Competitive Programming
Winter Term 23/24

Strings

Kirill Simonov
Algorithm Engineering Group
HPI

Michael Zündorf
Scalable Algorithms Group
KIT
Outline

- Prefix Function
- String Hashing
- Suffix Array
- Aho-Corasick
Prefix Function

prefix function $p$ and $z$ function are two fundamental string algorithms
Prefix Function

Prefix function $p$ and $z$ function are two fundamental string algorithms.

- $p(i)$ is the length of the longest proper prefix of $t$ that is a suffix of $t[0..i]$.

\[
p(i) = \text{length of the longest proper prefix of } t \text{ that is a suffix of } t[0..i]
\]

\[
t = \text{a b a b a a b c a b}
\]

\[
p = \text{0 0 1 2 3 1 2 0 1 2}
\]
Prefix Function

prefix function $p$ and $z$ function are two fundamental string algorithms

- $p(i)$ is the length of the longest proper prefix of $t$ that is a suffix of $t[0..i]$
- $z(i)$ is the length of the longest common prefix of $t$ and $t[i..n-1]$

$t = \text{a b a b a a b c a b}$

$p = 0 \; 0 \; 1 \; 2 \; 3 \; 1 \; 2 \; 0 \; 1 \; 2$

$z = 0 \; 0 \; 3 \; 0 \; 1 \; 2 \; 0 \; 0 \; 2 \; 0$
Prefix Function

Prefix function $p$ and $z$ function are two fundamental string algorithms:

- $p(i)$ is the length of the longest proper prefix of $t$ that is a suffix of $t[0..i]$
- $z(i)$ is the length of the longest common prefix of $t$ and $t[i \ldots n − 1]$

string $s$;

```c
int p[1<<22], i, j;
void prefix() {
    for(; i < size(s); p[i] = j + (s[i]==s[j]))
        for(j = p[i++]; j && s[i] - s[j]; j = p[j-1]);
}
```

$\begin{array}{ccccccccccc}
t & a & b & a & b & a & a & b & c & a & b \\
p & 0 & 0 & 1 & 2 & 3 & 1 & 2 & 0 & 1 & 2 \\
z & 0 & 0 & 3 & 0 & 1 & 2 & 0 & 0 & 2 & 0 \\
\end{array}$
Prefix Function

Prefix function $p$ and $z$ function are two fundamental string algorithms

- $p(i)$ is the length of the longest proper prefix of $t$ that is a suffix of $t[0..i]$
- $z(i)$ is the length of the longest common prefix of $t$ and $t[i..n−1]$

```plaintext
\[
t = \texttt{a b a b a a b c a b}
\]
\[
p = \{0, 0, 1, 2, 3, 1, 2, 0, 1, 2\}
\]
\[
z = \{0, 0, 3, 0, 1, 2, 0, 0, 2, 0\}
\]
```

```
string s;
int p[1<<22], i, j;

void prefix() {
    for(; i < size(s); p[i] = j + (s[i]==s[j]))
        for(j = p[i++]; j && s[i] - s[j]; j = p[j-1]);
}
```
Prefix Function

prefix function $p$ and $z$ function are two fundamental string algorithms

- $p(i)$ is the length of the longest proper prefix of $t$ that is a suffix of $t[0..i]$
- $z(i)$ is the length of the longest common prefix of $t$ and $t[i..n-1]$

```
p[i] = j + (s[i] == s[j])
for(j = p[i++]; j && s[i] - s[j]; j = p[j-1]);
```

string $s$;
int $p[1<<22]$, $i$, $j$;
void prefix() {
  for(; $i$ < size(s); $p[i]$ = $j$ + (s[$i$] == s[$j$]))
    for($j$ = $p[i++]$; $j$ && s[$i$] - s[$j$]; $j$ = $p[j-1]$);
}

```
t = a b a b a a b c a b
$\text{p} =$ 0 0 1 2 3 1 2 0 1 2
$\text{z} =$ 0 0 3 0 1 2 0 0 2 0
```

find period of periodic string?

```
n = 10
ababababab
0012345678
```

```
n = 10
ababababab
0012345678
```

```
n = 10
ababababab
0080604020
```
#hashing
String Hashing

target application: compare two strings of length $n$
String Hashing

target application: compare two strings of length $n$

- character by character takes $O(n)$
String Hashing

target application: compare two strings of length $n$

- character by character takes $O(n)$
- convert strings to integers (hashing) and compare ints in $O(1) + \text{cost}(\text{hashing})$
String Hashing

target application: compare two strings of length $n$

- character by character takes $O(n)$
- convert strings to integers (hashing) and compare ints in $O(1) + \text{cost(hashing)}$

if hash is different then strings are different
String Hashing

target application: compare two strings of length $n$

- character by character takes $O(n)$
- convert strings to integers (hashing) and compare ints in $O(1) + \text{cost(hashing)}$

  if hash is different then strings are different

  if hash is equal then strings are likely equal
  
  (depends on the quality of our hash function)
String Hashing

target application: compare two strings of length $n$

- character by character takes $O(n)$
- convert strings to integers (hashing) and compare ints in $O(1) + cost(hashing)$

  if hash is different then strings are different
  if hash is equal then strings are likely equal
    (depends on the quality of our hash function)

when is this helpful:
String Hashing

target application: compare two strings of length $n$

- character by character takes $O(n)$
- convert strings to integers (hashing) and compare ints in $O(1) + \text{cost(hashing)}$
  
  if hash is different then strings are different
  
  if hash is equal then strings are likely equal
    (depends on the quality of our hash function)

when is this helpful:

- when comparing the same string multiple times we only need to hash it once
String Hashing

target application: compare two strings of length $n$

- character by character takes $O(n)$
- convert strings to integers (hashing) and compare ints in $O(1) + \text{cost(hashing)}$

  if hash is different then strings are different
  if hash is equal then strings are likely equal
  (depends on the quality of our hash function)

when is this helpful:

- when comparing the same string multiple times we only need to hash it once
- $\text{cost(hashing)}$ can be lower than $O(n)$ when hashing multiple similar strings
Polynomial Rolling Hashes

Given a string $s$ of length $n$. 
Polynomial Rolling Hashes

Given a string $s$ of length $n$.

- Base $a$ bigger than alphabet.
- Module $p$ prime and as big as possible.
Polynomial Rolling Hashes

Given a string $s$ of length $n$.

- Base $a$ bigger than alphabet.
- Module $p$ prime and as big as possible.

$$H(s) = s[0]a^0 + s[1]a^1 + s[2]a^2 + \cdots + s[n-1]a^{n-1} \mod p$$

$$= \sum_{i=0}^{n-1} s[i] \cdot a^i \mod p$$
Polynomial Rolling Hashes

Given a string $s$ of length $n$.

- Base $a$ bigger than alphabet.
- Module $p$ prime and as big as possible.

$$H(s) = s[0]a^0 + s[1]a^1 + s[2]a^2 + \cdots + s[n-1]a^{n-1} \mod p$$

$$= \sum_{i=0}^{n-1} s[i] \cdot a^i \mod p$$

$s = \text{race}$
Polynomial Rolling Hashes

Given a string $s$ of length $n$.

- Base $a$ bigger than alphabet.
- Module $p$ prime and as big as possible.

$$H(s) = s[0]a^0 + s[1]a^1 + s[2]a^2 + \cdots + s[n - 1]a^{n-1} \mod p$$

$$= \sum_{i=0}^{n-1} s[i] \cdot a^i \mod p$$

$H(s) = ra^0 + aa^1 + ca^2 + ea^3$
Polynomial Rolling Hashes

Given a string $s$ of length $n$.

- Base $a$ bigger than alphabet.
- Module $p$ prime and as big as possible.

$$H(s) = s[0]a^0 + s[1]a^1 + s[2]a^2 + \cdots + s[n-1]a^{n-1} \mod p$$

$$= \sum_{i=0}^{n-1} s[i] \cdot a^i \mod p$$

$s = r\ a\ c\ e$

$H(s) = ra^0 + aa^1 + ca^2 + ea^3$

How good is this hash function?
Polynomial Rolling Hashes

Assume we have hashes for strings $s, t$. How to compute the hash of $st$?

$H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p$
Assume we have hashes for strings $s, t$. How to compute the hash of $st$?

$$H(s) = \left(\sum_{i=0}^{n-1} a^i \cdot s[i]\right) \mod p$$
Polynomial Rolling Hashes

Assume we have hashes for strings $s, t$. How to compute the hash of $st$?

$$H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p$$

$s = a_0 a_1 a_2 a_3$

$t = a_0 a_1 a_2 a_3$

$s = \text{race}$

$t = \text{mice}$

$$H(s) + H(t) = a_0 a_1 a_2 a_3 a_0 a_1 a_2 a_3$$

$$s = \text{race}$$

$$t = \text{mice}$$
Polynomial Rolling Hashes

Assume we have hashes for strings $s, t$. How to compute the hash of $st$?

$$H(s) = \left(\sum_{i=0}^{n-1} a^i \cdot s[i]\right) \mod p$$

\[ s = \begin{array}{c}
a^0 \\
a^1 \\
a^2 \\
a^3 \\
r \\
ace \end{array} \quad t = \begin{array}{c}
a^0 \\
a^1 \\
a^2 \\
a^3 \\
m \\
ic \end{array} \quad H(s) + H(t) = \begin{array}{c}
a^0 \\
a^1 \\
a^2 \\
a^3 \\
r \\
ace \end{array} + \begin{array}{c}
a^0 \\
a^1 \\
a^2 \\
a^3 \\
m \\
ic \end{array} = \text{wrong?} \]
Polynomial Rolling Hashes

Assume we have hashes for strings $s, t$. How to compute the hash of $st$?

\[
H(s) = \left( \sum_{i=0}^{n-1} a^i \cdot s[i] \right) \mod p
\]

\[
H(s) + H(t) = \left( \sum_{i=0}^{n-1} a^i \cdot s[i] \right) + \left( \sum_{i=0}^{n-1} a^i \cdot t[i] \right) \mod p
\]

- Is the result of adding the hashes of $s$ and $t$ correct? The coefficients on $t$ are too low.

$$s = r a c e$$

$$t = m i c e$$

$$H(s) + H(t) = r a c e m i c e$$
Polynomial Rolling Hashes

Assume we have hashes for strings $s, t$. How to compute the hash of $st$?

\[
H(s) = \left(\sum_{i=0}^{n-1} a^i \cdot s[i]\right) \mod p
\]

\[
s = \begin{array}{c}
  \text{r a c e}
  \\
  a^0 \ a^1 \ a^2 \ a^3
\end{array}
\]

\[
t = \begin{array}{c}
  \text{m i c e}
  \\
  a^0 \ a^1 \ a^2 \ a^3
\end{array}
\]

\[
H(s) + H(t) = \begin{array}{c}
  \text{r a c e m i c e}
  \\
  a^0 \ a^1 \ a^2 \ a^3 \ a^4 \ a^5 \ a^6 \ a^7
\end{array}
\]

= wrong? the coefficients on $t$ are too low

\[
H(st) = \begin{array}{c}
  \text{r a c e m i c e}
  \\
  a^0 \ a^1 \ a^2 \ a^3 \ a^4 \ a^5 \ a^6 \ a^7
\end{array}
\]
Polynomial Rolling Hashes

Assume we have hashes for strings $s$, $t$. How to compute the hash of $st$?

$$H(s) = \left( \sum_{i=0}^{n-1} a^i \cdot s[i] \right) \mod p$$

$H(s) + H(t) = \text{wrong? the coefficients on } t \text{ are too low}$

$H(s) + a^{|s|} \cdot H(t) = H(st)$
Polynomial Rolling Hashes

Assume we have the hash of $s[\ell, r]$. How to compute the hash of $s[\ell + 1, r + 1]$?

$$H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p$$
Polynomial Rolling Hashes

Assume we have the hash of $s[\ell, r]$. How to compute the hash of $s[\ell + 1, r + 1]$?

$$H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p$$
Polynomial Rolling Hashes

Assume we have the hash of $s[\ell, r]$. How to compute the hash of $s[\ell + 1, r + 1]$?

$H(s) = \left(\sum_{i=0}^{n-1} a^i \cdot s[i]\right) \mod p$
Polynomial Rolling Hashes

Assume we have the hash of $s[\ell, r]$. How to compute the hash of $s[\ell + 1, r + 1]$?

$H(s[\ell, r]) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p$

$H(s[\ell, r]) + a^{r+1-\ell} \cdot s[r + 1] - s[l]$
Polynomial Rolling Hashes

Assume we have the hash of $s[\ell, r]$. How to compute the hash of $s[\ell + 1, r + 1]$?

$$H(s[\ell, r]) + a^{r+1-\ell} \cdot s[r + 1] - s[\ell]$$
Polynomial Rolling Hashes

Assume we have the hash of $s[\ell, r]$. How to compute the hash of $s[\ell + 1, r + 1]$?

$H(s) = \left(\sum_{i=0}^{n-1} a^i \cdot s[i]\right) \mod p$

\[
H(s[\ell, r]) = (P^{r+1-\ell} \cdot s[r + 1] - s[\ell])
\]

\[
H(s[\ell + 1, r + 1]) = H(s[\ell, r]) + a^{r+1-\ell} \cdot s[r + 1] - s[\ell]
\]
Polynomial Rolling Hashes

Assume we have the hash of $s[\ell, r]$. How to compute the hash of $s[\ell + 1, r + 1]$?

\[
H(s[\ell, r]) + a^{r+1-\ell} \cdot s[r + 1] - s[\ell] = H(s[\ell + 1, r + 1])
\]

\[
H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p
\]
Polynomial Rolling Hashes

Assume we have the hash of every prefix of $s$. How to compute the hash of $s[\ell, r]$?

$$H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p$$

$s = \text{has sos race m ice}$

$H(s[\ell, r]) =$
Polynomial Rolling Hashes

Assume we have the hash of every prefix of $s$. How to compute the hash of $s[\ell, r]$?

\[
H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p
\]

$s = \text{has s o s r a c e m i c e}$

\[
H(s[\ell, r]) = H(s[0, r])
\]
Polynomial Rolling Hashes

Assume we have the hash of every prefix of $s$. How to compute the hash of $s[\ell, r]$?

$$H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p$$

Let $s = \text{has s o r a c e m i c e}$.

$$H(s[\ell, r]) = H(s[0, r]) - H(s[0, \ell - 1])$$
Polynomial Rolling Hashes

Assume we have the hash of every prefix of \( s \). How to compute the hash of \( s[\ell, r] \)?

\[
H(s) = (\sum_{i=0}^{n-1} a_i \cdot s[i]) \mod p
\]

\[
H(s[\ell, r]) = H(s[0, r]) - H(s[0, \ell - 1])
\]

\( s = \text{h a s s o s r a c e m i c e} \)
Polynomial Rolling Hashes

Assume we have the hash of every prefix of $s$. How to compute the hash of $s[\ell, r]$?

\[
H(s) = \left(\sum_{i=0}^{n-1} a^i \cdot s[i]\right) \mod p
\]

\[
H(s[\ell, r]) = H(s[0, r]) - H(s[0, \ell - 1]) + a^\ell
\]
Polynomial Rolling Hashes

Assume we have the hash of every prefix of $s$. How to compute the hash of $s[\ell, r]$?

$H(s[\ell, r]) = H(s[0, r]) - H(s[0, \ell - 1])$
Polynomial Rolling Hashes

Tips

\[ H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p \]
Polynomial Rolling Hashes

Tips

- precompute powers of $a$ and their inverse

$$H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p$$
Polynomial Rolling Hashes

Tips

- precompute powers of $a$ and their inverse
- range hashes w/o division or inverse by using descending powers of $a$

$$H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p$$
Polynomial Rolling Hashes

Tips

- precompute powers of \( a \) and their inverse
- range hashes w/o division or inverse by using descending powers of \( a \)

\[
H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p
\]
Polynomial Rolling Hashes

**Tips**

- precompute powers of $a$ and their inverse
- range hashes w/o division or inverse by using descending powers of $a$

$$H(s) = \left(\sum_{i=0}^{n-1} a^i \cdot s[i]\right) \mod p$$

$s = \text{h a s s o s r a c e m i c e}$

$$H(s) = \left(\sum_{i=0}^{n-1} a^{n-i-1} \cdot s[i]\right) \mod p$$
Polynomial Rolling Hashes

**Tips**

- Precompute powers of \( a \) and their inverse
- Range hashes w/o division or inverse by using descending powers of \( a \)

\[
H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p
\]

\[
H(s) = (\sum_{i=0}^{n-1} a^{n-i-1} \cdot s[i]) \mod p
\]

\[
H(s[\ell, r]) = H(s[0, r]) - a^{r-\ell+1} \cdot H(s[0, \ell - 1])
\]
Polynomial Rolling Hashes

**Tips**

- precompute powers of $a$ and their inverse
- range hashes w/o division or inverse by using descending powers of $a$

\[
H(s) = \left( \sum_{i=0}^{n-1} a^i \cdot s[i] \right) \mod p
\]

\[
H(s) = \left( \sum_{i=0}^{n-1} a^{n-i-1} \cdot s[i] \right) \mod p
\]

- don’t start character numbering at 0
Polynomial Rolling Hashes

Tips

- precompute powers of $a$ and their inverse
- range hashes w/o division or inverse by using descending powers of $a$

$$H(s) = \left(\sum_{i=0}^{n-1} a^i \cdot s[i]\right) \mod p$$

$$s=\text{hasaossracemice}$$

- don’t start character numbering at 0  why?
Polynomial Rolling Hashes

Tips

- precompute powers of $a$ and their inverse
- range hashes w/o division or inverse by using descending powers of $a$

$H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p$

$s = \text{h a s s o s r a c e m i c e}

\begin{align*}
H(s[\ell, r]) &= H(s[0, r]) - a^{r-\ell+1} \cdot H(s[0, \ell - 1]) \\
H(a) &= H(aaaa) = 0 \\
H(pizza) &= H(pizzaaa)
\end{align*}$

- don’t start character numbering at 0

why?
Polynomial Rolling Hashes

Collisions

\[ H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p \]
Polynomial Rolling Hashes

Collisions

- if we assume hashes to be uniform random, there is a collision with prob. $1/p$

$$H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p$$
Polynomial Rolling Hashes

Collisions

- if we assume hashes to be uniform random, there is a collision with prob. $1/p$

- birthday paradox says $\approx \sqrt{p}$ strings have a collision with prob. $> 1/2$
  with $p = 10^9 + 7$ that is $\approx 10^{4.5}$

$$H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p$$
Polynomial Rolling Hashes

Collisions

- if we assume hashes to be uniform random, there is a collision with prob. $1/p$

- birthday paradox says $\approx \sqrt{p}$ strings have a collision with prob. $> 1/2$
  
  with $p = 10^9 + 7$ that is $\approx 10^{4.5}$

- **Fix**: use a second (or more) hashes with another prime $p'$

\[
H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p
\]
Polynomial Rolling Hashes

Collisions

- if we assume hashes to be uniform random, there is a collision with prob. $1/p$
- birthday paradox says $\approx \sqrt{p}$ strings have a collision with prob. $> 1/2$
  with $p = 10^9 + 7$ that is $\approx 10^{4.5}$
- **Fix**: use a second (or more) hashes with another prime $p'$
  single collision with prob. $1/(pp')$ assuming uniform random hashes

$$H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \text{ mod } p$$
Polynomial Rolling Hashes

Collisions

- if we assume hashes to be uniform random, there is a collision with prob. $1/p$
- birthday paradox says $\approx \sqrt{p}$ strings have a collision with prob. $> 1/2$
  with $p = 10^9 + 7$ that is $\approx 10^{4.5}$
- **Fix**: use a second (or more) hashes with another prime $p'$
  single collision with prob. $1/(pp')$ assuming uniform random hashes

Hacking

- if your code is known, testcase with many collisions can be crafted against it

$$H(s) = \left( \sum_{i=0}^{n-1} a^i \cdot s[i] \right) \mod p$$

https://codeforces.com/blog/entry/60442
Polynomial Rolling Hashes

**Collisions**

- if we assume hashes to be uniform random, there is a collision with prob. $1/p$
- birthday paradox says $\approx \sqrt{p}$ strings have a collision with prob. $> 1/2$
  
  with $p = 10^9 + 7$ that is $\approx 10^{4.5}$

- **Fix:** use a second (or more) hashes with another prime $p'$
  
  single collision with prob. $1/(pp')$ assuming uniform random hashes

**Hacking**

- if your code is known, testcase with many collisions can be crafted against it

- **Fix:** randomize your base $a$ at runtime

\[ H(s) = \left( \sum_{i=0}^{n-1} a^i \cdot s[i] \right) \mod p \]
Polynomial Rolling Hashes

Collisions

- if we assume hashes to be uniform random, there is a collision with prob. $1/p$
- birthday paradox says $\approx \sqrt{p}$ strings have a collision with prob. $> 1/2$
  with $p = 10^9 + 7$ that is $\approx 10^{4.5}$
- **Fix:** use a second (or more) hashes with another prime $p'$
  single collision with prob. $1/(pp')$ assuming uniform random hashes

Hacking

- if your code is known, testcase with many collisions can be crafted against it
- **Fix:** randomize your base $a$ at runtime

Assume strings $s,t$ of length $n$. Collision when:

\[
H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p
\]
Polynomial Rolling Hashes

Collisions

- if we assume hashes to be uniform random, there is a collision with prob. $1/p$
- birthday paradox says $\approx \sqrt{p}$ strings have a collision with prob. $> 1/2$
  with $p = 10^9 + 7$ that is $\approx 10^{4.5}$
- **Fix:** use a second (or more) hashes with another prime $p'$
  single collision with prob. $1/(pp')$ assuming uniform random hashes

Hacking

- if your code is known, testcase with many collisions can be crafted against it
- **Fix:** randomize your base $a$ at runtime

Assume strings $s,t$ of length $n$. Collision when:

$$H(s) = \left(\sum_{i=0}^{n-1} a^i \cdot s[i]\right) \mod p$$

$$\sum_{i=0}^{n-1} a^i \cdot s[i] \equiv \sum_{i=0}^{n-1} a^i \cdot t[i] \mod p$$
Polynomial Rolling Hashes

Collisions

- if we assume hashes to be uniform random, there is a collision with prob. $1/p$
- birthday paradox says $\approx \sqrt{p}$ strings have a collision with prob. $> 1/2$
  with $p = 10^9 + 7$ that is $\approx 10^{4.5}$
- **Fix:** use a second (or more) hashes with another prime $p'$
  single collision with prob. $1/(pp')$ assuming uniform random hashes

Hacking

- if your code is known, testcase with many collisions can be crafted against it
- **Fix:** randomize your base $a$ at runtime

Assume strings $s, t$ of length $n$. Collision when:

$$H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p$$

$$\sum_{i=0}^{n-1} a^i \cdot s[i] \equiv \sum_{i=0}^{n-1} a^i \cdot t[i] \mod p$$

$$P(a) = \sum_{i=0}^{n-1} a^i \cdot (s[i] - t[i]) \equiv 0 \mod p$$

https://codeforces.com/blog/entry/60442
Polynomial Rolling Hashes

Collisions

- if we assume hashes to be uniform random, there is a collision with prob. $1/p$
- birthday paradox says $\approx \sqrt{p}$ strings have a collision with prob. $> 1/2$
  with $p = 10^9 + 7$ that is $\approx 10^{4.5}$

- **Fix:** use a second (or more) hashes with another prime $p'$
  single collision with prob. $1/(pp')$ assuming uniform random hashes

Hacking

- if your code is known, testcase with many collisions can be crafted against it
- **Fix:** randomize your base $a$ at runtime

Assume strings $s,t$ of length $n$. Collision when:

- $P(a)$ is polynomial of degree $n - 1$ over a field

$$H(s) = \left( \sum_{i=0}^{n-1} a^i \cdot s[i] \right) \mod p$$
Polynomial Rolling Hashes

Collisions

- if we assume hashes to be uniform random, there is a collision with prob. $1/p$
- birthday paradox says $\approx \sqrt{p}$ strings have a collision with prob. $> 1/2$
  with $p = 10^9 + 7$ that is $\approx 10^{4.5}$
- **Fix:** use a second (or more) hashes with another prime $p'$
  single collision with prob. $1/(pp')$ assuming uniform random hashes

Hacking

- if your code is known, testcase with many collisions can be crafted against it
- **Fix:** randomize your base $a$ at runtime

Assume strings $s,t$ of length $n$. Collision when:

- $P(a)$ is polynomial of degree $n - 1$ over a field
- at most $n - 1$ roots $\rightarrow$ collision prob. $< (n - 1)/p$

$$H(s) = (\sum_{i=0}^{n-1} a^i \cdot s[i]) \mod p$$
Given a string $s$ of length $n$. We call the substring $s[i \ldots n - 1]$ the $i$-th suffix of $s$. 
Suffix Array

- Given a string $s$ of length $n$. We call the substring $s[i \ldots n - 1]$ the $i$-th suffix of $s$.
- suffix array of $s$ contains lexicographic order of all suffixes of $s$. 
Suffix Array

- Given a string $s$ of length $n$. We call the substring $s[i \ldots n - 1]$ the $i$-th suffix of $s$.
- suffix array of $s$ contains lexicographic order of all suffixes of $s$.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$i$-th suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>racemice</td>
</tr>
<tr>
<td>1</td>
<td>acemice</td>
</tr>
<tr>
<td>2</td>
<td>cemice</td>
</tr>
<tr>
<td>3</td>
<td>emice</td>
</tr>
<tr>
<td>4</td>
<td>mice</td>
</tr>
<tr>
<td>5</td>
<td>ice</td>
</tr>
<tr>
<td>6</td>
<td>ce</td>
</tr>
<tr>
<td>7</td>
<td>e</td>
</tr>
</tbody>
</table>
Suffix Array

- Given a string $s$ of length $n$. We call the substring $s[i \ldots n-1]$ the $i$-th suffix of $s$.
- Suffix array of $s$ contains lexicographic order of all suffixes of $s$.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$i$-th suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>racemice</td>
</tr>
<tr>
<td>1</td>
<td>acemice</td>
</tr>
<tr>
<td>2</td>
<td>cemice</td>
</tr>
<tr>
<td>3</td>
<td>emice</td>
</tr>
<tr>
<td>4</td>
<td>mice</td>
</tr>
<tr>
<td>5</td>
<td>ice</td>
</tr>
<tr>
<td>6</td>
<td>ce</td>
</tr>
<tr>
<td>7</td>
<td>e</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$i$</th>
<th>$i$-th suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>acemice</td>
</tr>
<tr>
<td>6</td>
<td>ce</td>
</tr>
<tr>
<td>2</td>
<td>cemice</td>
</tr>
<tr>
<td>7</td>
<td>e</td>
</tr>
<tr>
<td>3</td>
<td>emice</td>
</tr>
<tr>
<td>5</td>
<td>ice</td>
</tr>
<tr>
<td>4</td>
<td>mice</td>
</tr>
<tr>
<td>0</td>
<td>racemice</td>
</tr>
</tbody>
</table>
Suffix Array

- Given a string $s$ of length $n$. We call the substring $s[i \ldots n - 1]$ the $i$-th suffix of $s$.
- suffix array of $s$ contains lexicographic order of all suffixes of $s$.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$i$-th suffix</th>
<th>$i$</th>
<th>$i$-th suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>racemice</td>
<td>1</td>
<td>acemice</td>
</tr>
<tr>
<td>1</td>
<td>acemice</td>
<td>6</td>
<td>ce</td>
</tr>
<tr>
<td>2</td>
<td>cemice</td>
<td>2</td>
<td>cemice</td>
</tr>
<tr>
<td>3</td>
<td>emice</td>
<td>7</td>
<td>e</td>
</tr>
<tr>
<td>4</td>
<td>mice</td>
<td>3</td>
<td>emice</td>
</tr>
<tr>
<td>5</td>
<td>ice</td>
<td>5</td>
<td>ice</td>
</tr>
<tr>
<td>6</td>
<td>ce</td>
<td>4</td>
<td>mice</td>
</tr>
<tr>
<td>7</td>
<td>e</td>
<td>0</td>
<td>racemice</td>
</tr>
</tbody>
</table>

- suffix array of “racemice” is $[1, 6, 2, 7, 3, 5, 4, 0]$
Suffix Array
Suffix Array

```
sorted(range(len(s)), key=lambda i: s[i:])
```
Suffix Array

\[
\text{sorted} \left( \text{range}(\text{len}(s)), \text{key} = \lambda \ i : s[i:] \right)
\]

- \( O(n \log n) \) comparisons
Suffix Array

\[ \text{sorted(range(len(s)), key=lambda i: s[i:])} \]

- \(O(n \log n)\) comparisons
- \(O(n)\) for each string comparison
Suffix Array

```python
sorted(range(len(s)), key=lambda i: s[i:])
```

- $O(n \log n)$ comparisons
- $O(n)$ for each string comparison

$\rightarrow O(n^2 \log n)$ in total
Suffix Array

\[
\text{sorted}(\text{range}(\text{len}(s)), \text{key} = \lambda i: s[i:])
\]

- \(O(n \log n)\) comparisons
- \(O(n)\) for each string comparison
  - \(\rightarrow O(n^2 \log n)\) in total
- let’s improve this with hashing
Suffix Array

```
sorted(range(len(s)), key=lambda i: s[i:])
```

- \(O(n \log n)\) comparisons
- \(O(n)\) for each string comparison
  - \(\rightarrow O(n^2 \log n)\) in total
- let’s improve this with hashing
- how to compare two strings lexicographically using hashing?
Suffix Array

```python
sorted(range(len(s)), key=lambda i: s[i:]),
```

- $O(n \log n)$ comparisons
- $O(n)$ for each string comparison
  - $\rightarrow O(n^2 \log n)$ in total
- let’s improve this with hashing
- how to compare two strings lexicographically using hashing?
  - find first different character (or LCP) with binary search. Then compare it.

LCP = longest common prefix

LCP:

```
cababca
```

```
cabcba
```
Suffix Array

```python
sorted(range(len(s)), key=lambda i: s[i:])
```

- $O(n \log n)$ comparisons
- $O(n)$ for each string comparison
  - $\rightarrow O(n^2 \log n)$ in total
- let’s improve this with hashing
- how to compare two strings lexicographically using hashing?
  - find first different character (or LCP) with binary search. Then compare it.
  - $O(\log n)$ for each string comparison
**Suffix Array**

```python
sorted(range(len(s)), key=lambda i: s[i:])
```

- \( O(n \log n) \) comparisons
- \( O(n) \) for each string comparison
  - \( \rightarrow O(n^2 \log n) \) in total
- let’s improve this with hashing

**how to compare two strings lexicographically using hashing?**

- find first different character (or LCP) with binary search. Then compare it.

- \( O(\log n) \) for each string comparison
  - \( \rightarrow O(n \log^2 n) \) in total
    - (very high constants in practice)

LCP = longest common prefix

LCP:  
```
cababca
```
```
cabcba
```
Suffix Array → Sort Cyclic Shifts

Given a string $s$ of length $n$. We call the string $s[i..n − 1] + s[0..i − 1]$ the $i$-th cyclic shift of $s$. 
Suffix Array → Sort Cyclic Shifts

Given a string $s$ of length $n$. We call the string $s[i..n-1] + s[0..i-1]$ the $i$-th cyclic shift of $s$.

- there is a cool $O(n \log n)$ DP to sort cyclic shifts
Given a string $s$ of length $n$. We call the string $s[i..n-1] + s[0..i-1]$ the $i$-th cyclic shift of $s$.

- there is a cool $O(n \log n)$ DP to sort cyclic shifts

idea:  
- append small character (e.g. $\$$) to $s$
Suffix Array → Sort Cyclic Shifts

Given a string $s$ of length $n$. We call the string $s[i..n-1] + s[0..i-1]$ the $i$-th cyclic shift of $s$.

- There is a cool $O(n \log n)$ DP to sort cyclic shifts.

**Idea:**
- Append small character (e.g. $\$) to $s$.
- Use cool DP to sort cyclic shifts of $s$.
Suffix Array → Sort Cyclic Shifts

Given a string $s$ of length $n$. We call the string $s[i..n − 1] + s[0..i − 1]$ the $i$-th cyclic shift of $s$.

- there is a cool $O(n \log n)$ DP to sort cyclic shifts

idea:
- append small character (e.g. $\$$) to $s$
- use cool DP to sort cyclic shifts of $s$$
- ignore first entry in result array
Suffix Array $\rightarrow$ Sort Cyclic Shifts

Given a string $s$ of length $n$. We call the string $s[i..n-1] + s[0..i-1]$ the $i$-th cyclic shift of $s$.

- there is a cool $O(n \log n)$ DP to sort cyclic shifts

**idea:**

- append small character (e.g. $\$$) to $s$
- use cool DP to sort cyclic shifts of $s$
- ignore first entry in result array

<table>
<thead>
<tr>
<th>$i$</th>
<th>$i$-th cyclic shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>$$racemice</td>
</tr>
<tr>
<td>1</td>
<td>acemice$$$</td>
</tr>
<tr>
<td>6</td>
<td>ce$$racemi</td>
</tr>
<tr>
<td>2</td>
<td>cemice$$ra</td>
</tr>
<tr>
<td>7</td>
<td>e$$racemic</td>
</tr>
<tr>
<td>3</td>
<td>emice$$rac</td>
</tr>
<tr>
<td>5</td>
<td>ice$$racem</td>
</tr>
<tr>
<td>4</td>
<td>mice$$race</td>
</tr>
<tr>
<td>0</td>
<td>racemice$$</td>
</tr>
</tbody>
</table>
Suffix Array → Sort Cyclic Shifts

Given a string $s$ of length $n$. We call the string $s[i...n-1] + s[0...i-1]$ the $i$-th cyclic shift of $s$.

- there is a cool $O(n \log n)$ DP to sort cyclic shifts

**idea:**
- append small character (e.g. $) to $s$
- use cool DP to sort cyclic shifts of $s$
- ignore first entry in result array

<table>
<thead>
<tr>
<th>$i$</th>
<th>$i$-th cyclic shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>$\text{racemice}$</td>
</tr>
<tr>
<td>1</td>
<td>acemice$\text{r}$</td>
</tr>
<tr>
<td>6</td>
<td>ce$\text{racemi}$</td>
</tr>
<tr>
<td>2</td>
<td>cemice$\text{ra}$</td>
</tr>
<tr>
<td>7</td>
<td>e$\text{racemic}$</td>
</tr>
<tr>
<td>3</td>
<td>emice$\text{rac}$</td>
</tr>
<tr>
<td>5</td>
<td>ice$\text{racem}$</td>
</tr>
<tr>
<td>4</td>
<td>mice$\text{race}$</td>
</tr>
<tr>
<td>0</td>
<td>racemice$</td>
</tr>
</tbody>
</table>
Suffix Array → Sort Cyclic Shifts

Given a string $s$ of length $n$. We call the string $s[i..n − 1] + s[0..i − 1]$ the $i$-th cyclic shift of $s$.

- there is a cool $O(n \log n)$ DP to sort cyclic shifts

**Idea:**
- append small character (e.g. $\$$) to $s$
- use cool DP to sort cyclic shifts of $s$
- ignore first entry in result array

<table>
<thead>
<tr>
<th>$i$</th>
<th>$i$-th suffix</th>
<th>$i$</th>
<th>$i$-th cyclic shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>acemice</td>
<td>8</td>
<td>$$$racemice</td>
</tr>
<tr>
<td>6</td>
<td>ce</td>
<td>1</td>
<td>acemice$r$</td>
</tr>
<tr>
<td>2</td>
<td>cemice</td>
<td>6</td>
<td>ce$racemice$</td>
</tr>
<tr>
<td>7</td>
<td>e</td>
<td>2</td>
<td>cemice$ra$</td>
</tr>
<tr>
<td>3</td>
<td>emice</td>
<td>7</td>
<td>e$racemice$</td>
</tr>
<tr>
<td>5</td>
<td>ice</td>
<td>3</td>
<td>emice$rac$</td>
</tr>
<tr>
<td>4</td>
<td>mice</td>
<td>5</td>
<td>ice$racem$</td>
</tr>
<tr>
<td>0</td>
<td>racemice</td>
<td>4</td>
<td>mice$race$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>racemice$</td>
</tr>
</tbody>
</table>
Sort Cyclic Shifts

Assume cyclic substrings of length $2^k$ are already sorted

How do we sort cyclic substrings of length $2^{k+1}$?
Sort Cyclic Shifts

Assume cyclic substrings of length $2^k$ are already sorted

How do we sort cyclic substrings of length $2^{k+1}$?

- compare substrings of length $2^{k+1}$ by comparing their first and second halves
Sort Cyclic Shifts

Assume cyclic substrings of length $2^k$ are already sorted

How do we sort cyclic substrings of length $2^{k+1}$?

- compare substrings of length $2^{k+1}$ by comparing their first and second halves
- for this, we need to know if two length $2^k$ substrings are equal
Sort Cyclic Shifts

Assume cyclic substrings of length $2^k$ are already sorted

How do we sort cyclic substrings of length $2^{k+1}$?

- compare substrings of length $2^{k+1}$ by comparing their first and second halves
- for this, we need to know if two length $2^k$ substrings are equal
- let’s also compute equivalence classes
Sort Cyclic Shifts

Assume cyclic substrings of length $2^k$ are already sorted

How do we sort cyclic substrings of length $2^{k+1}$?

- compare substrings of length $2^{k+1}$ by comparing their first and second halves
- for this, we need to know if two length $2^k$ substrings are equal
- let’s also compute equivalence classes

How do we build equivalence classes?
Sort Cyclic Shifts

Assume cyclic substrings of length $2^k$ are already sorted

How do we sort cyclic substrings of length $2^{k+1}$?

- compare substrings of length $2^{k+1}$ by comparing their first and second halves
- for this, we need to know if two length $2^k$ substrings are equal
- let’s also compute equivalence classes

How do we build equivalence classes?

- give first substring in sorted order
eq. class $0$
Sort Cyclic Shifts

Assume cyclic substrings of length $2^k$ are already sorted

How do we sort cyclic substrings of length $2^{k+1}$?

- compare substrings of length $2^{k+1}$ by comparing their first and second halves
- for this, we need to know if two length $2^k$ substrings are equal
- let’s also compute equivalence classes

How do we build equivalence classes?

- give first substring in sorted order
eq. class 0
- go through sorted substrings and give each the eq. class of previous substring if they’re equal or higher class if they’re different
Sort Cyclic Shifts

Assume cyclic substrings of length $2^k$ are already sorted

How do we sort cyclic substrings of length $2^{k+1}$?

- compare substrings of length $2^{k+1}$ by comparing their first and second halves
- for this, we need to know if two length $2^k$ substrings are equal
- let’s also compute equivalence classes

How do we build equivalence classes?

- give first substring in sorted order eq. class 0
- go through sorted substrings and give each the eq. class of previous substring if they’re equal or higher class if they’re different

Sort Cyclic Shifts
initialize length 1 (= $2^0$) equivalence classes for $k \in [0, \log n)$:

sort length $2^{k+1}$ substrings
build classes for length $2^{k+1}$ strings
Sort Cyclic Shifts

Assume cyclic substrings of length $2^k$ are already sorted

How do we sort cyclic substrings of length $2^{k+1}$?

- compare substrings of length $2^{k+1}$ by comparing their first and second halves
- for this, we need to know if two length $2^k$ substrings are equal
- let’s also compute equivalence classes

How do we build equivalence classes?

- give first substring in sorted order
  eq. class 0
- go through sorted substrings and give each the eq. class of previous substring if they’re equal or higher class if they’re different

initialize length 1 (= $2^0$) equivalence classes
for $k \in [0, \log n)$:
  sort length $2^{k+1}$ substrings
  build classes for length $2^{k+1}$ strings

$\rightarrow O(n \log^2 n)$ if using std::sort
Sort Cyclic Shifts

// p stands for permutation and holds the result
int n = size(s);
vector p(n, 0);
iota(all(p), 0);

// init length 1 (=2^0) eq. classes
rep(i, n) c[0][i] = s[i];

for (int k = 0; (1 << k) < n; k++) {
  // sort length 2^{k+1} (cyclic) substrings
  auto parts = [&](int i) { return pair(c[k][i], c[k][(i + (1 << k)) % n]); };
  sort(all(p), [&](int a, int b) { return parts(a) < parts(b); });
  // build length 2^{k+1} eq. classes
  rep(i, n - 1) c[k + 1][p[i + 1]] = c[k + 1][p[i]] + (parts(p[i + 1]) != parts(p[i]));
}
Sort Cyclic Shifts

```cpp
// p stands for permutation and holds the result
int n = size(s);
vector p(n,0);
iota(all(p),0);

// init length 1 (=2^0) eq. classes
vector<vector<int>> c(ceil(log2(n))+1, vector(n,0));
rep(i,n) c[0][i] = s[i];

for (int k = 0; (1<<k) < n; k++) {
    // sort length 2^{k+1} (cyclic) substrings
    auto parts = [&](int i) { return pair(c[k][i], c[k][(i+(1<<k))%n]); };
    sort(all(p), [&](int a, int b) { return parts(a) < parts(b); });

    // build length 2^{k+1} eq. classes
    rep(i,n-1) c[k+1][p[i+1]] = c[k+1][p[i]] + (parts(p[i+1])!=parts(p[i]));
}
```
Sort Cyclic Shifts

// p stands for permutation and holds the result
int n = size(s);
vector p(n,0);
iota(all(p),0);

// init length 1 (=2^0) eq. classes
vector<vector<int>> c(ceil(log2(n))+1, vector(n,0));
rep(i,n) c[0][i] = s[i];

for (int k = 0; (1<<k) < n; k++) {
    // sort length 2^{k+1} (cyclic) substrings
    auto parts = [&](int i) { return pair(c[k][i], c[k][(i+(1<<k))%n]); }; 
    sort(all(p), [&](int a, int b) { return parts(a) < parts(b); });
}

Sort Cyclic Shifts

// p stands for permutation and holds the result
int n = size(s);
vector p(n,0);
iota(all(p),0);

// init length 1 (=2^0) eq. classes
vector<vector<int>> c(ceil(log2(n))+1, vector(n,0));
rep(i,n) c[0][i] = s[i];

for (int k = 0; (1<<k) < n; k++) {
    // sort length 2^{k+1} (cyclic) substrings
    auto parts = [&](int i) { return pair(c[k][i], c[k][(i+(1<<k))%n]); };,
    sort(all(p), [&](int a, int b) { return parts(a) < parts(b); });

    // build length 2^{k+1} eq. classes
    rep(i,n-1) c[k+1][p[i+1]] = c[k+1][p[i]] + (parts(p[i+1])!=parts(p[i]));
}
Sort Cyclic Shifts

WHERE \( O(N \log N) \)?
Sort Cyclic Shifts

- equiv. classes of the parts do not exceed $n$
Sort Cyclic Shifts

- equiv. classes of the parts do not exceed $n$
- how to do counting sort on pairs?
Sort Cyclic Shifts

- equiv. classes of the parts do not exceed $n$
- how to do counting sort on pairs?
- use radix sort
Sort Cyclic Shifts

- equiv. classes of the parts do not exceed $n$
- how to do counting sort on pairs?
- use radix sort
  - sort by second half
Sort Cyclic Shifts

- equiv. classes of the parts do not exceed $n$
- how to do counting sort on pairs?
- use radix sort
  - sort by second half
  - then stable sort by first half (with counting sort)
Sort Cyclic Shifts

- equiv. classes of the parts do not exceed $n$
- how to do counting sort on pairs?
- use radix sort
  - sort by second half
  - then stable sort by first half (with counting sort)
- observation: substrings are already sorted by their first $2^k$ characters from last iteration
Sort Cyclic Shifts

- equiv. classes of the parts do not exceed \( n \)
- how to do counting sort on pairs?
- use radix sort
  - sort by second half
  - then stable sort by first half (with counting sort)
- observation: substrings are already sorted by their first \( 2^k \) characters from last iteration

<table>
<thead>
<tr>
<th>( i )</th>
<th>( i )-th cyclic shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>racemice</td>
</tr>
<tr>
<td>6</td>
<td>racemice</td>
</tr>
<tr>
<td>2</td>
<td>racemice</td>
</tr>
<tr>
<td>3</td>
<td>racemice</td>
</tr>
<tr>
<td>7</td>
<td>racemice</td>
</tr>
<tr>
<td>5</td>
<td>racemice</td>
</tr>
<tr>
<td>4</td>
<td>racemice</td>
</tr>
<tr>
<td>0</td>
<td>racemice</td>
</tr>
</tbody>
</table>
Sort Cyclic Shifts

- equiv. classes of the parts do not exceed $n$
- how to do counting sort on pairs?
- use radix sort
  - sort by second half
  - then stable sort by first half (with counting sort)
- observation: substrings are already sorted by their first $2^k$ characters from last iteration
- but we want them to be sorted by second half
Sort Cyclic Shifts

- equiv. classes of the parts do not exceed $n$
- how to do counting sort on pairs?
- use radix sort
  - sort by second half
  - then stable sort by first half (with counting sort)

- observation: substrings are already sorted by their first $2^k$ characters from last iteration
- but we want them to be sorted by second half
- just shift the strings by $2^k$ to the left to obtain the sorted order by second half

<table>
<thead>
<tr>
<th>$i$</th>
<th>i-th cyclic shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>racemice</td>
</tr>
<tr>
<td>6</td>
<td>racemice</td>
</tr>
<tr>
<td>2</td>
<td>racemice</td>
</tr>
<tr>
<td>3</td>
<td>racemice</td>
</tr>
<tr>
<td>7</td>
<td>racemice</td>
</tr>
<tr>
<td>5</td>
<td>racemice</td>
</tr>
<tr>
<td>4</td>
<td>racemice</td>
</tr>
<tr>
<td>0</td>
<td>racemice</td>
</tr>
</tbody>
</table>
Sort Cyclic Shifts

- equiv. classes of the parts do not exceed $n$
- how to do counting sort on pairs?
- use radix sort
  - sort by second half
  - then stable sort by first half (with counting sort)

- observation: substrings are already sorted by their first $2^k$ characters from last iteration
- but we want them to be sorted by second half
- just shift the strings by $2^k$ to the left to obtain the sorted order by second half

<table>
<thead>
<tr>
<th>$i$</th>
<th>$i$-th cyclic shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>racemice</td>
</tr>
<tr>
<td>1</td>
<td>racemice</td>
</tr>
<tr>
<td>2</td>
<td>racemice</td>
</tr>
<tr>
<td>3</td>
<td>racemice</td>
</tr>
<tr>
<td>4</td>
<td>racemice</td>
</tr>
<tr>
<td>5</td>
<td>racemice</td>
</tr>
<tr>
<td>6</td>
<td>racemice</td>
</tr>
<tr>
<td>7</td>
<td>racemice</td>
</tr>
</tbody>
</table>
Sort Cyclic Shifts

sort length 1 substrings

initialize length 1 (= $2^0$) equivalence classes

for $k \in [0, \log n)$:

  sort by second half (via shift)

  stable counting sort by first half

  build classes for length $2^{k+1}$ strings
vector order(n,0);
iota(all(order),0);
sort(all(order), [&](int i, int j){
    return s[i]<s[j];
});

Sort Cyclic Shifts

sort length 1 substrings

initialize length 1 (= 2^0) equivalence classes

for k ∈ [0, log n):

    sort by second half (via shift)

    stable counting sort by first half

    build classes for length 2^{k+1} strings
Sort Cyclic Shifts

vector order(n,0);
iota(all(order),0);
sort(all(order), [&](int i, int j){
    return s[i]<s[j];
});

by_second = [(i-2**k)%n for i in order]

**Sort Cyclic Shifts**

- sort length 1 substrings
- initialize length 1 (= $2^0$) equivalence classes for $k \in [0, \log n)$:
  - sort by second half (via shift)
  - stable counting sort by first half
  - build classes for length $2^{k+1}$ strings
Sort Cyclic Shifts

vector order(n,0);
iota(all(order),0);
sort(all(order), [&](int i, int j){
    return s[i]<s[j];
});

by_second = [(i-2**k)%n for i in order]

vector cnt(n,0);
rep(i,n) cnt[eqclass[i]]++;
partial_sum(all(cnt),begin(cnt));
reverse(all(by_second));
for(auto i : by_second)
    order[--cnt[eqclass[i]]] = i;

Sort Cyclic Shifts
sort length 1 substrings
initialize length 1 (= 2^0) equivalence classes
for k ∈ [0, log n):
    sort by second half (via shift)
    stable counting sort by first half
    build classes for length 2^{k+1} strings

initialize length 1 (= 2^0) equivalence classes
for k ∈ [0, log n):
    sort by second half (via shift)
    stable counting sort by first half
    build classes for length 2^{k+1} strings
Sort Cyclic Shifts

vector order(n,0);
iota(all(order),0);
sort(all(order), [&](int i, int j){
    return s[i]<s[j];
});

by_second = [(i-2**k)%n for i in order]

vector cnt(n,0);
rep(i,n) cnt[eqclass[i]]++;
partial_sum(all(cnt),begin(cnt));
reverse(all(by_second));
for(auto i : by_second)
    order[--cnt[eqclass[i]]] = i;

Sort Cyclic Shifts

sort length 1 substrings

initialize length 1 (= 2^0) equivalence classes

for k ∈ [0, log n):
    sort by second half (via shift)
    stable counting sort by first half
    build classes for length 2^{k+1} strings

initial eq. classes must be in [0, n) now!
Sort Cyclic Shifts

```cpp
vector order(n, 0);
iota(all(order), 0);
sort(all(order), [&](int i, int j){
    return s[i] < s[j];
});
```

```cpp
vector cnt(n, 0);
rep(i, n) cnt[eqclass[i]]++;
partial_sum(all(cnt), begin(cnt));
reverse(all(by_second));
for(auto i : by_second)
    order[--cnt[eqclass[i]]] = i;
```

`by_second = [(i-2**k)%n for i in order]`

```cpp
initial eq. classes must be in [0, n) now!
```

**Sort Cyclic Shifts**

- sort length 1 substrings
- initialize length 1 (= 2^0) equivalence classes
- for `k ∈ [0, log n)`: sort by second half (via shift)
- stable counting sort by first half
- build classes for length 2^{k+1} strings

\[ → O(n \log n) \text{ suffix array} \]
Longest Common Prefix (LCP)

The lcp of two suffixes is equal to the size of the longest common prefix of those suffixes.
Longest Common Prefix (LCP)

The lcp of two suffixes is equal to the size of the longest common prefix of those suffixes.

\[
\begin{array}{c}
i \\
\downarrow \\
CGAAGTAAATAAGTAC
\end{array} \quad \begin{array}{c}
j \\
\downarrow \\
CGAAGTAAATAAGTAC
\end{array}
\]
Longest Common Prefix (LCP)

The lcp of two suffixes is equal to the size of the longest common prefix of those suffixes.

\[ \text{lcp}(i, j) = |\text{AAGTA}| = 5 \]
Longest Common Prefix (LCP)

The lcp of two suffixes is equal to the size of the longest common prefix of those suffixes.

\[ \text{lcp}(i, j) = |AAGTA| = 5 \]

How do we compute the lcp of two suffixes starting at indices \( i \) and \( j \)?

w/o hashing + binary search
Longest Common Prefix (LCP)

The lcp of two suffixes is equal to the size of the longest common prefix of those suffixes.

\[
lcp(i, j) = |\text{AAGTA}| = 5
\]

How do we compute the lcp of two suffixes starting at indices \(i\) and \(j\)?

- We can compare substrings of length \(2^k\) in \(O(1)\) using eq. classes from suffix sorting.

- Without hashing + binary search.
Longest Common Prefix (LCP)

The lcp of two suffixes is equal to the size of the longest common prefix of those suffixes.

\[
\text{lcp}(i, j) = |AAGTA| = 5
\]

- How do we compute the lcp of two suffixes starting at indices \(i\) and \(j\)?
- We can compare substrings of length \(2^k\) in \(O(1)\) using eq. classes from suffix sorting
- **Idea**: extend common prefix by powers of two from highest to lowest
Longest Common Prefix (LCP)

The lcp of two suffixes is equal to the size of the longest common prefix of those suffixes.

\[ \text{lcp}(i, j) = |\text{AAGTA}| = 5 \]

- **How do we compute the lcp of two suffixes starting at indices } \ i \text{ and } j?**
  - **Idea**: extend common prefix by powers of two from highest to lowest

- We can compare substrings of length } \ 2^k \text{ in } O(1) \text{ using eq. classes from suffix sorting}

- **LCP of Suffixes starting at } \ i, j \**
  
  ans = 0
  
  for } \ k \text{ from logn to 0:}
  
  if } \ s[i, i + 2^k) == s[j, j + 2^k): \text{:
      ans += } 2^k
      \ i += 2^k
      \ j += 2^k

w/o hashing + binary search
Longest Common Prefix (LCP)

The lcp of two suffixes is equal to the size of the longest common prefix of those suffixes.
Longest Common Prefix (LCP)

The lcp of two suffixes is equal to the size of the longest common prefix of those suffixes.

The lcp array stores at position $i$ the lcp of the $i$-th smallest and $(i + 1)$-th smallest suffix.
Longest Common Prefix (LCP)

The lcp of two suffixes is equal to the size of the longest common prefix of those suffixes.

The lcp array stores at position $i$ the lcp of the $i$-th smallest and $(i + 1)$-th smallest suffix.

<table>
<thead>
<tr>
<th>sa[$i$]</th>
<th>sa[$i$]-th suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>AAATAAGTAC</td>
</tr>
<tr>
<td>2</td>
<td>AAGTAAATAAGTAC</td>
</tr>
<tr>
<td>10</td>
<td>AAGTAC</td>
</tr>
<tr>
<td>7</td>
<td>AATAAGTAC</td>
</tr>
<tr>
<td>14</td>
<td>AC</td>
</tr>
<tr>
<td>3</td>
<td>AGTAAATAAGTAC</td>
</tr>
<tr>
<td>11</td>
<td>AGTAC</td>
</tr>
<tr>
<td>8</td>
<td>ATAAAGTAC</td>
</tr>
<tr>
<td>15</td>
<td>C</td>
</tr>
<tr>
<td>0</td>
<td>CGAAGTAAATAAGTAC</td>
</tr>
<tr>
<td>1</td>
<td>GAAGTAAATAAGTAC</td>
</tr>
<tr>
<td>4</td>
<td>GTAAATAAGTAC</td>
</tr>
<tr>
<td>12</td>
<td>GTAC</td>
</tr>
<tr>
<td>5</td>
<td>TAAATAAGTAC</td>
</tr>
<tr>
<td>9</td>
<td>TAAGTAC</td>
</tr>
<tr>
<td>13</td>
<td>TAC</td>
</tr>
</tbody>
</table>

$s = \text{CGAAGTAAATAAGTAC}$
The LCP array stores at position $i$ the LCP of the $i$-th smallest and $(i+1)$-th smallest suffix.

$s = $CGAAGTAAATAAGTAC
Longest Common Prefix (LCP)

The lcp of two suffixes is equal to the size of the longest common prefix of those suffixes. The lcp array stores at position $i$ the lcp of the $i$-th smallest and $(i + 1)$-th smallest suffix.

<table>
<thead>
<tr>
<th>sa[i]</th>
<th>sa[i]-th suffix</th>
<th>lcp-array[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>AAATAAGTAC</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>AAGTAAATAAGTAC</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>AAGTAC</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>ĀATAAGTAC</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>AC</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>ĀGTAATAAGTAC</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>ĀGTAC</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>ĀTAAGTAC</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>CGAAGTAAATAAGTAC</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>GAAGTAAATAAGTAC</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>GTAAATAAGTAC</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>GTAC</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>TAAATAAGTAC</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>TAAGTAC</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>TAC</td>
<td></td>
</tr>
</tbody>
</table>

$s = \text{CGAAGTAAATAAGTAC}$
Longest Common Prefix (LCP)

The lcp of two suffixes is equal to the size of the longest common prefix of those suffixes.

The lcp array stores at position \( i \) the lcp of the \( i \)-th smallest and \( (i + 1) \)-th smallest suffix.

\[
\begin{align*}
\text{sa[i]} & \quad \text{sa[i]-th suffix} & \quad \text{lcp-array[i]} \\
6 & \text{AAATAAGTAC} & 2 \\
2 & \text{AAAGTAAATAAGTAC} & 5 \\
10 & \text{AAGTAC} & 2 \\
7 & \text{ATAAGTAC} & 1 \\
14 & \text{AC} & 1 \\
3 & \text{AGTAAATAAGTAC} & 4 \\
11 & \text{AGTAC} & 1 \\
8 & \text{ATAAGTAC} & 0 \\
15 & \text{C} & 1 \\
0 & \text{CGAAGTAAATAAGTAC} & 0 \\
1 & \text{GAAGTAAATAAGTAC} & 1 \\
4 & \text{GTAATAAGTAC} & 3 \\
12 & \text{GTAC} & 0 \\
5 & \text{TAAATAAGTAC} & 3 \\
9 & \text{TAAGTAC} & 2 \\
13 & \text{TAC} & \\
\end{align*}
\]

\( s = \text{CGAAGTAAATAAGTAC} \)
Longest Common Prefix (LCP)

The lcp of two suffixes is equal to the size of the longest common prefix of those suffixes. The lcp array stores at position $i$ the lcp of the $i$-th smallest and $(i + 1)$-th smallest suffix.

<table>
<thead>
<tr>
<th>$sa[i]$</th>
<th>$sa[i]$-th suffix</th>
<th>lcp-array[$i$] = lcp($sa[i], sa[i + 1]$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>AAATAAGTAC</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>AAGTAAATAAGTAC</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>AAGTAC</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>ATAAGTAC</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>AC</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>ATTAAGTAC</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>AGTAC</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>ATAAAGTAC</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>CGAAGTAAATAAGTAC</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>GAAGTAAATAAGTAC</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>GTAAATAAGTAC</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>GTAC</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>TAAATAAGTAC</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>TAAGTAC</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>TAC</td>
<td></td>
</tr>
</tbody>
</table>

$s = \text{CGAAGTAAATAAGTAC}$

The lcp of two suffixes is equal to the size of the longest common prefix of those suffixes.

Value of $lcp(2, 11)$ is 1.
Longest Common Prefix (LCP)

The lcp of two suffixes is equal to the size of the longest common prefix of those suffixes.

The lcp array stores at position $i$ the lcp of the $i$-th smallest and $(i + 1)$-th smallest suffix.

$s = \text{CGAAGTAATAAGTAC}$

<table>
<thead>
<tr>
<th>$sa[i]$</th>
<th>$sa[i]$-th suffix</th>
<th>lcp-array[$i$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>AAATAAGTAC</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>AAGTAAATAAGTAC</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>AAGTAC</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>ATAAAGTAC</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>AC</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>AATTAATAAGTAC</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>ATAAAGTAC</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>ATAAAGTAC</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>CGAAGTAATAAGTAC</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>GAAGTAATAAGTAC</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>GTAAATAAGTAC</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>GTAC</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>TAAATAAGTAC</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>TAAGTAC</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>TAC</td>
<td></td>
</tr>
</tbody>
</table>

$lcp(2, 11) = 1$
Longest Common Prefix (LCP)

The \( lcp \) of two suffixes is equal to the size of the longest common prefix of those suffixes.

The \( lcp \) array stores at position \( i \) the \( lcp \) of the \( i \)-th smallest and \( (i + 1) \)-th smallest suffix.

\[
\begin{array}{cc|c|cc|c}
\text{sa}[i] & \text{sa}[i]-\text{th suffix} & \text{lcp-array}[i] & \text{s} = \text{CGAAGTAAATAAGTAC} \\
6 & \text{AAATAAGTAC} & 2 & \text{lcp}(2, 11) = 1 \\
2 & \text{AAATTAATAAGTAC} & 5 & \\
10 & \text{AAGTAC} & 2 & \\
7 & \text{AAATAAGTAC} & 1 & \\
14 & \text{AC} & 1 & \\
3 & \text{AGTAAATAAGTAC} & 4 & \\
11 & \text{AGTAC} & 1 & \\
8 & \text{ATAAGTAC} & 0 & \\
15 & \text{C} & 1 & \\
0 & \text{CGAAGTAAATAAGTAC} & 0 & \\
1 & \text{GAAGTAAATAAGTAC} & 1 & \\
4 & \text{GTAATAAGTAC} & 3 & \\
12 & \text{GTAC} & 0 & \\
5 & \text{TAAATAAGTAC} & 3 & \\
9 & \text{TAAGTAC} & 2 & \\
13 & \text{TAC} & 1 & \\
\end{array}
\]

Observation: \( lcp(\text{sa}[i], \text{sa}[j]) = \min(\text{lcp-array}[i, j]) \)
Longest Common Prefix (LCP)

The LCP of two suffixes is equal to the size of the longest common prefix of those suffixes.

The LCP array stores at position \(i\) the LCP of the \(i\)-th smallest and \((i + 1)\)-th smallest suffix.

\[
\begin{array}{|c|c||c|}
\hline
sa[i] & sa[i]-th suffix & lcp-array[i] \\
\hline
6 & AAATAAGTAC & 2 \\
2 & AAGTAATAAGTAC & 5 \\
10 & AAGTAC & 2 \\
7 & ÁATAAGTAC & 1 \\
14 & AC & 1 \\
3 & ÁGTAATAAGTAC & 4 \\
11 & AGTAC & 1 \\
8 & ÁTAAGTAC & 0 \\
15 & C & 1 \\
0 & CGAAGTAAATAAGTAC & 0 \\
1 & GAAGTAAATAAGTAC & 1 \\
4 & GTAAATAAGTAC & 3 \\
12 & GTAC & 0 \\
5 & TAAATAAGTAC & 3 \\
9 & TAAGTAC & 2 \\
13 & TAC & \\
\hline
\end{array}
\]

\[
lcp(sa[i], sa[i + 1]) = \min(lcp-array[i, j])
\]

Observation: \(\min(lcp-array[i, j])\)

\[
lcp(2, 11) = 1
\]

\[
\rightarrow \text{LCP can be reduced to range minimum queries}
\]
Suffix Array and LCP Array Overview

Suffix Array
### Suffix Array and LCP Array Overview

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suffix Array</td>
<td>$O(n^2 \log n)$</td>
</tr>
<tr>
<td>Naive</td>
<td></td>
</tr>
</tbody>
</table>
Suffix Array and LCP Array Overview

| Suffix Array | Naive          | $O(n^2 \log n)$ |
|             | Hashing        | $O(n \log^2 n)$ |
# Suffix Array and LCP Array Overview

<table>
<thead>
<tr>
<th>Suffix Array</th>
<th>Naive</th>
<th>$O(n^2 \log n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hashing</td>
<td>$O(n \log^2 n)$</td>
</tr>
<tr>
<td></td>
<td>Cyclic Shift DP + std::sort</td>
<td>$O(n \log^2 n)$</td>
</tr>
</tbody>
</table>
## Suffix Array and LCP Array Overview

<table>
<thead>
<tr>
<th>Suffix Array</th>
<th>Naive</th>
<th>$O(n^2 \log n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hashing</td>
<td>$O(n \log^2 n)$</td>
<td></td>
</tr>
<tr>
<td>Cyclic Shift DP + <code>std::sort</code></td>
<td>$O(n \log^2 n)$</td>
<td></td>
</tr>
<tr>
<td>Cyclic Shift DP + Counting Sort</td>
<td>$O(n \log n)$</td>
<td></td>
</tr>
</tbody>
</table>
## Suffix Array and LCP Array Overview

<table>
<thead>
<tr>
<th>Suffix Array Method</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>$O(n^2 \log n)$</td>
</tr>
<tr>
<td>Hashing</td>
<td>$O(n \log^2 n)$</td>
</tr>
<tr>
<td>Cyclic Shift DP + std::sort</td>
<td>$O(n \log^2 n)$</td>
</tr>
<tr>
<td>Cyclic Shift DP + Counting Sort</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Best Known</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
### Suffix Array and LCP Array Overview

<table>
<thead>
<tr>
<th>Suffix Array</th>
<th>Naive</th>
<th>$O(n^2 \log n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hashing</td>
<td>$O(n \log^2 n)$</td>
</tr>
<tr>
<td></td>
<td>Cyclic Shift DP + std::sort</td>
<td>$O(n \log^2 n)$</td>
</tr>
<tr>
<td></td>
<td>Cyclic Shift DP + Counting Sort</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td></td>
<td>Best Known</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LCP Array</th>
<th>Kasai’s algorithm (from just SA)</th>
<th>$O(n)$</th>
</tr>
</thead>
</table>
### Suffix Array and LCP Array Overview

<table>
<thead>
<tr>
<th>Suffix Array</th>
<th>Naive</th>
<th>$O(n^2 \log n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hashing</td>
<td></td>
<td>$O(n \log^2 n)$</td>
</tr>
<tr>
<td>Cyclic Shift DP + std::sort</td>
<td></td>
<td>$O(n \log^2 n)$</td>
</tr>
<tr>
<td>Cyclic Shift DP + Counting Sort</td>
<td></td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Best Known</td>
<td></td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LCP Array</th>
<th>Kasai’s algorithm (from just SA)</th>
<th>$O(n)$</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>LCP</th>
<th>Fit 2s powers with SA classes</th>
<th>$O(\log n)$ query; $O(n \log n)$ construct</th>
</tr>
</thead>
</table>
## Suffix Array and LCP Array Overview

<table>
<thead>
<tr>
<th>Suffix Array</th>
<th>Naive</th>
<th>$O(n^2 \log n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hashing</td>
<td></td>
<td>$O(n \log^2 n)$</td>
</tr>
<tr>
<td>Cyclic Shift DP + std::sort</td>
<td></td>
<td>$O(n \log^2 n)$</td>
</tr>
<tr>
<td>Cyclic Shift DP + Counting Sort</td>
<td></td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Best Known</td>
<td></td>
<td>$O(n)$</td>
</tr>
<tr>
<td>LCP Array</td>
<td>Kasai’s algorithm (from just SA)</td>
<td>$O(n)$</td>
</tr>
<tr>
<td></td>
<td>$n - 1$ LCP queries wit SA-classes</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>LCP</td>
<td>Fit 2s powers with SA classes</td>
<td>$O(\log n)$ query; $O(n \log n)$ construct</td>
</tr>
</tbody>
</table>
Suffix Array and LCP Array Overview

### Suffix Array
- **Naive**
- **Hashing**
- **Cyclic Shift DP + std::sort**
- **Cyclic Shift DP + Counting Sort**
- **Best Known**

### LCP Array
- **Kasai’s algorithm (from just SA)**
- **n – 1 LCP queries wit SA-classes**

### LCP
- **Fit 2s powers with SA classes**
- **Range-Min (RMQ) on LCP-Array**

**Complexity**
- **Suffix Array**
  - Naive: $O(n^2 \log n)$
  - Hashing: $O(n \log^2 n)$
  - Cyclic Shift DP + std::sort: $O(n \log^2 n)$
  - Cyclic Shift DP + Counting Sort: $O(n \log n)$
  - Best Known: $O(n)$

- **LCP Array**
  - Kasai’s algorithm: $O(n)$
  - $n – 1$ LCP queries: $O(n \log n)$

- **LCP**
  - Fit 2s powers: $O(\log n)$ query; $O(n \log n)$ construct
  - Range-Min (RMQ) on LCP-Array: $O(1)$ query; $O(n)$ construct
## Suffix Array and LCP Array Overview

| Suffix Array | Naive          | $O(n^2 \log n)$ |
|             | Hashing        | $O(n \log^2 n)$ |
|             | Cyclic Shift DP + std::sort | $O(n \log^2 n)$ |
|             | Cyclic Shift DP + Counting Sort | $O(n \log n)$ |
|             | Best Known     | $O(n)$ |

| LCP Array   | Kasai’s algorithm (from just SA) | $O(n)$ |
|             | $n - 1$ LCP queries wit SA-classes | $O(n \log n)$ |

| LCP         | Fit $2^s$ powers with SA classes | $O(\log n)$ query; $O(n \log n)$ construct |
|             | Range-Min (RMQ) on LCP-Array     | $O(1)$ query; $O(n)$ construct |

| RMQ         | Best Known                     | $O(1)$ query; $O(n)$ construct |
## Suffix Array and LCP Array Overview

<table>
<thead>
<tr>
<th>Suffix Array</th>
<th>Naive</th>
<th>$O(n^2 \log n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hashing</td>
<td>$O(n \log^2 n)$</td>
</tr>
<tr>
<td></td>
<td>Cyclic Shift DP + std::sort</td>
<td>$O(n \log^2 n)$</td>
</tr>
<tr>
<td></td>
<td>Cyclic Shift DP + Counting Sort</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td></td>
<td>Best Known</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LCP Array</th>
<th>Kasai’s algorithm (from just SA)</th>
<th>$O(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$n-1$ LCP queries with SA-classes</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LCP</th>
<th>Fit 2s powers with SA classes</th>
<th>$O(n \log n)$ query; $O(n \log n)$ construct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Range-Min (RMQ) on LCP-Array</td>
<td>$O(1)$ query; $O(n)$ construct</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RMQ</th>
<th>Best Known</th>
<th>$O(1)$ query; $O(n \log n)$ construct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sparse Table</td>
<td>$O(1)$ query; $O(n)$ construct</td>
</tr>
</tbody>
</table>
## Suffix Array and LCP Array Overview

<table>
<thead>
<tr>
<th>Suffix Array</th>
<th>Naive</th>
<th>$O(n^2 \log n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hashing</td>
<td>$O(n \log^2 n)$</td>
<td></td>
</tr>
<tr>
<td>Cyclic Shift DP + std::sort</td>
<td>$O(n \log^2 n)$</td>
<td></td>
</tr>
<tr>
<td>Cyclic Shift DP + Counting Sort</td>
<td>$O(n \log n)$</td>
<td></td>
</tr>
<tr>
<td><strong>Best Known</strong></td>
<td></td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LCP Array</th>
<th>Kasai’s algorithm (from just SA)</th>
<th>$O(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$n - 1$ LCP queries with SA-classes</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LCP</th>
<th>Fit 2s powers with SA classes</th>
<th>$O(\log n)$ query; $O(n \log n)$ construct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Range-Min (RMQ) on LCP-Array</td>
<td>$O(1)$ query; $O(n)$ construct</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RMQ</th>
<th>Best Known</th>
<th>$O(1)$ query; $O(n)$ construct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sparse Table</td>
<td>$O(1)$ query; $O(n \log n)$ construct</td>
</tr>
</tbody>
</table>

*relevant for us
15min Break

**Longest Common Substring:**
Given strings $s, t$, find longest string that appears in both.

$|s| = n$
$|t| = m$

**Distinct Substring Count:**
Given string $s$, find the number of different substrings of $s$.

**Minimum Rotation:**
Find lexicographically smallest cyclic shift of string $s$. 
15min Break

Longest Common Substring:
Given strings $s,t$, find longest string that appears in both.

dp: $O(nm)$

|s| = n
|t| = m

Distinct Substring Count:
Given string $s$, find the number of different substrings of $s$.

naive: $O(n^3 \log n)$

Minimum Rotation:
Find lexicographically smallest cyclic shift of string $s$.

naive: $O(n^2)$
15min Break

**Longest Common Substring:**
Given strings $s, t$, find longest string that appears in both.

- **dp:** $O(nm)$
- **hashing:** $O((n + m) \log^2(n + m))$

| $|s| = n$ | $|t| = m$ |

- **naive:** $O(n^3 \log n)$
- **hashing:** $O(n^2 \log n)$

**Distinct Substring Count:**
Given string $s$, find the number of different substrings of $s$.

- **naive:** $O(n^2)$
- **hashing:** $O(n \log n)$

**Minimum Rotation:**
Find lexicographically smallest cyclic shift of string $s$.

- **naive:** $O(n^2)$
- **hashing:** $O(n \log n)$
15min Break

**Longest Common Substring:**
Given strings $s, t$, find longest string that appears in both.

- **dp:** $O(nm)$
- **hashing:** $O((n + m) \log^2(n + m))$
- **suffix array:** $O((n + m) \log(n + m))$ *

- **naive:** $O(n^3 \log n)$
- **hashing:** $O(n^2 \log n)$
- **suffix array:** $O(n \log n)$ *

**Distinct Substring Count:**
Given string $s$, find the number of different substrings of $s$.

**Minimum Rotation:**
Find lexicographically smallest cyclic shift of string $s$.

- **naive:** $O(n^2)$
- **hashing:** $O(n \log n)$
- **suffix array:** $O(n \log n)$ *

* assumes $O(n \log n)$ suffix array
15min Break

**Longest Common Substring:**
Given strings $s, t$, find longest string that appears in both.

- **dp:** $O(nm)$
- **hashing:** $O((n + m)\log^2(n + m))$
- **suffix array:** $O((n + m)\log(n + m))$

<table>
<thead>
<tr>
<th>$s$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$m$</td>
</tr>
</tbody>
</table>

- **naive:** $O(n^3 \log n)$
- **hashing:** $O(n^2 \log n)$
- **suffix array:** $O(n \log n)$

**Distinct Substring Count:**
Given string $s$, find the number of different substrings of $s$.

- $\#\text{substrings} - \sum(\text{lcp-array})$

(https://cp-algorithms.com/string/suffix-array.html)

**Minimum Rotation:**
Find lexicographically smallest cyclic shift of string $s$.

- **naive:** $O(n^2)$
- **hashing:** $O(n \log n)$
- **suffix array:** $O(n \log n)$

* assumes $O(n \log n)$ suffix array

---

Competitive Programming | Algorithm Engineering Group (HPI) & Scalable Algorithms Group (KIT)
Aho-Corasick: Motivation

Find Pattern in Text:
Find all occurrences of patterns $t_1, t_2 \ldots$ in string $s$. 

$|s| = n$

$|t_i| = m_i$
Aho-Corasick: Motivation

**Find Pattern in Text:**
Find all occurrences of patterns \( t_1, t_2 \ldots \) in string \( s \).

naive: \( O(nm_i) \) per pattern

\[ |s| = n \]
\[ |t_i| = m_i \]
Aho-Corasick: Motivation

Find Pattern in Text:
Find all occurrences of patterns $t_1, t_2 \ldots$ in string $s$.

naive: $O(nm_i)$ per pattern

dp: $O(n + m_i)$ per pattern
Aho-Corasick: Motivation

|s| = n
|t| = m_i

**Find Pattern in Text:**
Find all occurrences of patterns $t_1, t_2 \ldots$ in string $s$.

naive: $O(n m_i)$ per pattern

dp: $O(n + m_i)$ per pattern

hashing: $O(n + m_i)$ per pattern
Aho-Corasick: Motivation

Find Pattern in Text:
Find all occurrences of patterns $t_1, t_2 \ldots$ in string $s$.

naive: $O(nm_i)$ per pattern

dp: $O(n + m_i)$ per pattern

hashing: $O(n + m_i)$ per pattern

suffix array: $O(n \log n + \sum m_i \log n)$
Aho-Corasick: Motivation

Find Pattern in Text:
Find all occurrences of patterns $t_1, t_2 \ldots$ in string $s$.

naive: $O(nm_i)$ per pattern

dp: $O(n + m_i)$ per pattern

hashing: $O(n + m_i)$ per pattern

suffix array: $O(n \log n + \sum m_i \log n)$

Aho-Corasick: $O(n + \sum m_i)$
Aho-Corasick: Motivation

|s| = n
|t_i| = m_i

Find Pattern in Text:
Find all occurrences of patterns t_1, t_2 ... in string s.

naive: O(nm_i) per pattern

dp: O(n + m_i) per pattern

hashing: O(n + m_i) per pattern

suffix array: O(n \log n + \sum m_i \log n)

Aho-Corasick: O(n + \sum m_i)

finite state machine (automaton) on the trie of t_1, t_2 ...
A trie stores a set of strings. It is a rooted tree with the following features:

Trie for $ab$
A **trie** stores a set of strings. It is a rooted tree with the following features:

- there is a root vertex

Trie for `ab`
A trie stores a set of strings. It is a rooted tree with the following features:

- there is a root vertex
- every edge is labeled by some character

Trie for $ab$
A **trie** stores a set of strings. It is a rooted tree with the following features:

- there is a root vertex
- every edge is labeled by some character
- path from root to a vertex corresponds to prefix of a stored string

Trie for \textit{ab}
Aho-Corasick: Tries

A **trie** stores a set of strings. It is a rooted tree with the following features:

- there is a root vertex
- every edge is labeled by some character
- path from root to a vertex corresponds to prefix of a stored string
- a vertex is marked if its corresponding string is one of the stored strings

![Trie for ab](image.png)
Aho-Corasick: Tries

A **trie** stores a set of strings. It is a rooted tree with the following features:

- there is a root vertex
- every edge is labeled by some character
- path from root to a vertex corresponds to prefix of a stored string
- a vertex is marked if its corresponding string is one of the stored strings

![Trie for ab](image1.png)

Trie for ab

![Trie for ab and aa](image2.png)

Trie for ab and aa
Aho-Corasick: Tries

A **trie** stores a set of strings. It is a rooted tree with the following features:

- there is a root vertex
- every edge is labeled by some character
- path from root to a vertex corresponds to prefix of a stored string
- a vertex is marked if its corresponding string is one of the stored strings

Construction takes $O(nK)$ time, where $n = \sum m_i$ and $K$ alphabet size.

[i.e., linear in input for constant alphabet]
Aho-Corasick: Trie Construction

Trie Construction

```c++
struct AhoCorasick {
    const static int A = 3; // alphabet size
    struct Vert {
        array<int, A> next = {};  // custom data
        bool marked = false;
    };
    vector<Vert> trie(1); // init with root
    void add_string(const string& s);
    void finalize(); // build automaton from trie
};
```
Aho-Corasick: Trie Construction

Trie Construction

- start at root

```cpp
struct AhoCorasick {
    const static int A = 3; // alphabet size
    struct Vert {
        array<int, A> next = {};
        bool marked = false; // custom data
    };
    vector<Vert> trie(1); // init with root
    void add_string(const string& s);
    void finalize(); // build automaton from trie
};
```
Aho-Corasick: Trie Construction

Trie Construction

- start at root
- move along edges corresponding to characters of \( s \)

```c++
struct AhoCorasick {
    const static int A = 3; // alphabet size
    struct Vert {
        array<int, A> next = {};
        bool marked = false; // custom data
    };
    vector<Vert> trie(1); // init with root
    void add_string(const string& s);
    void finalize(); // build automaton from trie
};
```
Aho-Corasick: Trie Construction

Trie Construction

- start at root
- move along edges corresponding to characters of $s$
- if no corresponding edge for a character exists, create new vertex and missing edge

```cpp
struct AhoCorasick {
    const static int A = 3; // alphabet size
    struct Vert {
        array<int,A> next = {};
        bool marked = false; // custom data
    };
    vector<Vert> trie(1); // init with root
    void add_string(const string& s);
    void finalize(); // build automaton from trie
};
```
Aho-Corasick: Trie Construction

Trie Construction

- start at root
- move along edges corresponding to characters of \( s \)
- if no corresponding edge for a character exists, create new vertex and missing edge
- mark the vertex you end up at

```cpp
struct AhoCorasick {
    const static int A = 3; // alphabet size
    struct Vert {
        array<int, A> next = {};
        bool marked = false; // custom data
    };
    vector<Vert> trie(1); // init with root
    void add_string(const string& s);
    void finalize(); // build automaton from trie
};
```
Aho-Corasick: Trie Construction

Trie Construction

- start at root
- move along edges corresponding to characters of s
- if no corresponding edge for a character exists, create new vertex and missing edge
- mark the vertex you end up at

```cpp
struct AhoCorasick {
    const static int A = 3; // alphabet size
    struct Vert {
        array<int,A> next = {};  // custom data
        bool marked = false;
    }
    vector<Vert> trie(1);  // init with root
    void add_string(const string& s) {
        int v = 0;
        for(char c : s) {
            c -= 'a';
            if(!trie[v].next[c]) {
                trie[v].next[c] = trie.size();
                trie.emplace_back();
            }
            v = trie[v].next[c];
        }
        trie[v].marked = true;
    }
    void finalize();  // build automaton from trie
};
```
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.

- each vertex in the trie becomes a state
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.

- each vertex in the trie becomes a state
- while feeding chars into the automaton we are always in the longest matching state
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.

- each vertex in the trie becomes a state
- while feeding chars into the automaton we are always in the longest *matching* state

underlying trie vertex of a state $\iff$ longest prefix of some $s$ ending at the current input
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.

- each vertex in the trie becomes a state
- while feeding chars into the automaton we are always in the longest matching state

underlying trie vertex of a state ⇔ longest prefix of some $s$ ending at the current input
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.

- each vertex in the trie becomes a state
- while feeding chars into the automaton we are always in the longest *matching* state

underlying trie vertex of a state $\Leftrightarrow$ longest prefix of some $s$ ending at the current input
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.

- each vertex in the trie becomes a state
- while feeding chars into the automaton we are always in the longest matching state

underlying trie vertex of a state $\iff$ longest prefix of some $s$ ending at the current input

Strings

$abc, ab, bcd$

Text

$a$

```
c
  “ab”
    b
      a
        root

“abc”
    “bcd”
  c
```

Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.

- each vertex in the trie becomes a state
- while feeding chars into the automaton we are always in the longest matching state

underlying trie vertex of a state $\iff$ longest prefix of some $s$ ending at the current input

Strings

$abc$, $ab$, $bcd$

Text

a a

```
Strings
abc, ab, bcd

Text
a a

```

```
root

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>&quot;ab&quot;</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>
| "abc" | "bcd"
```
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.

- each vertex in the trie becomes a state
- while feeding chars into the automaton we are always in the longest matching state

underlying trie vertex of a state $\iff$ longest prefix of some $s$ ending at the current input

---

**Strings**

$abc$, $ab$, $bcd$

**Text**

a a b

---

Diagram:

- **Strings**: $abc$, $ab$, $bcd$
- **Text**: a a b

Diagram shows:

- Root
- Vertex a
- Vertex b
- Vertex c
- Vertex d
- Vertex "ab"
- Vertex "abc"
- Vertex "bcd"
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.

- each vertex in the trie becomes a state
- while feeding chars into the automaton we are always in the longest matching state

underlying trie vertex of a state $\iff$ longest prefix of some $s$ ending at the current input

Strings

$abc, \ ab, \ bcd$

Text

a a b c
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.

- each vertex in the trie becomes a state
- while feeding chars into the automaton we are always in the longest *matching state*

underlying trie vertex of a state ⇔ longest prefix of some $s$ ending at the current input

Strings

\[
abc, \ ab, \ bcd
\]

Text

\[
a \ a \ b \ c \ d
\]
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.

- each vertex in the trie becomes a state
- while feeding chars into the automaton we are always in the longest matching state

underlying trie vertex of a state $\iff$ longest prefix of some $s$ ending at the current input
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.

- needs a transition for every possible character at each state

```
root
  \_ a
  \  \_ b
   \_ \_ c
   \  \_ d
    \_ c
```
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.

- needs a transition for every possible character at each state
- no direct child for character $c$ at vertex? Try from vertex of longest suffix, i.e., $\epsilon$-transitions via so called **suffix-links**
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.

- needs a transition for every possible character at each state
- no direct child for character c at vertex? Try from vertex of longest suffix, i.e., \( \varepsilon \)-transitions via so called suffix-links

  e.g. for vertex \( abc \) the longest suffix is \( bc \)
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.

- needs a transition for every possible character at each state
- no direct child for character $c$ at vertex? Try from vertex of longest suffix, i.e., $\varepsilon$-transitions via so called **suffix-links**
  
  e.g. for vertex $abc$ the longest suffix is $bc$
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.

- needs a transition for every possible character at each state
- no direct child for character $c$ at vertex? Try from vertex of longest suffix, i.e., $\epsilon$-transitions via so called suffix-links
  
  e.g. for vertex $abc$ the longest suffix is $bc$
- when computing suffix links & transitions in BFS-order, then suffix link already has all transitions
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.

- needs a transition for every possible character at each state
- no direct child for character c at vertex? Try from vertex of longest suffix, i.e., $\epsilon$-transitions via so called suffix-links
  
  e.g. for vertex abc the longest suffix is bc

- when computing suffix links & transitions in BFS-order, then suffix link already has all transitions
- with $pch =$ character that leads parent vertex to us; suffix link can be computed as follows
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.

- needs a transition for every possible character at each state
- no direct child for character \( c \) at vertex? Try from vertex of longest suffix, i.e., \( \varepsilon \)-transitions via so called **suffix-links**
  
  e.g. for vertex \( abc \) the longest suffix is \( bc \)

- when computing suffix links & transitions in BFS-order, then suffix link already has all transitions
- with \( pch \) = character that leads parent vertex to us; suffix link can be computed as follows

  - go to parent
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.

- needs a transition for every possible character at each state
- no direct child for character $c$ at vertex? Try from vertex of longest suffix, i.e., $\varepsilon$-transitions via so called **suffix-links**

  e.g. for vertex $abc$ the longest suffix is $bc$

- when computing suffix links & transitions in BFS-order, then suffix link already has all transitions
- with $pch =$ character that leads parent vertex to us; suffix link can be computed as follows

  - go to parent
  - use suffix-link from there
Aho-Corasick: Automaton Construction

Build an automaton based on the trie representing the set of strings.

- needs a transition for every possible character at each state
- no direct child for character $c$ at vertex? Try from vertex of longest suffix, i.e., $\varepsilon$-transitions via so-called **suffix-links**

  e.g. for vertex $abc$ the longest suffix is $bc$

- when computing suffix links & transitions in BFS-order, then suffix link already has all transitions
- with $pch =$ character that leads parent vertex to us; suffix link can be computed as follows
  
  - go to parent
  - use suffix-link from there
  - go the transition for $pch$
Aho-Corasick: Automaton Construction

Build an automaton based on the trie

```cpp
struct AhoCorasick {
    const static int A = 3; // alphabet size
    struct Vert {
        array<int,A> next = {}; // custom data
        bool marked = false;
    }
    vector<Vert> trie(1); // init with root
    void add_string(const string& s);
    void finalize(); // build automaton from trie
};
```
Aho-Corasick: Automaton Construction

Build an automaton based on the trie

- do a BFS and for each vertex

```cpp
struct AhoCorasick {
    const static int A = 3; // alphabet size
    struct Vert {
        array<int, A> next = {};
        bool marked = false; // custom data
    };
    vector<Vert> trie(1); // init with root
    void add_string(const string& s);
    void finalize(); // build automaton from trie
};
```
Aho-Corasick: Automaton Construction

Build an automaton based on the trie

- do a BFS and for each vertex
- compute the suffix link as \( \text{trie}[\text{link}(\text{parent})].\text{next}[\text{pch}] \)

```cpp
struct AhoCorasick {
    const static int A = 3; // alphabet size
    struct Vert {
        array<int, A> next = {};
        bool marked = false; // custom data
    }
    vector<Vert> trie(1); // init with root
    void add_string(const string& s);
    void finalize(); // build automaton from trie
};
```
Aho-Corasick: Automaton Construction

Build an automaton based on the trie

- do a BFS and for each vertex
- compute the suffix link as $\text{trie}[\text{link(parent)}].\text{next}[\text{pch}]$
- if there is no child for char $c$, copy transition from suffix link

```cpp
struct AhoCorasick {
    const static int A = 3; // alphabet size
    struct Vert {
        array<int, A> next = {};
        bool marked = false; // custom data
    };
    vector<Vert> trie(1); // init with root
    void add_string(const string& s);
    void finalize(); // build automaton from trie
};
```
Aho-Corasick: Automaton Construction

Build an automaton based on the trie

- do a BFS and for each vertex
- compute the suffix link as `trie[link(parent)].next[pch]`
- if there is no child for char `c`, copy transition from suffix link

```cpp
struct AhoCorasick {
  const static int A = 3; // alphabet size
  struct Vert {
    array<int, A> next = {};  // custom data
    bool marked = false;
  };
  vector<Vert> trie(1); // init with root
  void add_string(const string& s);
  void finalize() {
    queue<array<int, 4>> q{{0, 0, 0, 0}};
    while (size(q)) {
      auto [v, p, plink, pch] = q.front(); q.pop();
      int link = p == 0 ? 0 : trie[plink].next[pch];
      rep(c, A) {
        if (!trie[v].next[c]) // no child for c
          trie[v].next[c] = trie[link].next[c];
        else
          q.push({trie[v].next[c], v, link, c});
      } // after this loop: compute v's aggregates
    } // build automaton
  }
};
```
Aho-Corasick: Automaton Construction

Build an automaton based on the trie

- do a BFS and for each vertex
- compute the suffix link as $\text{trie[link(parent)].next[pch]}$.
- if there is no child for char $c$, copy transition from suffix link.

```cpp
struct AhoCorasick {
    const static int A = 3; // alphabet size
    struct Vert {
        array<int, A> next = {};
        bool marked = false; // custom data
    };
    vector<Vert> trie(1); // init with root
    void add_string(const string& s);
    void finalize() {
        queue<array<int,4>> q{{{0,0,0,0}}};
        while(size(q)) {
            auto [v, p, plink, pch] = q.front(); q.pop();
            int link = p == 0 ? 0 : trie[plink].next[pch];
            rep(c,A) {
                if(!trie[v].next[c]) // no child for c
                    trie[v].next[c] = trie[link].next[c];
                else
                    q.push({trie[v].next[c], v, link, c});
            }
            // after this loop: compute v's aggregates
        }
    }
};
```
Aho-Corasick: Automaton Construction

Build an automaton based on the trie

- do a BFS and for each vertex
- compute the suffix link as $\text{trie}[\text{link}(\text{parent})].\text{next}[\text{pch}]$
- if there is no child for char $c$, copy transition from suffix link

Why the $p == 0$ check?

```
struct AhoCorasick {
    const static int A = 3; // alphabet size
    struct Vert {
        array<int,A> next = {}; // custom data
        bool marked = false;
    };
    vector<Vert> trie(1); // init with root
    void add_string(const string& s);
    void finalize() {
        queue<array<int,4>> q{{{0,0,0,0}}};
        while(size(q)) {
            auto [v, p, plink, pch] = q.front(); q.pop();
            int link = p == 0 ? 0 : trie[plink].next[pch];
            rep(c,A) {
                if(!trie[v].next[c]) // no child for c
                    trie[v].next[c] = trie[link].next[pch];
                else
                    q.push({trie[v].next[c], v, link, c});
            }
        } // after this loop: compute v's aggregates
    }
}; // build automaton
```
Aho-Corasick: Automaton Construction

Build an automaton based on the trie

- do a BFS and for each vertex
- compute the suffix link as $\text{trie}[\text{link}(\text{parent})].\text{next}[\text{pch}]$
- if there is no child for char $c$, copy transition from suffix link

Why the $p == 0$ check?

Is this enough for find pattern in text?

```cpp
struct AhoCorasick {
    const static int A = 3; // alphabet size
    struct Vert {
        array<int,A> next = {};
        bool marked = false; // custom data
    };
    vector<Vert> trie(1); // init with root
    void add_string(const string& s);
    void finalize() {
        queue<array<int,4>> q{{{0,0,0,0}}};
        while(size(q)) {
            auto [v, p, plink, pch] = q.front(); q.pop();
            int link = p == 0 ? 0 : trie[plink].next[pch];
            rep(c,A) {
                if(!trie[v].next[c]) // no child for c
                    trie[v].next[c] = trie[link].next[c];
                else
                    q.push({trie[v].next[c], v, link, c});
            // after this loop: compute v's aggregates
        }
    // build automaton
    }
};
```

Why the $p == 0$ check? Is this enough for find pattern in text?
Aho-Corasick: Automaton Construction

Build an automaton based on the trie:

- do a BFS and for each vertex
- compute the suffix link as
  $trie[link(parent)].next[pch]$ if there is no child for char c, copy transition from suffix link
- if there is no child for char c, copy transition from suffix link

Why the $p == 0$ check?

Is this enough for finding patterns in text?

```
struct AhoCorasick {
    const static int A = 3; // alphabet size
    struct Vert {
        array<int,A> next = {};
        bool marked = false; // custom data
    };
    vector<Vert> trie(1); // init with root
    void add_string(const string& s);
    void finalize() {
        queue<array<int,4>> q{{{0,0,0,0}}};
        while(size(q)) {
            auto [v, p, plink, pch] = q.front(); q.pop();
            int link = p == 0 ? 0 : trie[plink].next[pch];
            rep(c,A) {
                if(!trie[v].next[c]) // no child for c
                    trie[v].next[c] = trie[link].next[c];
                else
                    q.push({trie[v].next[c], v, link, c});
            } // after this loop: compute v's aggregates
        } // build automaton
    }
};
```
Aho-Corasick: Automaton Construction

Build an automaton based on the trie:

■ do a BFS and for each vertex
■ compute the suffix link as $trie[link(parent)].next[pch]$;
■ if there is no child for char $c$, copy transition from suffix link.

Why the $p == 0$ check?

Is this enough for find pattern in text?

for our example (find pattern in text) we must aggregate the marked flag along suffix links.

```cpp
struct AhoCorasick {
    const static int A = 3; // alphabet size
    struct Vert {
        array<int, A> next = {};
        bool marked = false; // custom data
    };
    vector<Vert> trie(1); // init with root
    void add_string(const string& s);
    void finalize() {
        queue<array<int, 4>> q{{{0, 0, 0, 0}}};
        while(size(q)) {
            auto [v, p, plink, pch] = q.front(); q.pop();
            int link = p == 0 ? 0 : trie[plink].next[pch];
            rep(c, A) {
                if(!trie[v].next[c]) // no child for c
                    trie[v].next[c] = trie[link].next[c];
                else
                    q.push({trie[v].next[c], v, link, c});
            } // after this loop: compute v's aggregates
        } // build automaton
    }
};
```

Why the $p == 0$ check? Is this enough for find pattern in text?
Aho-Corasick: Automaton Construction

Build an automaton based on the trie

- do a BFS and for each vertex
- compute the suffix link as \texttt{trie[link(parent)].next[pch]} 
- if there is no child for char \texttt{c}, copy transition from suffix link

Why the \texttt{p == 0} check?

Is this enough for \textit{find pattern in text}? 

- for our example (find pattern in text) we must aggregate the marked flag along suffix links

\begin{Verbatim}
trie[v].marked |= trie[link].marked;
\end{Verbatim}

\begin{Verbatim}
struct AhoCorasick {
    const static int A = 3; // alphabet size
    struct Vert {
        array<int,A> next = {};
        bool marked = false; // custom data
    }
    vector<Vert> trie(1); // init with root
    void add_string(const string& s);
    void finalize() {
        queue<array<int,4>> q{{{0,0,0,0}}};
        while(size(q)) {
            auto [v, p, plink, pch] = q.front(); q.pop();
            int link = p == 0 ? 0 : trie[plink].next[pch];
            rep(c,A) {
                if(!trie[v].next[c]) // no child for c
                    trie[v].next[c] = trie[link].next[c];
                else
                    q.push({trie[v].next[c], v, link, c});
            }
        } // after this loop: compute v’s aggregates
    }
}
\end{Verbatim}
Aho-Corasick: Examples

Find a shortest common superstring of $k$ strings over $\Sigma$ with total length $n$. 
Aho-Corasick: Examples

Find a shortest common superstring of $k$ strings over $\Sigma$ with total length $n$.

- Bitmasks for already contained strings
Aho-Corasick: Examples

Find a shortest common superstring of $k$ strings over $\Sigma$ with total length $n$.

- Bitmasks for already contained strings
- Consider automaton state + bitmask as a node
Aho-Corasick: Examples

Find a shortest common superstring of $k$ strings over $\Sigma$ with total length $n$.

- Bitmasks for already contained strings
- Consider automaton state + bitmask as a node
- BFS from root (vertex = 0, mask = 0) to (vertex =?, mask = $2^k – 1$)
Aho-Corasick: Examples

Find a shortest common superstring of $k$ strings over $\Sigma$ with total length $n$.

- Bitmasks for already contained strings
- Consider automaton state + bitmask as a node
- BFS from root ($vertex = 0, mask = 0$) to ($vertex = ?, mask = 2^k - 1$)
- worst case: $O(2^k|\Sigma|n)$
Aho-Corasick: Examples

Find a shortest common superstring of $k$ strings over $\Sigma$ with total length $n$.

- Bitmasks for already contained strings
- Consider automaton state + bitmask as a node
- BFS from root ($\text{vertex} = 0, \text{mask} = 0$) to ($\text{vertex} = ?, \text{mask} = 2^k - 1$)
- worst case: $O(2^k |\Sigma| n)$
- 2-approximation in $O(n)$
Aho-Corasick: Examples

Find a shortest common superstring of \( k \) strings over \( \Sigma \) with total length \( n \).

- Bitmasks for already contained strings
- Consider automaton state + bitmask as a node
- BFS from root \((\text{vertex} = 0, \text{mask} = 0)\) to \((\text{vertex} = ?, \text{mask} = 2^k - 1)\)
- worst case: \( O(2^k|\Sigma|n) \)
- 2-approximation in \( O(n) \)

Given set \( S \) of strings. Build string of length \( k \) with maximum number of matches with \( S \).
Aho-Corasick: Examples

Find a shortest common superstring of $k$ strings over $\Sigma$ with total length $n$.

- Bitmasks for already contained strings
- Consider automaton state + bitmask as a node
- BFS from root $(vertex = 0, mask = 0)$ to $(vertex = ?, mask = 2^k - 1)$
- worst case: $O(2^k|\Sigma|n)$
- 2-approximation in $O(n)$

Given set $S$ of strings. Build string of length $k$ with maximum number of matches with $S$.

- build Aho-Corasick automaton. Then do DP.
Aho-Corasick: Examples

Find a shortest common superstring of $k$ strings over $\Sigma$ with total length $n$.

- Bitmasks for already contained strings
- Consider automaton state + bitmask as a node
- BFS from root ($\text{vertex} = 0, \text{mask} = 0$) to ($\text{vertex} =?, \text{mask} = 2^k − 1$)
- worst case: $O(2^k|\Sigma|n)$
- 2-approximation in $O(n)$

Given set $S$ of strings. Build string of length $k$ with maximum number of matches with $S$.

- build Aho-Corasick automaton. Then do DP.
- $\text{DP}[k][\nu]$ is max. #matches with len $k$ input ending in state $\nu$
Aho-Corasick: Examples

Find a shortest common superstring of $k$ strings over $\Sigma$ with total length $n$.

- Bitmasks for already contained strings
- Consider automaton state + bitmask as a node
- BFS from root ($vertex = 0, mask = 0$) to ($vertex = ?, mask = 2^k - 1$)
- worst case: $O(2^k|\Sigma|n)$
- 2-approximation in $O(n)$

Given set $S$ of strings. Build string of length $k$ with maximum number of matches with $S$.

- build Aho-Corasick automaton. Then do DP.
- $DP[k][v]$ is max. #matches with len $k$ input ending in state $v$
- for $k$ rounds:
  - for each state extend maximum via all transitions to other vertices