Rolling hash

The binary way of seeing it

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$$|\Sigma| = a = 4$$

$$A = 00, C = 01, G = 10, T = 11$$

$$k = 4$$

Hash of ACCG: $H = 00\ 01\ 01\ 10$ (10110 is 22)

Remove left letter from hash value:

$$H \& = 00111111 \rightarrow H = 00\ 01\ 01\ 10 (\mathcal{O}(1))$$

All ^are increased:

$$H \ll 2 \rightarrow H = 00\ 01\ 01\ 10\ 00 (\mathcal{O}(1))$$

Add the new last character value:

$$H + \text{val}(T) \rightarrow H = 00\ 01\ 01\ 10\ 11 (\mathcal{O}(1))$$

Rolling hash

The binary way of seeing it

$S = \text{ACCGT}$ Let's take $\Sigma = A, C, G, T$

$$|\Sigma| = a = 4$$

$$A = 00, C = 01, G = 10, T = 11$$

$$k = 4$$

Hash of ACCG: $H = 00\ 01\ 01\ 10$ (10110 is 22)

Remove left letter from hash value:

$$H \& = 00111111 \rightarrow H = 00\ 01\ 01\ 10 (\mathcal{O}(1))$$

All ^are increased:

$$H \ll 2 \rightarrow H = 00\ 01\ 01\ 10\ 00 (\mathcal{O}(1))$$

Add the new last character value:

$$H + \text{val}(T) \rightarrow H = 00\ 01\ 01\ 10\ 11 (\mathcal{O}(1))$$

Hash of CCGT: $H = 01\ 01\ 10\ 11$ (1011011 is 91)

Rabin–Karp using rolling hash

With the rolling hash, we can use the naive algorithm.

Rabin–Karp using rolling hash

With the rolling hash, we can use the naive algorithm.

The best- and average-case running time of Rabin-Karp is $\mathcal{O}(|T| + |P|)$. The rolling hash step takes $\mathcal{O}(|T|)$ time and once the algorithm finds a potential match, it must verify each letter to make sure that the match is true and not the result of a hashing collision: therefore it must check each of the $|P|$ letters in the pattern.

Rabin–Karp using rolling hash

With the rolling hash, we can use the naive algorithm.

The best- and average-case running time of Rabin-Karp is $\mathcal{O}(|T| + |P|)$. The rolling hash step takes $\mathcal{O}(|T|)$ time and once the algorithm finds a potential match, it must verify each letter to make sure that the match is true and not the result of a hashing collision: therefore it must check each of the $|P|$ letters in the pattern.

The rolling hash presented here is simplistic, for teaching purpose. In Python, the naive algorithm is faster. In the real hash function of Rabin-Karp, all the operations are done modulo a prime number to avoid dealing with large numbers. For the sake of readability and simplicity, the modulo has been excluded, but the calculations still hold when modulo is present.

A bit more... Aho-Corasick

Aho-Corasick algorithm indexes patterns (in a modified trie with links) and search on the text, only one time each letter.

A bit more... Aho-Corasick

Aho-Corasick algorithm indexes patterns (in a modified trie with links) and search on the text, only one time each letter.

It finds all occurrences of all patterns at once. One of the best for this task.

A bit more... Aho-Corasick

Aho-Corasick algorithm indexes patterns (in a modified trie with links) and search on the text, only one time each letter.

It finds all occurrences of all patterns at once. One of the best for this task.

Time complexity: $\mathcal{O}(|T| + |P| + |occ|)$

Thanks for your attention!



Any question?

<https://www.hackerearth.com/fr/practice/notes/exact-string-matching-algorithms/>
<https://brilliant.org/wiki/rabin-karp-algorithm/>