Karlsruhe Institute of Technology

Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

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
imgflip.com

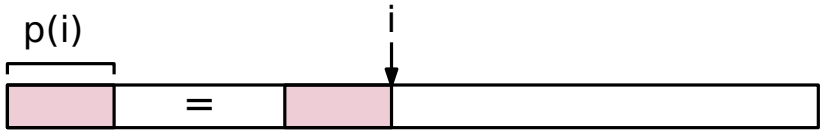
Prefix Function

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 $t[0..i]$

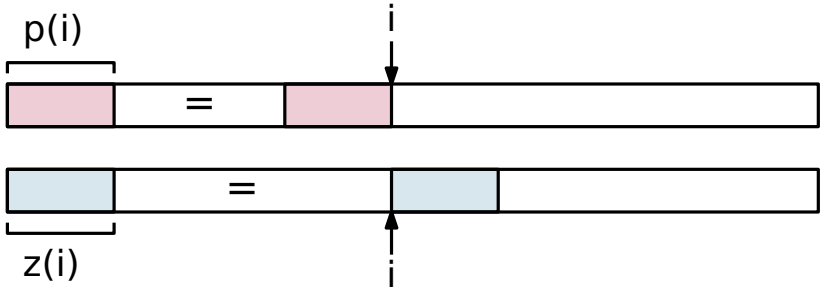


$t =$	a	b	a	b	a	a	b	c	a	b
$p =$	0	0	1	2	3	1	2	0	1	2

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 $t[i \dots n-1]$

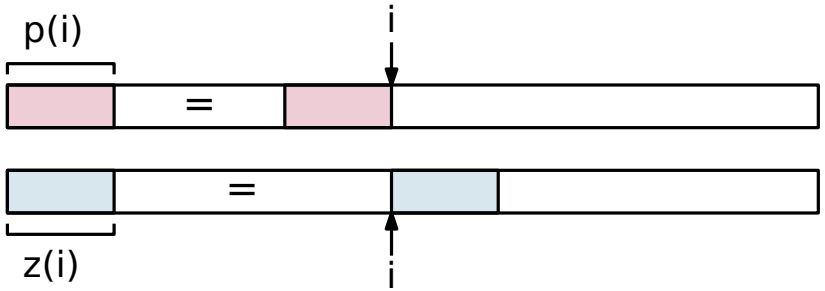


$t =$	a	b	a	b	a	a	b	c	a	b
$p =$	0	0	1	2	3	1	2	0	1	2
$z =$	0	0	3	0	1	2	0	0	2	0

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 $t[i \ldots n-1]$



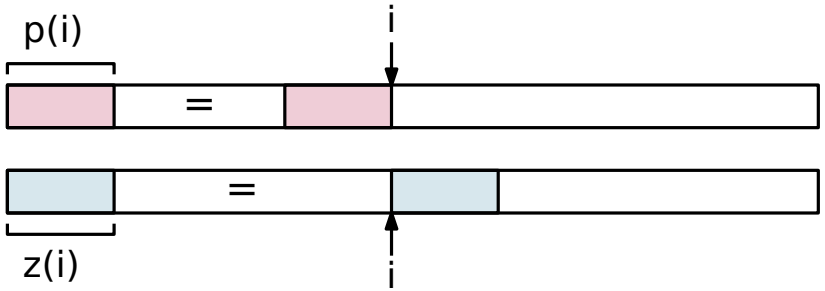
$t =$	a	b	a	b	a	a	b	c	a	b
$p =$	0	0	1	2	3	1	2	0	1	2
$z =$	0	0	3	0	1	2	0	0	2	0

```
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 $t[i \ldots n-1]$



$t =$	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

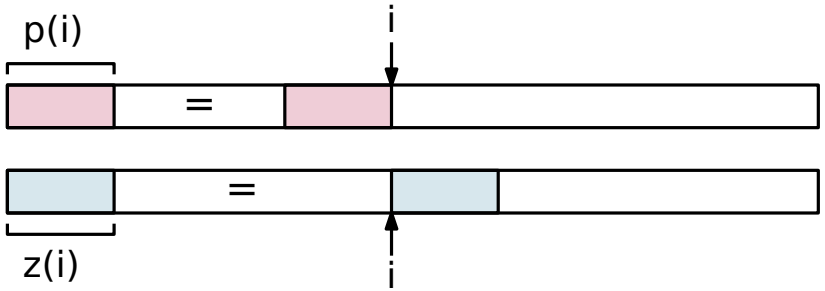
find period of periodic string?

```
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 $t[i \dots n-1]$



$t =$	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

find period of periodic string?

$n = 10$

```
0012345678
ababababab
0080604020
```

```
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]);
}
```

#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 $O(1) + cost(hashing)$

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 $O(1) + cost(hashing)$

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 $O(1) + cost(hashing)$

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 $O(1) + cost(hashing)$

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 $O(1) + cost(hashing)$

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
- $cost(hashing)$ 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.

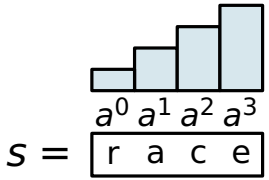
$$\begin{aligned} H(s) = \quad & s[0]a^0 + s[1]a^1 + s[2]a^2 + \cdots + s[n-1]a^{n-1} \mod p \\ = & \sum_{i=0}^{n-1} s[i] \cdot a^i \mod p \end{aligned}$$

Polynomial Rolling Hashes

Given a string s of length n.

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

$$\begin{aligned} H(s) &= s[0]a^0 + s[1]a^1 + s[2]a^2 + \cdots + s[n-1]a^{n-1} \mod p \\ &= \sum_{i=0}^{n-1} s[i] \cdot a^i \mod p \end{aligned}$$

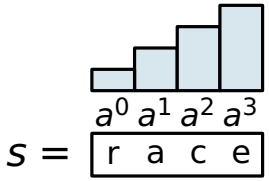


Polynomial Rolling Hashes

Given a string s of length n.

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

$$\begin{aligned} \mathrm{H(s)} &= s[0]a^0 + s[1]a^1 + s[2]a^2 + \cdots + s[n-1]a^{n-1} \mod p \\ &= \sum_{i=0}^{n-1} s[i] \cdot a^i \mod p \end{aligned}$$



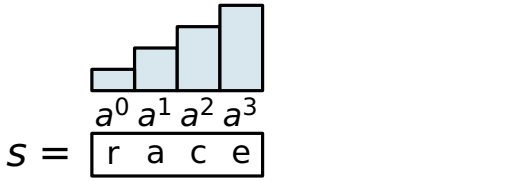
$$\mathrm{H(s)} = \mathtt{r}a^0 + \mathtt{a}a^1 + \mathtt{c}a^2 + \mathtt{e}a^3$$

Polynomial Rolling Hashes

Given a string s of length n.

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

$$\begin{aligned} H(s) &= s[0]a^0 + s[1]a^1 + s[2]a^2 + \cdots + s[n-1]a^{n-1} \mod p \\ &= \sum_{i=0}^{n-1} s[i] \cdot a^i \mod p \end{aligned}$$



$$H(s) = \texttt{r}a^0 + \texttt{a}a^1 + \texttt{c}a^2 + \texttt{e}a^3$$

how good is this hash function?

Polynomial Rolling Hashes

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

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

Polynomial Rolling Hashes

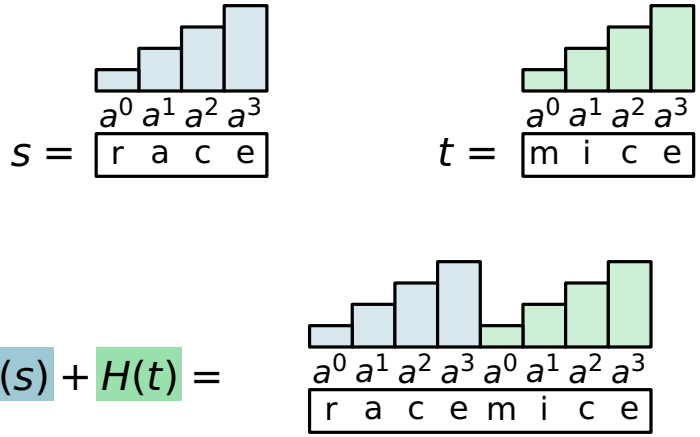
$$H(s) = \left(\sum_{i=0}^{n-1} a^i \cdot s[i]\right) \mod p$$

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

Polynomial Rolling Hashes

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

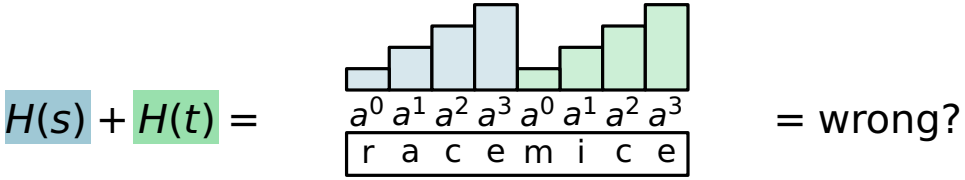
Assume we have hashes for strings s, t. How to compute the hash of st?



Polynomial Rolling Hashes

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

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

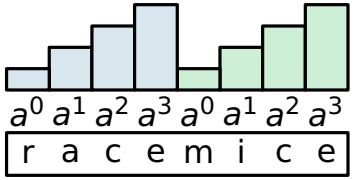


Polynomial Rolling Hashes

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

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

$H(s) + H(t) =$



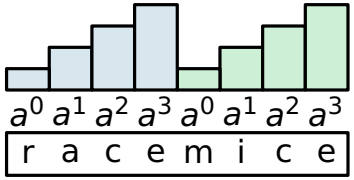
= wrong? the coefficients on t are too low

Polynomial Rolling Hashes

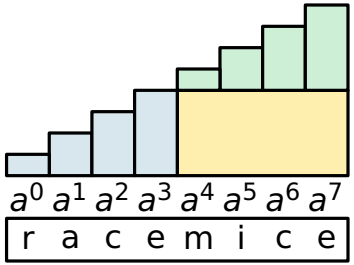
$$H(s) = \left(\sum_{i=0}^{n-1} a^i \cdot s[i]\right) \mod p$$

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

$H(s) + H(t) =$



= wrong? the coefficients on t are too low



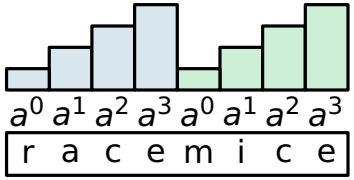
$= H(st)$

Polynomial Rolling Hashes

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

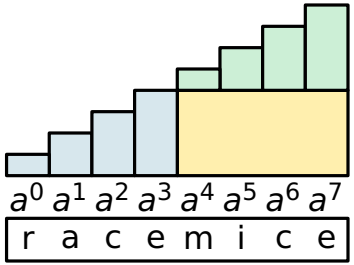
Assume we have hashes for strings s, t. How to compute the hash of st?

$H(s) + H(t) =$



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

$H(s) + a^{|s|} \cdot H(t) =$



$= H(st)$

Polynomial Rolling Hashes

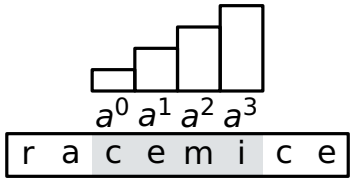
$$H(s) = \left(\sum_{i=0}^{n-1} a^i \cdot s[i]\right) \mod p$$

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

Polynomial Rolling Hashes

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

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

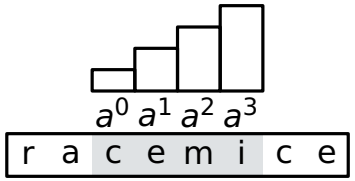


$H(s[\ell, r])$

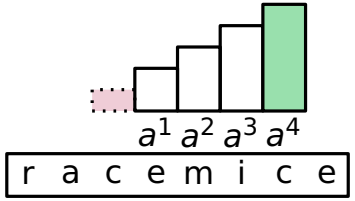
Polynomial Rolling Hashes

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

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



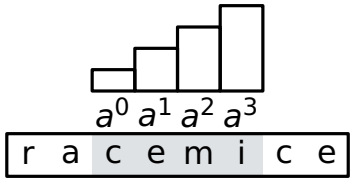
$H(s[\ell, r])$



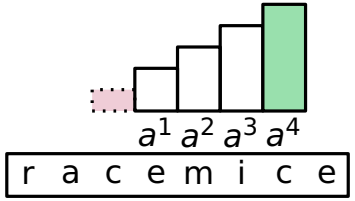
Polynomial Rolling Hashes

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

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



$H(s[\ell, r])$

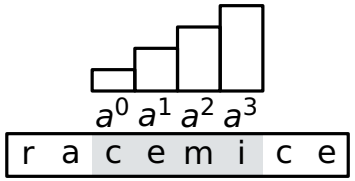


$H(s[\ell, r]) + a^{r+1-\ell} \cdot s[r+1] - s[l]$

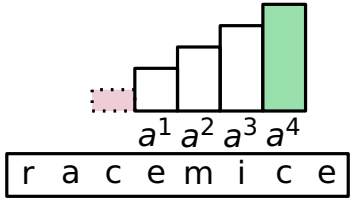
Polynomial Rolling Hashes

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

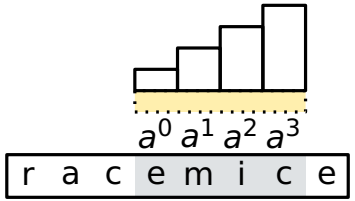
Assume we have the hash of $s[\ell, r]$. How to compute the hash of $s[\ell+1, r+1]$?



$H(s[\ell, r])$



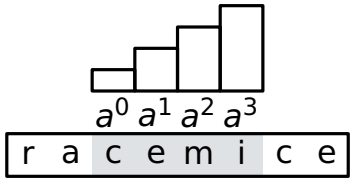
$H(s[\ell, r]) + a^{r+1-\ell} \cdot s[r+1] - s[l]$



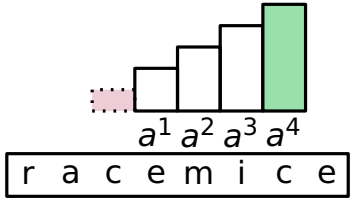
Polynomial Rolling Hashes

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

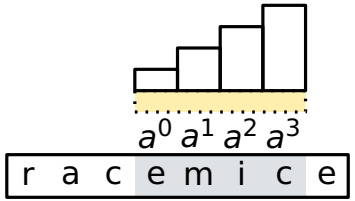
Assume we have the hash of $s[\ell, r]$. How to compute the hash of $s[\ell+1, r+1]$?



$$H(s[\ell, r])$$



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

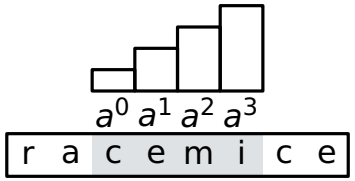


$$\frac{H(s[\ell, r]) + a^{r+1-\ell} \cdot s[r+1] - s[l]}{a}$$

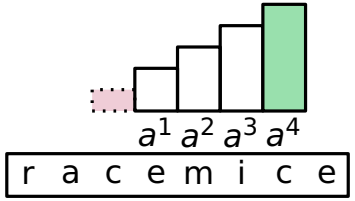
Polynomial Rolling Hashes

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

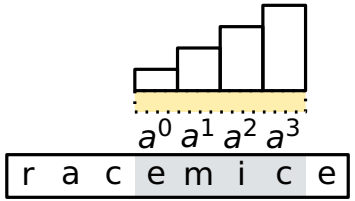
Assume we have the hash of $s[\ell, r]$. How to compute the hash of $s[\ell+1, r+1]$?



$$H(s[\ell, r])$$



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

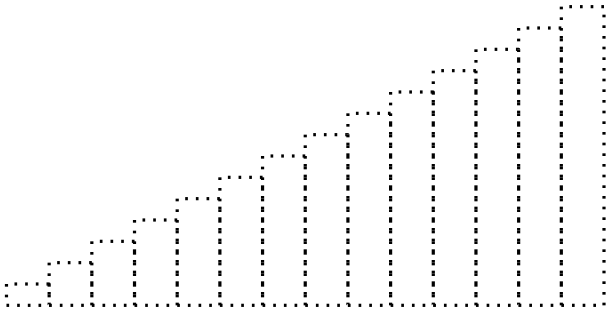


$$\frac{H(s[\ell, r]) + a^{r+1-\ell} \cdot s[r+1] - s[l]}{a} = H(s[\ell+1, r+1])$$

Polynomial Rolling Hashes

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

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



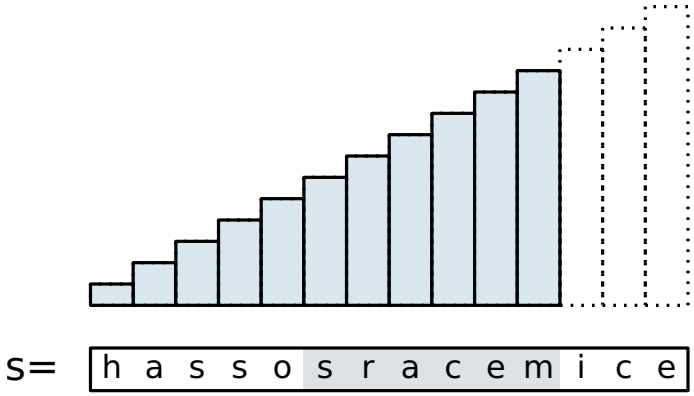
s= h a s s o s r a c e m i c e

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

Polynomial Rolling Hashes

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

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

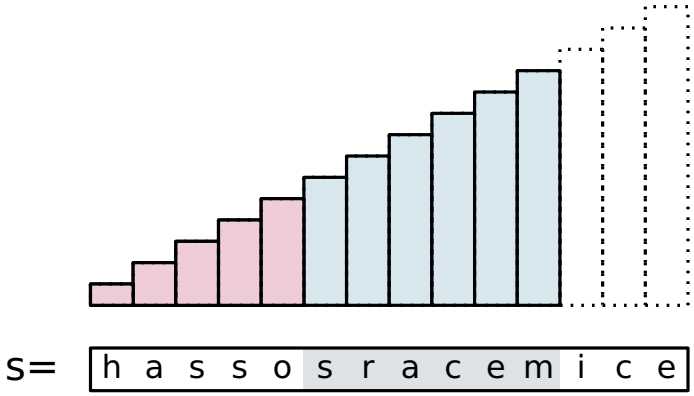


$$H(s[\ell, r]) = H(s[0, r])$$

Polynomial Rolling Hashes

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

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

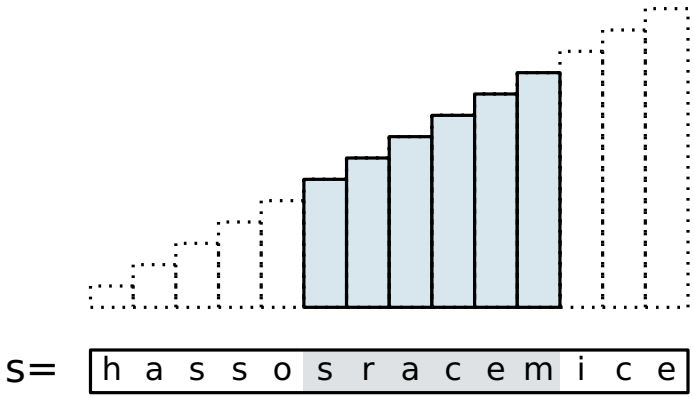


$$H(s[\ell, r]) = H(s[0, r]) - H(s[0, \ell - 1])$$

Polynomial Rolling Hashes

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

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

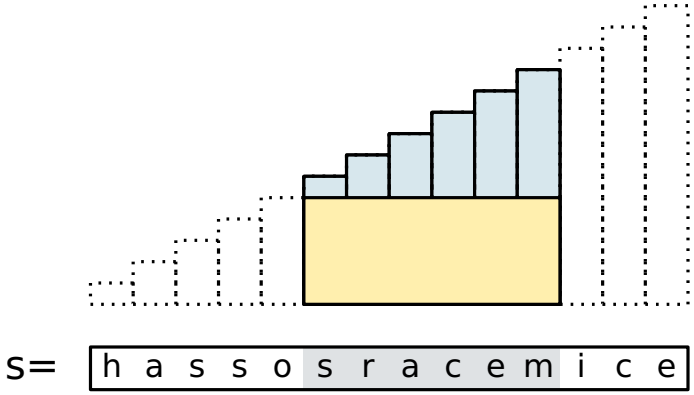


$$H(s[\ell, r]) = H(s[0, r]) - H(s[0, \ell - 1])$$

Polynomial Rolling Hashes

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

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

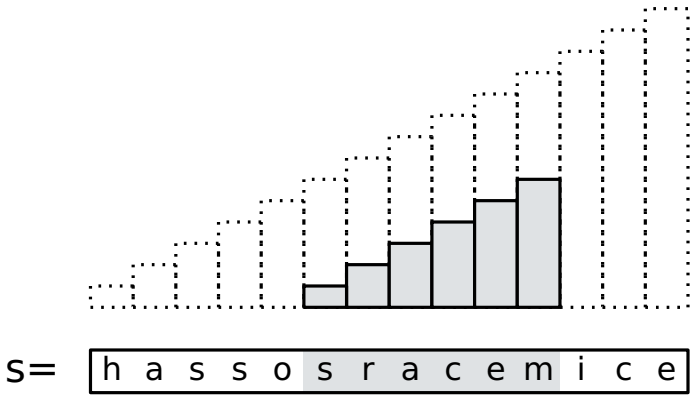


$$H(s[\ell, r]) = \frac{H(s[0, r]) - H(s[0, \ell - 1])}{a^\ell}$$

Polynomial Rolling Hashes

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

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



$$H(s[\ell, r]) = \frac{H(s[0, r]) - H(s[0, \ell - 1])}{a^\ell}$$

Polynomial Rolling Hashes

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

Tips

Polynomial Rolling Hashes

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

Tips

- precompute powers of a and their inverse

Polynomial Rolling Hashes

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

Tips

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

Polynomial Rolling Hashes

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

Tips

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

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

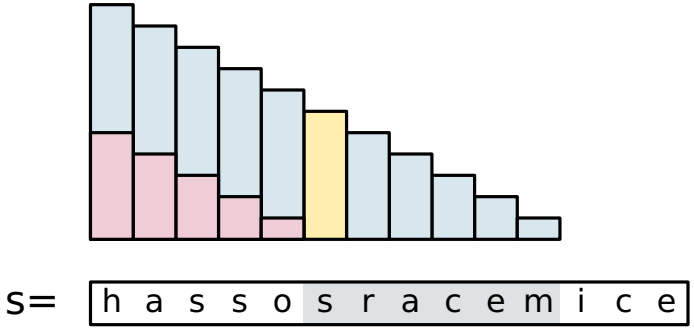
Polynomial Rolling Hashes

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

Tips

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

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



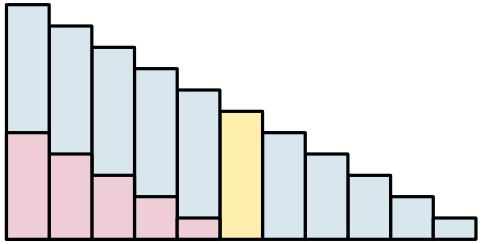
Polynomial Rolling Hashes

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

Tips

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

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



s= h a s s o s r a c e m i c e

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

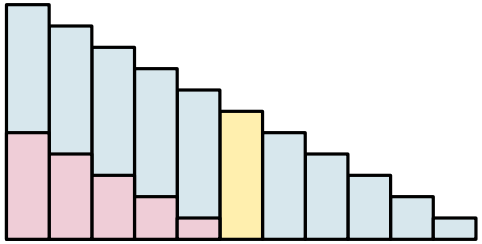
Polynomial Rolling Hashes

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

Tips

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

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



s= h a s s o s r a c e m i c e

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

- don't start character numbering at 0

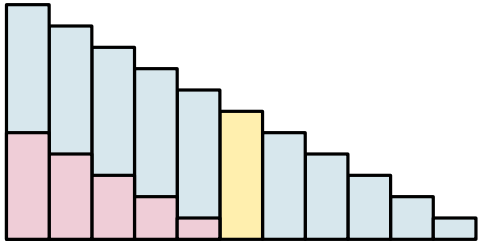
Polynomial Rolling Hashes

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

Tips

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

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



s= h a s s o s r a c e m i c e

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

- don't start character numbering at 0 why?

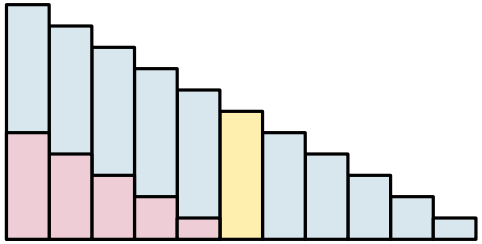
Polynomial Rolling Hashes

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

Tips

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

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



s= h a s s o s r a c e m i c e

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

- don't start character numbering at 0 why?

$$H(a) = H(aaaa) = 0$$

$$H(pizza) = H(pizzaaa)$$

Polynomial Rolling Hashes

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

Collisions

Polynomial Rolling Hashes

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

Collisions

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

Polynomial Rolling Hashes

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

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 $\approx 10^{4.5}$

Polynomial Rolling Hashes

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

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'

Polynomial Rolling Hashes

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

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

Polynomial Rolling Hashes

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

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

https://codeforces.com/blog/entry/60442

Polynomial Rolling Hashes

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

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
  https://codeforces.com/blog/entry/60442
- **Fix:** randomize your base a at runtime

Polynomial Rolling Hashes

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

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
  https://codeforces.com/blog/entry/60442
- **Fix:** randomize your base a at runtime

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

Polynomial Rolling Hashes

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

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
  https://codeforces.com/blog/entry/60442
- **Fix:** randomize your base a at runtime

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

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

Polynomial Rolling Hashes

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

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
  https://codeforces.com/blog/entry/60442
- **Fix:** randomize your base a at runtime

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

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

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

Polynomial Rolling Hashes

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

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
  https://codeforces.com/blog/entry/60442
- **Fix:** randomize your base a at runtime

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

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

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

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

Polynomial Rolling Hashes

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

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
  https://codeforces.com/blog/entry/60442
- **Fix:** randomize your base a at runtime

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

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

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

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

Suffix Array

- Given a string s of length n. We call the substring $s[i \dots n-1]$ the i-th suffix of s.

Suffix Array

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

i	i-th suffix
0	racemice
1	acemice
2	cemice
3	emice
4	mice
5	ice
6	ce
7	e

Suffix Array

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

i	i-th suffix
0	racemice
1	acemice
2	cemice
3	emice
4	mice
5	ice
6	ce
7	e

i	i-th suffix
1	acemice
6	ce
2	cemice
7	e
3	emice
5	ice
4	mice
0	racemice

Suffix Array

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

i	i-th suffix
0	racemice
1	acemice
2	cemice
3	emice
4	mice
5	ice
6	ce
7	e

i	i-th suffix
1	acemice
6	ce
2	cemice
7	e
3	emice
5	ice
4	mice
0	racemice

- 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

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

- $O(n \log n)$ comparisons

Suffix Array

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

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

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

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

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

```
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

```
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

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

LCP:
cababca
cabcba

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

    LCP = longest common prefix
- $O(\log n)$ for each string comparison

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

  (very high constants in practice)

LCP:
cababca
cabcba

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.

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

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

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\$

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

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

i	i-th cyclic shift
8	$racemice
1	acemice$r
6	ce$racemi
2	cemice$ra
7	e$racemic
3	emice$rac
5	ice$racem
4	mice$race
0	racemice$

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

i	i-th cyclic shift
8	$racemice
1	acemice$r
6	ce$racemi
2	cemice$ra
7	e$racemic
3	emice$rac
5	ice$racem
4	mice$race
0	racemice$

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

i	i-th suffix
1	acemice
6	ce
2	cemice
7	e
3	emice
5	ice
4	mice
0	racemice

i	i-th cyclic shift
8	$racemice
1	acemice$r
6	ce$racemi
2	cemice$ra
7	e$racemic
3	emice$rac
5	ice$racem
4	mice$race
0	racemice$

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

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

$\rightarrow O(n \log^2 n)$ 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);
```

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];
```

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

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

i	i-th cyclic shift
1	racemice
6	racemice
2	racemice
3	racemice
7	racemice
5	racemice
4	racemice
0	racemice

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

i	i-th cyclic shift
1	racemice
6	racemice
2	racemice
3	racemice
7	racemice
5	racemice
4	racemice
0	racemice

Sort Cyclic Shifts

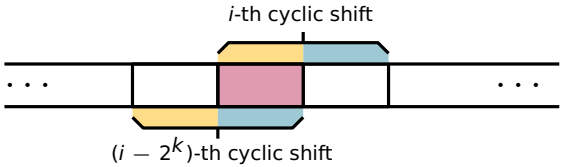
- equiv. classes of the parts do not exceed n
- how to do counting sort on pairs?
- use radix sort
  - sort by second half
  - then stable sort by first half
    (with counting sort)
- observation: substrings are already sorted by their first 2^k characters from last iteration
- but we want them to be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

i	i-th cyclic shift
1	racemice
6	racemice
2	racemice
3	racemice
7	racemice
5	racemice
4	racemice
0	racemice

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

i	i-th cyclic shift
1	racemice
6	racemice
2	racemice
3	racemice
7	racemice
5	racemice
4	racemice
0	racemice



Sort Cyclic Shifts

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];
});
```

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]
```

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

- 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;
```

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

Sort Cyclic Shifts

sort length 1 substrings

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

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

- 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;
```

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

Sort Cyclic Shifts

sort length 1 substrings

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

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

- sort by second half (via shift)
- stable counting sort by first half
- build classes for length 2^{k+1} strings

$\rightarrow O(n \log n)$ 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.

i j

↓ ↓

CGAAGTAAATAAGTAC

Longest Common Prefix (LCP)

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

$$\begin{array}{ccc} i & & j \\ \downarrow & & \downarrow \\ \text{CG}\underbrace{\text{AAGTA}}\text{AAT}\underbrace{\text{AAGTA}}\text{C} & & \end{array}$$

$\mathrm{lcp}(i, j) = |\mathrm{AAGTA}| = 5$

Longest Common Prefix (LCP)

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

$$\text{CG}\underbrace{\text{AAGTA}}\text{AAT}\underbrace{\text{AAGTA}}\text{C}$$

i j

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

- 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.

$$\begin{array}{ccc} i & & j \\ \downarrow & & \downarrow \\ \text{CG}\underbrace{\text{AAGTA}}\text{AAT}\underbrace{\text{AAGTA}}\text{C} & & \end{array}$$

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

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

  w/o hashing + binary search

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

Longest Common Prefix (LCP)

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

$$\text{CG}\underbrace{\text{AAGTA}}\text{AAT}\underbrace{\text{AAGTA}}\text{C} \qquad \text{lcp}(i,j) = |\text{AAGTA}| = 5$$

- How do we compute the lcp of two suffixes starting at indices i and j?
  w/o hashing + binary search
- 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.

i j

CGAAGTAAATAAGTAC

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

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

  w/o hashing + binary search

- 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

LCP of Suffixes starting at i, j

ans = 0

for k from logn to 0:

  if $s[i, i + 2^k) == s[j, j + 2^k)$:

    ans += 2^k

    i += 2^k

    j += 2^k

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.

sa[i]	$sa[i]$-th suffix
6	AAATAAGTAC
2	AAGTAAATAAGTAC
10	AAGTAC
7	AATAAGTAC
14	AC
3	AGTAAATAAGTAC
11	AGTAC
8	ATAAGTAC
15	C
0	CGAAGTAAATAAGTAC
1	GAAGTAAATAAGTAC
4	GTAAATAAGTAC
12	GTAC
5	TAAATAAGTAC
9	TAAGTAC
13	TAC

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.

sa[i]	*sa*[*i*]-th suffix
6	AAATAAGTAC
2	AAGTAAATAAGTAC
10	AAGTAC
7	AATAAGTAC
14	AC
3	AGTAAATAAGTAC
11	AGTAC
8	ATAAGTAC
15	C
0	CGAAGTAAATAAGTAC
1	GAAGTAAATAAGTAC
4	GTAAATAAGTAC
12	GTAC
5	TAAATAAGTAC
9	TAAGTAC
13	TAC

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.

sa[i]	*sa*[*i*]-th suffix	lcp-array[i]
6	AAATAAGTAC	2
2	AAGTAAATAAGTAC	5
10	AAGTAC	2
7	AATAAGTAC	1
14	AC	1
3	AGTAAATAAGTAC	4
11	AGTAC	1
8	ATAAGTAC	0
15	C	1
0	CGAAGTAAATAAGTAC	0
1	GAAGTAAATAAGTAC	1
4	GTAAATAAGTAC	3
12	GTAC	0
5	TAAATAAGTAC	3
9	TAAGTAC	2
13	TAC	

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.

sa[i]	$sa[i]$-th suffix	lcp-array[i] $= \text{lcp}(sa[i], sa[i+1])$
6	AAATAAGTAC	2
2	AAGTAAATAAGTAC	5
10	AAGTAC	2
7	AATAAGTAC	1
14	AC	1
3	AGTAAATAAGTAC	4
11	AGTAC	1
8	ATAAGTAC	0
15	C	1
0	CGAAGTAAATAAGTAC	0
1	GAAGTAAATAAGTAC	1
4	GTAAATAAGTAC	3
12	GTAC	0
5	TAAATAAGTAC	3
9	TAAGTAC	2
13	TAC	

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.

sa[i]	$sa[i]$-th suffix	lcp-array[i] $= \text{lcp}(sa[i], sa[i+1])$
6	AAATAAGTAC	2
2	AAGTAAATAAGTAC	5
10	AAGTAC	2
7	AATAAGTAC	1
14	AC	1
3	AGTAAATAAGTAC	4
11	AGTAC	1
8	ATAAGTAC	0
15	C	1
0	CGAAGTAAATAAGTAC	0
1	GAAGTAAATAAGTAC	1
4	GTAAATAAGTAC	3
12	GTAC	0
5	TAAATAAGTAC	3
9	TAAGTAC	2
13	TAC	

s=CGAAGTAAATAAGTAC

$\text{lcp}(2, 11)$

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.

sa[i]	$sa[i]$-th suffix	lcp-array[i] $= \text{lcp}(sa[i], sa[i+1])$
6	AAATAAGTAC	2
2	AAGTAAATAAGTAC	5
10	AAGTAC	2
7	AATAAGTAC	1
14	AC	1
3	AGTAAATAAGTAC	4
11	AGTAC	1
8	ATAAGTAC	0
15	C	1
0	CGAAGTAAATAAGTAC	0
1	GAAGTAAATAAGTAC	1
4	GTAAATAAGTAC	3
12	GTAC	0
5	TAAATAAGTAC	3
9	TAAGTAC	2
13	TAC	

s=CGAAGTAAATAAGTAC

$\text{lcp}(2, 11) = 1$

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.

sa[i]	$sa[i]$-th suffix	lcp-array[i] $= \text{lcp}(sa[i], sa[i+1])$
6	AAATAAGTAC	2
2	AAGTAAATAAGTAC	5
10	AAGTAC	2
7	AATAAGTAC	1
14	AC	1
3	AGTAAATAAGTAC	4
11	AGTAC	1
8	ATAAGTAC	0
15	C	1
0	CGAAGTAAATAAGTAC	0
1	GAAGTAAATAAGTAC	1
4	GTAAATAAGTAC	3
12	GTAC	0
5	TAAATAAGTAC	3
9	TAAGTAC	2
13	TAC	

min

s=CGAAGTAAATAAGTAC

Observation: $\text{lcp}(sa[i], sa[j]) = \min(\text{lcp-array}[i, j))$

$\text{lcp}(2, 11) = 1$

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.

sa[i]	$sa[i]$-th suffix	lcp-array[i] $= \text{lcp}(sa[i], sa[i+1])$
6	AAATAAGTAC	2
2	AAGTAAATAAGTAC	5
10	AAGTAC	2
7	AATAAGTAC	1
14	AC	1
3	AGTAAATAAGTAC	4
11	AGTAC	1
8	ATAAGTAC	0
15	C	1
0	CGAAGTAAATAAGTAC	0
1	GAAGTAAATAAGTAC	1
4	GTAAATAAGTAC	3
12	GTAC	0
5	TAAATAAGTAC	3
9	TAAGTAC	2
13	TAC	

min

s=CGAAGTAAATAAGTAC

Observation: $\text{lcp}(sa[i], sa[j]) = \min(\text{lcp-array}[i, j))$

$\rightarrow$ lcp can be reducted to range minimum queries

$\text{lcp}(2, 11) = 1$

Suffix Array and LCP Array Overview

Suffix Array

Suffix Array and LCP Array Overview

Suffix Array	Naive	$O(n^2 \log n)$

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

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

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	$O(n \log n)$

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	$O(n)$

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)	$O(n)$

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)	$O(n)$
LCP	Fit 2s powers with SA classes	$O(\log n)$ query; $O(n \log n)$ 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 2s powers with SA classes	$O(\log n)$ query; $O(n \log n)$ 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 2s 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

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 2s 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

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 2s 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
	Sparse Table	$O(1)$ query; $O(n \log n)$ 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 2s 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
	Sparse Table	$O(1)$ query; $O(n \log n)$ construct

*relevant for us

15min Break

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

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

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: $O(nm)$

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

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

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: $O(n^2)$

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.

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: $O((n+m)\log(n+m))^*$

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

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: $O((n+m)\log(n+m))^*$

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

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.

#substrings - sum(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

Aho-Corasick: Motivation

$|s| = n$
$|t_i| = m_i$

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

Aho-Corasick: Motivation

$|s| = n$
$|t_i| = m_i$

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

naive: $O(nm_i)$ per pattern

Aho-Corasick: Motivation

$|s| = n$
$|t_i| = m_i$

Find Pattern in Text:
Find all occurences 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

$|s| = n$
$|t_i| = m_i$

Find Pattern in Text:
Find all occurences 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

Aho-Corasick: Motivation

$|s| = n$
$|t_i| = m_i$

Find Pattern in Text:
Find all occurences 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: $O\left(n \log n + \sum m_i \log n\right)$

Aho-Corasick: Motivation

$|s| = n$
$|t_i| = m_i$

Find Pattern in Text:
Find all occurences 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: $O\left(n \log n + \sum m_i \log n\right)$

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

Aho-Corasick: Motivation

$|s| = n$
$|t_i| = m_i$

Find Pattern in Text:
Find all occurences 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: $O\left(n \log n + \sum m_i \log n\right)$

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

finite state machine (automaton) on the trie of $t_1, t_2 \ldots$

Aho-Corasick: Tries

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

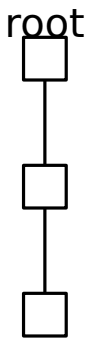
Trie for *ab*

Aho-Corasick: Tries

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

- there is a root vertex

Trie for *ab*

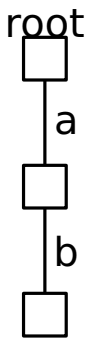


Aho-Corasick: Tries

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

Trie for *ab*

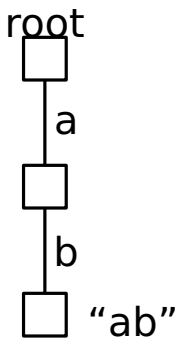


Aho-Corasick: Tries

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

Trie for *ab*

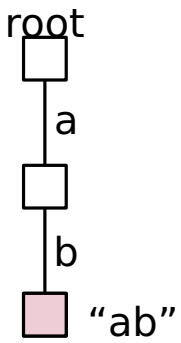


Aho-Corasick: Tries

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*

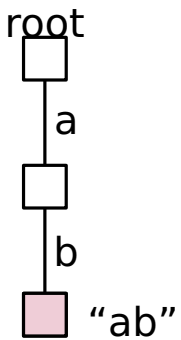


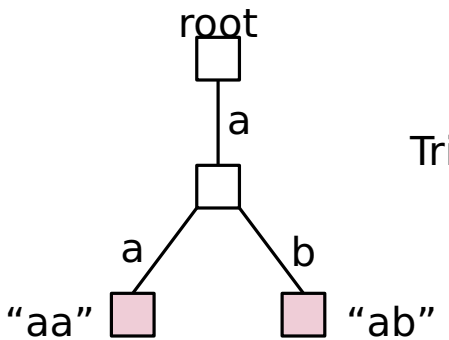
Aho-Corasick: Tries

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*





Trie for *ab* and *aa*

Aho-Corasick: Tries

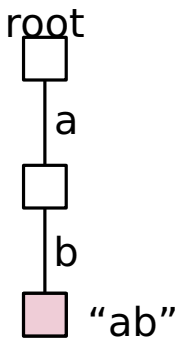
A **trie** stores a set of strings. It is a rooted tree with the following features:

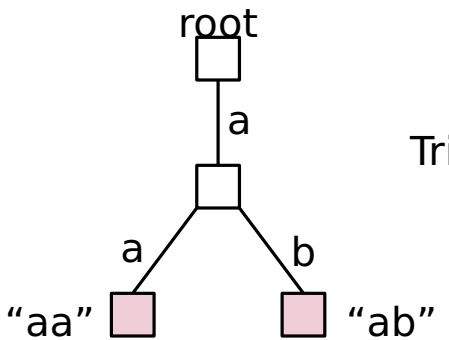
- there is a root vertex
- every edge is labeled by some character
- path from root to a vertex corresponds to prefix of a stored string
- a vertex is marked if its corresponding string is one of the stored strings

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

i.e., linear in input for constant alphabet

Trie for *ab*





Trie for *ab* and *aa*

Aho-Corasick: Trie Construction

Trie Construction

```
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

```
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*

```
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

```
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

```
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

```
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) {
        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 $\Leftrightarrow$ longest prefix of some s ending at the current input

Aho-Corasick: Automaton Construction

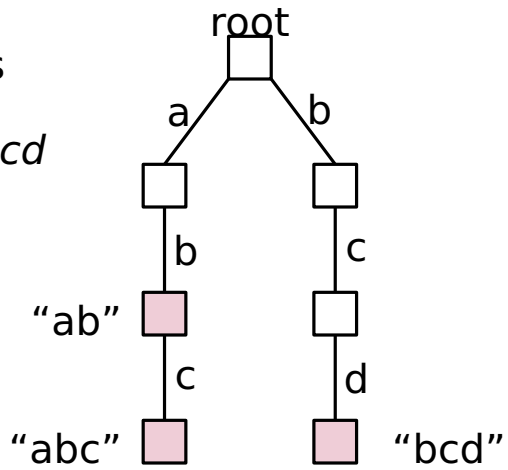
Build an automaton based on the trie representing the set of strings.

- 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 ⇔ longest prefix of some *s* ending at the current input

Strings

abc, *ab*, *bcd*



Text

Aho-Corasick: Automaton Construction

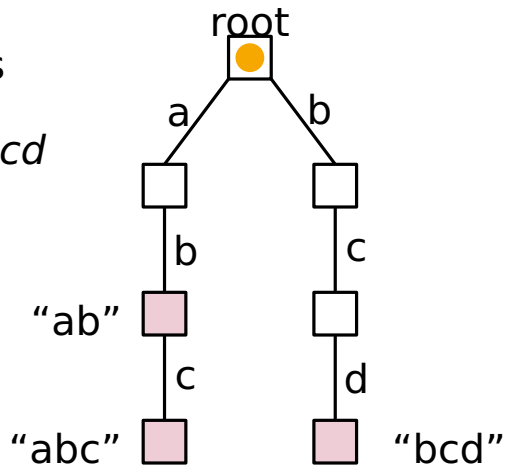
Build an automaton based on the trie representing the set of strings.

- 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

Aho-Corasick: Automaton Construction

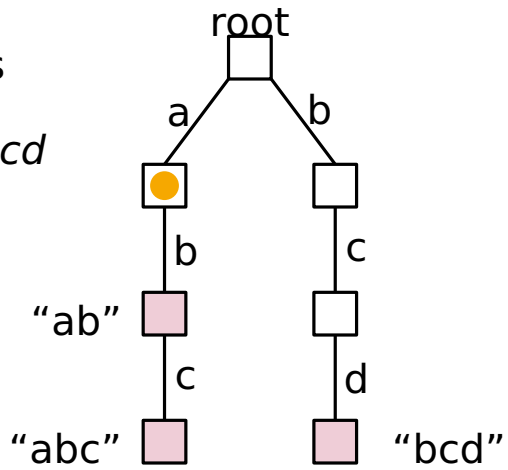
Build an automaton based on the trie representing the set of strings.

- 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

Aho-Corasick: Automaton Construction

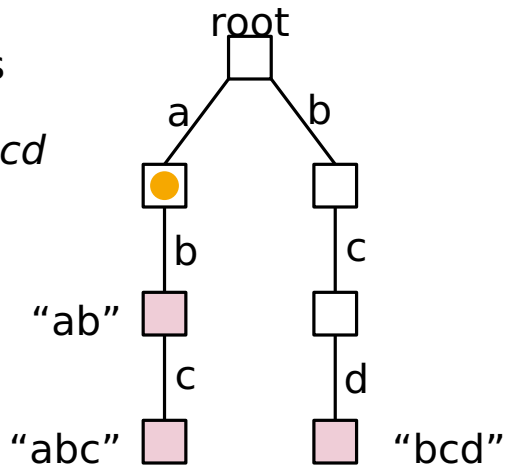
Build an automaton based on the trie representing the set of strings.

- 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

Aho-Corasick: Automaton Construction

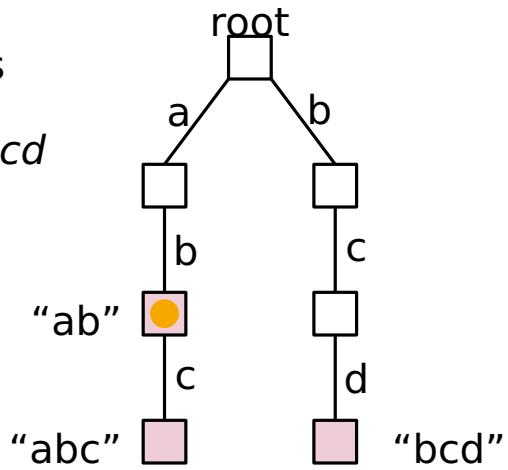
Build an automaton based on the trie representing the set of strings.

- 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

Aho-Corasick: Automaton Construction

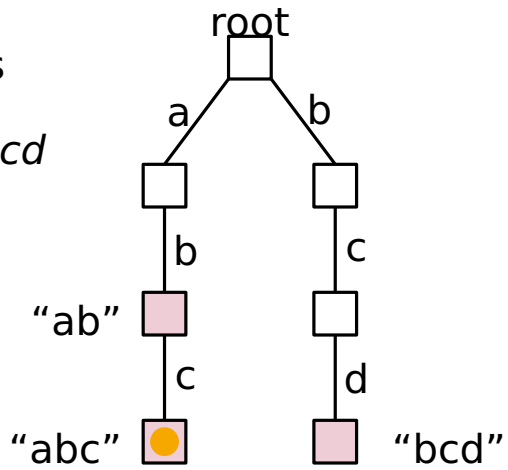
Build an automaton based on the trie representing the set of strings.

- 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

Aho-Corasick: Automaton Construction

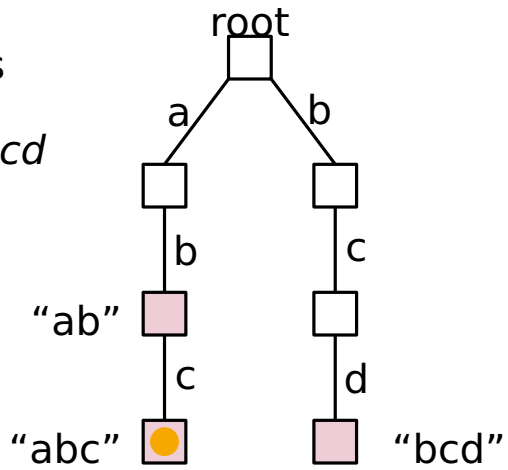
Build an automaton based on the trie representing the set of strings.

- 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

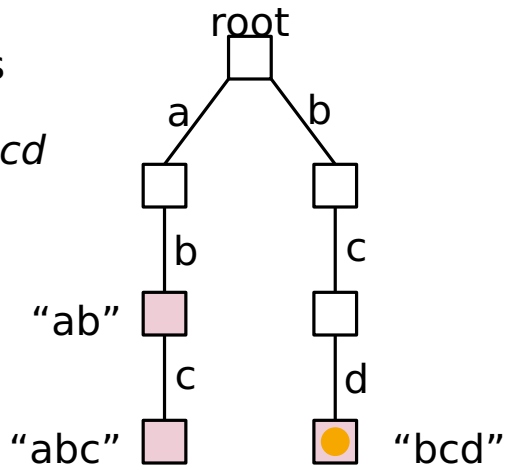
Build an automaton based on the trie representing the set of strings.

- 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 ⇔ longest prefix of some *s* ending at the current input

Strings

abc, *ab*, *bcd*

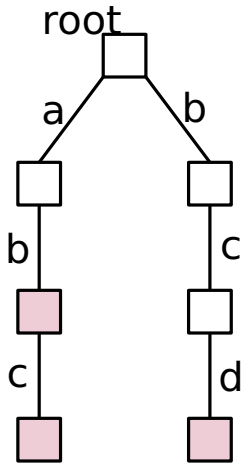


Text

a a b c d

Aho-Corasick: Automaton Construction

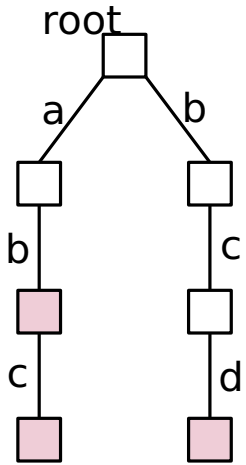
Build an automaton based on the trie representing the set of strings.



Aho-Corasick: Automaton Construction

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

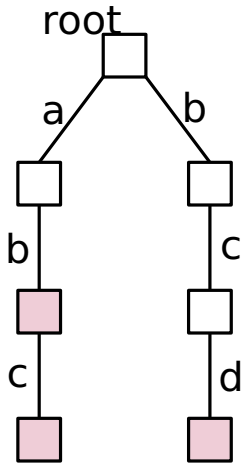
- needs a transition for every possible character at each state



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., ε-transitions via so called **suffix-links**

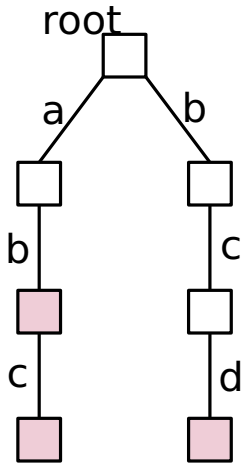


Aho-Corasick: Automaton Construction

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., ε-transitions via so called **suffix-links**

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

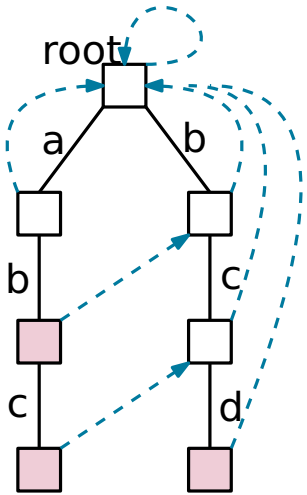


Aho-Corasick: Automaton Construction

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., ε-transitions via so called **suffix-links**

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

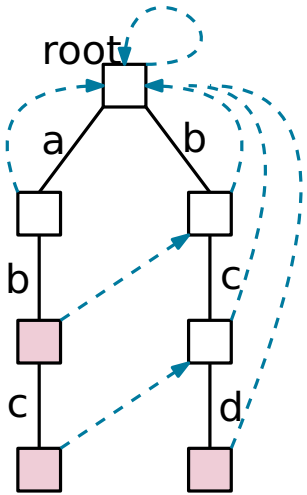


Aho-Corasick: Automaton Construction

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., ε-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

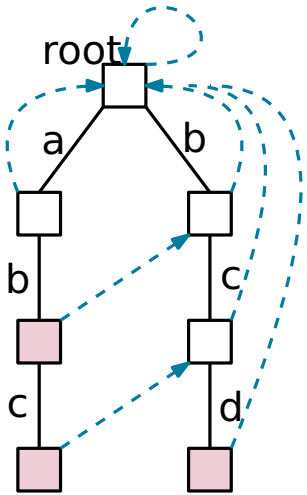


Aho-Corasick: Automaton Construction

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., ε-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



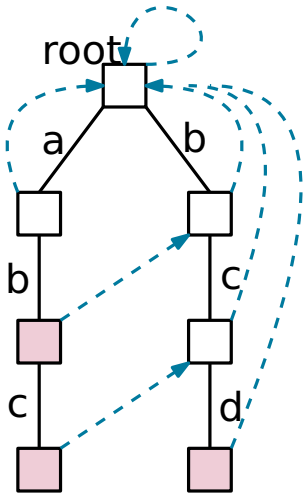
Aho-Corasick: Automaton Construction

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., ε-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



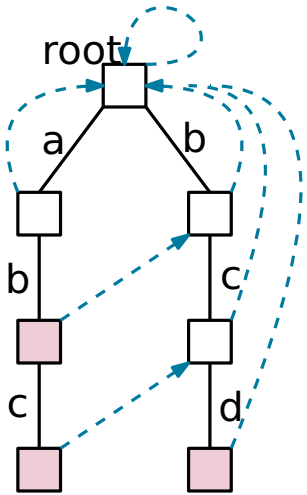
Aho-Corasick: Automaton Construction

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., ε-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



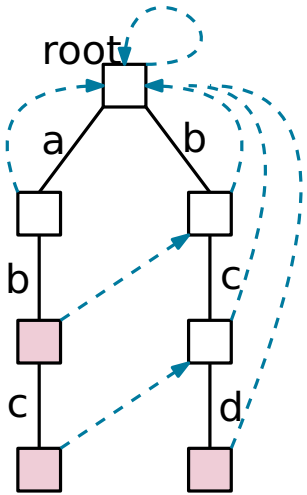
Aho-Corasick: Automaton Construction

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., ε-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

```
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

```
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*]

```
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

```
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

```
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

```
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?

```
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*?

```
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
};
```

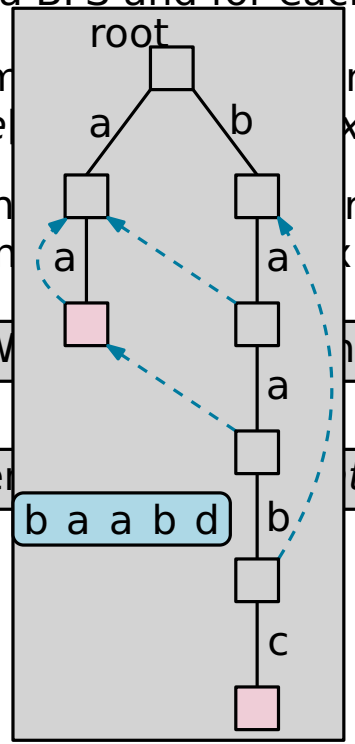
Aho-Corasick: Automaton Construction

Build an automaton based on the trie

- do a BFS and for each vertex
- com nk as *trie xt[pch]*
- if th r char *c*, copy tran link

W heck?

Is this e ttern 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

```
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

```
trie[v].marked |= trie[link].marked;
```

```
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: Examples

Find a shortest common superstring of k strings over Σ with total length n.

Aho-Corasick: Examples

Find a shortest common superstring of k strings over Σ with total length n.

- Bitmasks for already contained strings

Aho-Corasick: Examples

Find a shortest common superstring of k strings over Σ 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 Σ 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 Σ 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 Σ 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 $O(n)$

Ukkonen; 1990; A linear-time algorithm for finding approximate shortest common superstrings

Aho-Corasick: Examples

Find a shortest common superstring of k strings over Σ 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 $O(n)$

Ukkonen; 1990; A linear-time algorithm for finding approximate shortest common superstrings

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 Σ 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 $O(n)$

Ukkonen; 1990; A linear-time algorithm for finding approximate shortest common superstrings

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 Σ 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 $O(n)$

Ukkonen; 1990; A linear-time algorithm for finding approximate shortest common superstrings

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

Aho-Corasick: Examples

Find a shortest common superstring of k strings over Σ 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 $O(n)$

Ukkonen; 1990; A linear-time algorithm for finding approximate shortest common superstrings

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