Aalto University

Competitive Programming 2020

3 Strings algorithms: searching & indexing

Today's program

- **12:15:** Lecture (Zoom)
- **13:00:** Practice contest (CSES)
  - multiple problems to choose from
  - *try to solve at least 2 problems*
  - try to solve first on your own
  - if no progress: help available starting at **14:00**
- **16:00:** Post-contest wrap-up (Zoom)

String algorithms

- **Searching & pattern matching**
  - answer ***one*** query efficiently
- **Indexing & sorting**
  - preprocess text so that you can answer ***many*** queries efficiently

String algorithms

- Many efficient string algorithms:
  - ***very short:*** just a few lines of code
  - but hard to understand, hard to remember
  - lots of possibilities for ±1 mistakes, etc.
- **Today:** some examples of algorithms that are ***easy to remember & get right***

String algorithms

- **Right perspective:**
  - *character ≈ integer*
  - *string ≈ array*
- **Wrong perspective:**
  - string ≈ integer

Substring search

- Does **pattern *P*** occur in **text *T*** as substring?
  - find **“ram”** in **“programming”**?
- Trivial if *P* short: brute force
- Trivial if *T* “random”: brute force
  - mismatched detected very soon

Substring search

- Does **pattern *P*** occur in **text *T*** as substring?
- What if P and T are long and "adversarial"?
  - ***P* = "aaaaaaaa"**
  - ***T* = "aaaaaaaabaaaaaaaabaaaaaaa"**
- Naive brute force easily $O(n^2)$ for $|P| \approx |T| = n$

Rolling hash

Rolling hash

- Hash for all length-k substrings:
  - Input: **“programming”**, $k = 3$
  - Compute: hash(**“pro”**), hash(**“rog”**), hash(**“ogr”**), hash(**“gra”**), hash(**“ram”**), hash(**“amm”**), …
- Compare each of these with hash(**“ram”**) to find matches
- Trivial $O(kn)$, can be done in $O(n)$

Rolling hash: idea

- hash(**a,b,c,d**) = 1000**a** + 100**b** + 10**c** + 1**d**
  - assume you just calculated hash(**a,b,c,d**)
  - how could you easily calculate hash(**b,c,d,e**)?

Rolling hash: idea

- hash(**a,b,c,d**) = 1000**a** + 100**b** + 10**c** + 1**d**
  - assume you just calculated hash(**a,b,c,d**)
  - how could you easily calculate hash(**b,c,d,e**)?
- hash(**b,c,d,e**) = 1000**b** + 100**c** + 10**d** + 1**e**

Rolling hash: idea

- hash(**a,b,c,d**) = 1000**a** + 100**b** + 10**c** + 1**d**
  - assume you just calculated hash(**a,b,c,d**)
  - how could you easily calculate hash(**b,c,d,e**)?
- hash(**b,c,d,e**) = 1000**b** + 100**c** + 10**d** + 1**e**
  = 10 · (100**b** + 10**c** + 1**d**) + 1**e**

Rolling hash: idea

- hash(**a,b,c,d**) = 1000**a** + 100**b** + 10**c** + 1**d**
  - assume you just calculated hash(**a,b,c,d**)
  - how could you easily calculate hash(**b,c,d,e**)?
- hash(**b,c,d,e**) = 1000**b** + 100**c** + 10**d** + 1**e**
  = 10 · (100**b** + 10**c** + 1**d**) + 1**e**
  = 10 · hash(**a,b,c,d**) + 1**e** − 10000**a**

Rolling hash: idea

- hash(**a,b,c,d**) = 1000**a** + 100**b** + 10**c** + 1**d**
- hash(**b,c,d,e**) = 10 · hash(**a,b,c,d**) + 1**e** – 10000**a**
- Replace factor "10" with some other factor *X*
  - why? how to choose a good *X*?
- Calculate everything modulo *M*
  - why? how to choose a good *M*?

Rolling hash: uses

- Search for ***multiple patterns***
  - pre-compute hash(pattern), tabulate
- Find ***repeated substrings*** inside string
  - see if there are any collisions of hash(substring)
- Find ***common substrings*** of two strings
  - calculate & tabulate hash(substring) for both strings

Rolling hash: details

- One obstacle: **fixed *k***
- Multiple patterns:
  - group together patterns of ***similar sizes***
  - ***round down*** pattern length a bit
- Longest common substring:
  - ***binary search*** for the largest *k*

Z array

Z array

```
aabcaacaabac
 10021031010

abababab
 0604020

aaaa
 321
```

Z array

aabcaacaabac
 100210310 10

abababab
 0604020

aaaa
 321

Z array

aabcaacaabac
 10021031010

abababab
 0604020

aaaa
 321

Z array

- Simple "index structure"
- Easy to compute in $O(n)$ time
- Can be used for pattern matching:

```
ram#programming
 0000100300000
```

Compute Z array

aaaabaaabac

??????????

Compute Z array

aaaabaaabac

3

Compute Z array

aaaabaaabac
 32

Compute Z array

aaaabaaabac
 321

Compute Z array

```
aaaabaaabac
 3210
```

Compute Z array

aaaabaaabac
 32103

Compute Z array

aaaabaaabac
 321032

Compute Z array

aaaabaaabac
 3210321

Compute Z array

aaaabaaabac
 32103210

Compute Z array

aaaabaaabac
 32103210 1

Compute Z array

```
aaaabaaaabac
 321032101 0
```

Compute Z array

aabcaacaabac

??????????????

Compute Z array

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

1

Compute Z array

```
aabcaacaabac
 10
```

Compute Z array

```
aabcaacaabac
 100
```

Compute Z array

```
aabcaacaabac
  1002
```

Compute Z array

```
aabcaacaabac
 10021
```

Compute Z array

```
aabcaacaabac
 100210
```

Compute Z array

aabcaacaabac
 1002103

Compute Z array

```
aabcaacaabac
 10021031
```

Compute Z array

```
aabcaacaabac
 100210310
```

Compute Z array

aabcaacaabac
 1002103101

Compute Z array

aabcaacaabac
 10021031010

Suffix array

Suffix array

- All suffixes in alphabetical order

```
al
ational
ernational
international
ional
l
nal
national
nternational
onal
rnational
ternational
tional
```

Suffix array

- All suffixes in alphabetical order
- Easy to find e.g. ***repeating substrings***

```
al
ational
ernational
international
ional
l
nal
national
nternational
onal
rnational
ternational
tional
```

Suffix array

- $O(n)$ space
- Trivial algorithm: $O(n^2)$ time
- Easy algorithm: $O(n \text{ polylog } n)$ time

```
al
ational
ernational
international
ional
l
nal
national
nternational
onal
rnational
ternational
tional
```

Pairs		Sort		Renumber		Sort back
i 1	→ in 1	→ al 12	→	1 al 12	→	4 in 1
n 2	nt 2	at 7		2 at 7		8 nt 2
t 3	te 3	er 4		3 er 4		11 te 3
e 4	er 4	in 1		4 in 1		3 er 4
r 5	rn 5	io 9		5 io 9		10 rn 5
n 6	na 6	l_ 13		6 l_ 13		7 na 6
a 7	at 7	na 11		7 na 11		2 at 7
t 8	ti 8	na 6		7 na 6		12 ti 8
i 9	io 9	nt 2		8 nt 2		5 io 9
o 10	on 10	on 10		9 on 10		9 on 10
n 11	na 11	rn 5		10 rn 5		7 na 11
a 12	al 12	te 3		11 te 3		1 al 12
l 13	l_ 13	ti 8		12 ti 8		6 l_ 13

Pairs → **Sort** → **Renumber, sort back, repeat …**

4	in	1		4,11	inte	1		1, _	al__ 12
8	nt	2		8, 3	nter	2		2, 5	atio 7
11	te	3		11,10	tern	3		3, 7	erna 4
3	er	4		3, 7	erna	4		4,11	inte 1
10	rn	5		10, 2	rnat	5		5, 7	iona 9
7	na	6		7,12	nati	6		6, _	l___ 13
2	at	7		2, 5	atio	7		7, 6	nal_ 11
12	ti	8		12, 9	tion	8		7,12	nati 6
5	io	9		5, 7	iona	9		8, 3	nter 2
9	on	10		9, 1	onal	10		9, 1	onal 10
7	na	11		7, 6	nal_	11		10, 2	rnat 5
1	al	12		1, _	al__	12		11,10	tern 3
6	l_	13		6, _	l___	13		12, 9	tion 8

suffixes of length 4 ordered

Summary

- **Z array:** finding overlap between prefix and internal parts

  needleabcdefneedleghijklmnopqr

- **Suffix array:** finding overlap between internal parts

  abcdefneedleghijklneedlemnopqr

In practice

- Good ideas:
  - ***hashing***
  - ***sorting***
  - ***arrays***
- Bad ideas:
  - search tree data structures