String Algorithms

HKOI 2018 training

Anson Ho

Content

- introduction
- trie
- KMP
- hashing
- suffix array
- sth else…

What is String?

a string is traditionally a sequence of characters (Wiki)

or more familiarly

- char s[SIZE];
- "Hello World!"
- {'a','b','c'}

ASCII code

- **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
- 8 bits = 1 byte = sizeof(char)
- 256 different states (0~255)
- '0' = 48 '9' = 48+9 = 57
- 'A' = 65 'Z' = 65+26-1 = 90
- 'a' = 97 'z' = 97+26-1 = 122
- ' ' = 32 '+' = 43 '-' = 45
- null= 0 '\n'= 10 '\r'= 13

Glossary

concatenation

- "addition" in string

- 1+10 = 11
- "1"+"10" = "110"

- C++: "1""10"
- Haskell: "1"++"10"

Glossary

lexicographical order

- comparison in string
- dictionary order
- alphabetical order
- numerical order of ASCII code

- "1" < "2" < "A" < "B" < "a" < "b"

Glossary

lexicographical order

- same lengths -> numerical order
- "123" < "132"
- "ab" < "ac"

- different lengths -> add zeros
- "a" < "ab" {97,0} < {97,98}
- "ab" < "z" {97,98} < {122,0}

Glossary

prefix

- bicycle
- triangle
- hyperlink

suffix

- girls
- beautiful

Glossary

substring

- prefix of suffix, suffix of prefix
- contiguous subarray

subsequence

- obtained by deleting characters
- order is kept

Glossary

"Alice and Bob"

substring

- "", "ce a", "Alice and Bob"

subsequence

- all of the above
- "A B", "AaBb"

Glossary

palindrome

- same as the reversed string

- "a", ""
- "radar", "abccba"

- longest palindromic subsequence
- O(N^2) with DP

String vs Sequence of numbers

- String
  - fixed size of alphabet (set of input symbols)
    - sometimes appears in time/memory complexity
  - order of symbols is less emphasized
    - except lexicographical order
  - mostly similar in other aspects
    - some string tasks can be solved by algorithms for sequence, vice versa

I/O

- Input and Output

- How to read a string from stdin?
- (in C++)

I/O

scanf printf

- read until space, line break or EOF
- '\0' is automatically added

- int n;
- char x[5]; // max 4 char.
- scanf("%d%s", &n, x);
- printf("%s", x);

I/O

gets puts

- read until line break or EOF
- '\0' is automatically added
- input will not include '\n'
- output with '\n' at the end

- ```
  char x[5];
  gets(x);
  puts(x);
  ```
  ```

I/O

gets

- deprecated, or even removed, in new C++ version

- replacement:

```
fgets(x, sizeof(x), stdin);
```

I/O

cin cout

- SLOW
- sync_with_stdio(false)

getchar putchar

- single character
- = scanf printf with %c
- unlocked

I/O

sscanf sprintf

- read from a string
- write to a string

- int n, m;
- char x[12];
- sprintf(x, "%d%d", n, m);
- sscanf(x, "%d", &n);

Container

char[]
- static size
- easy for I/O
- null terminator ('\0’)
  - remember to reserve one space for it

std::vector<char>
- dynamic size
- built-in compare (lexicographical order)
- compatible with other STL tools
  - sort vector< vector<char> >
  - map, set

Container

<string>

- C++ object
- supports +, ==, <, <=, …

- char x[5];
- string str(x);
- str[0]=str[1];
- printf("%s\n",str.c_str());

<cstring>

- memcpy strcpy
- memcmp strcmp
- memset
- strlen

- don't use if you forget how to use
- same time complexity as brute force
- maybe faster in terms of constant

See also

- AM Session (intermediate)
  - Data Processing
    - Data manipulation
      - String functions
- <cstring>
  - http://www.cplusplus.com/reference/cstring/
- <string>
  - http://www.cplusplus.com/reference/string/

Trie

- not a formal word
- (not found in Oxford Dictionary)

- a tree
- storing multiple strings

- searching/counting strings/prefixes

Trie

- C++ implementation

- ```
  struct node {
      int count, child[26];
      node *child[26];
  };
  std::vector<node> trie;
  ```
  ```

Trie

- each node represents a character
  - except the root
- stores the positions/pointers of its children which are the next characters
- stores the count of strings ending at the node or in its subtree

Trie

N: total length

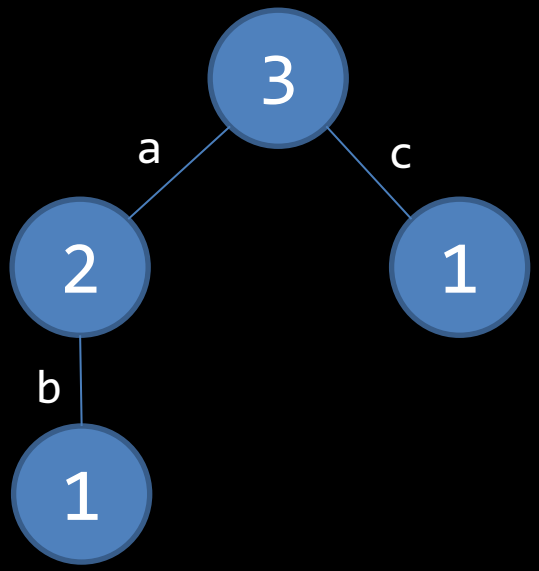
M: length of the target string

- time complexity:
  - insertion: $O(M)$
  - deletion: $O(M)$
  - searching/counting: $O(M)$
  - sorting: $O(N)$
- memory complexity: $O(26N)$

actually all of them are DFS

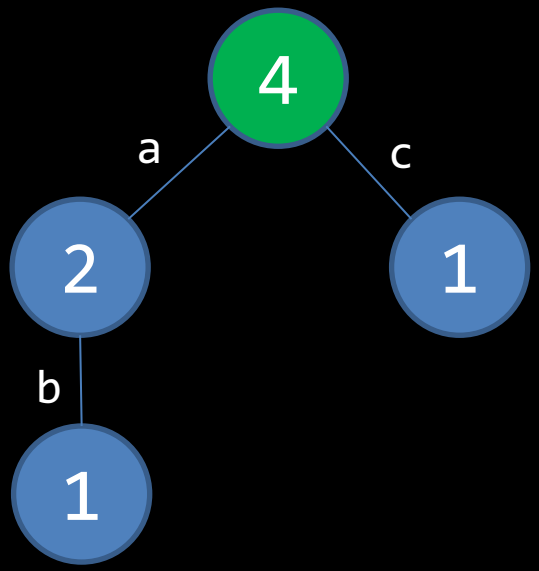
Trie

insert "abc" to {"a","ab","c"}



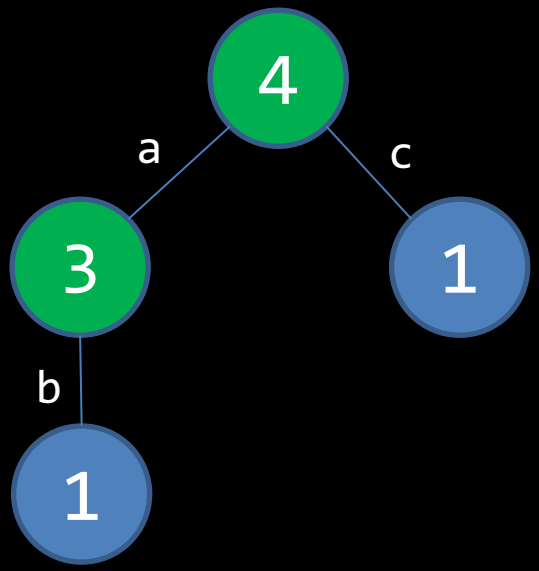
Trie

insert "abc" to {"a","ab","c"}



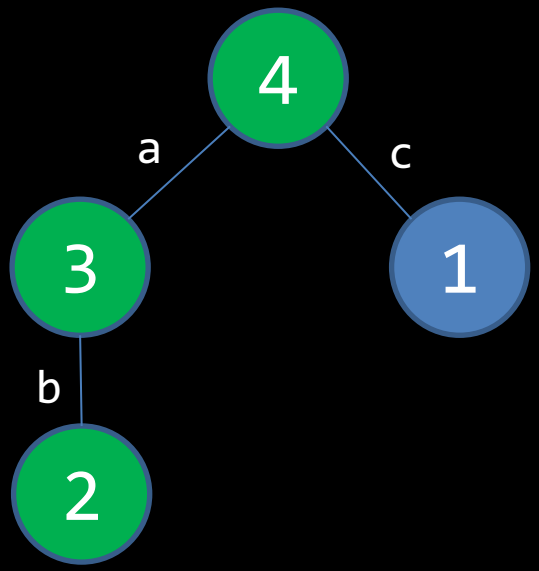
Trie

insert "abc" to {"a","ab","c"}



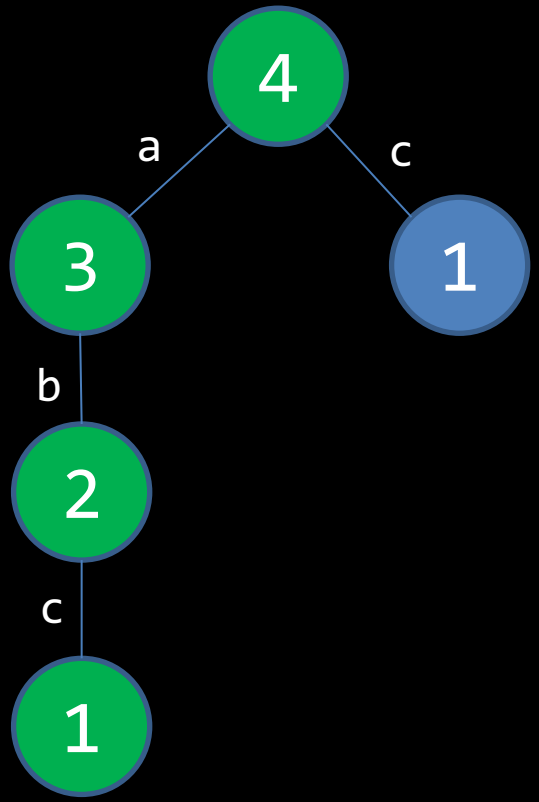
Trie

insert "abc" to {"a","ab","c"}



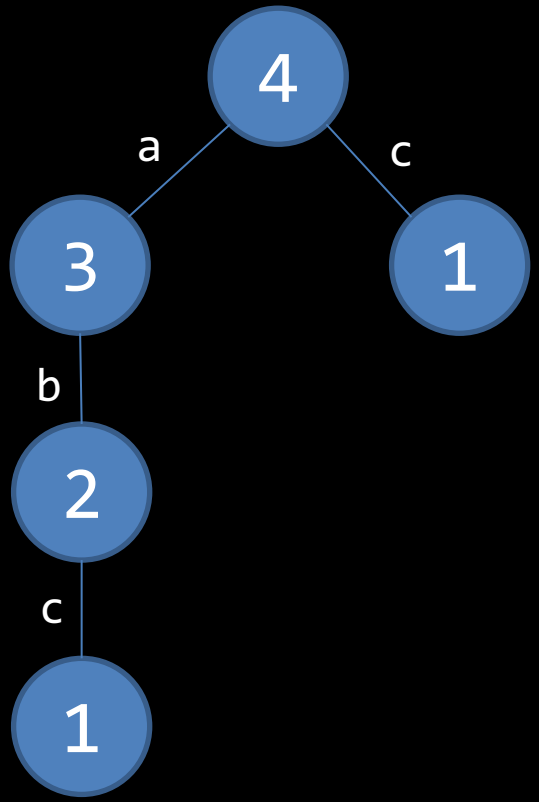
Trie

insert "abc" to {"a","ab","c"}



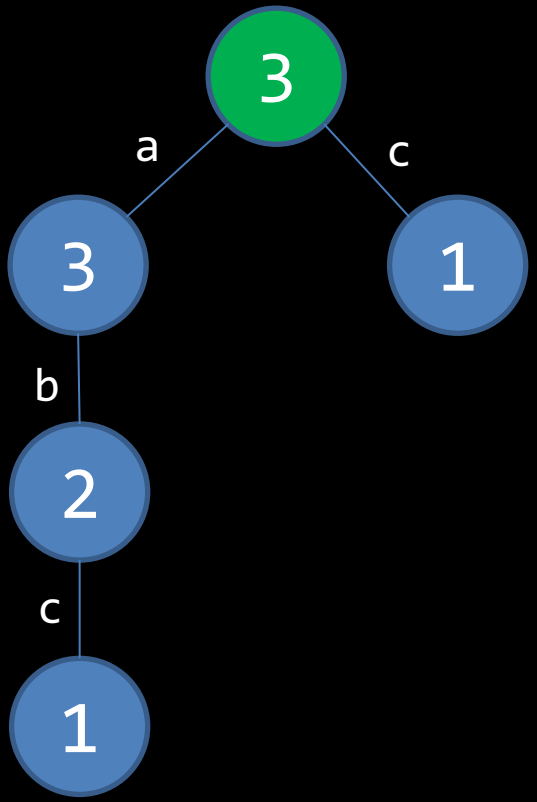
Trie

delete "ab" from {"a","ab","abc","c"}



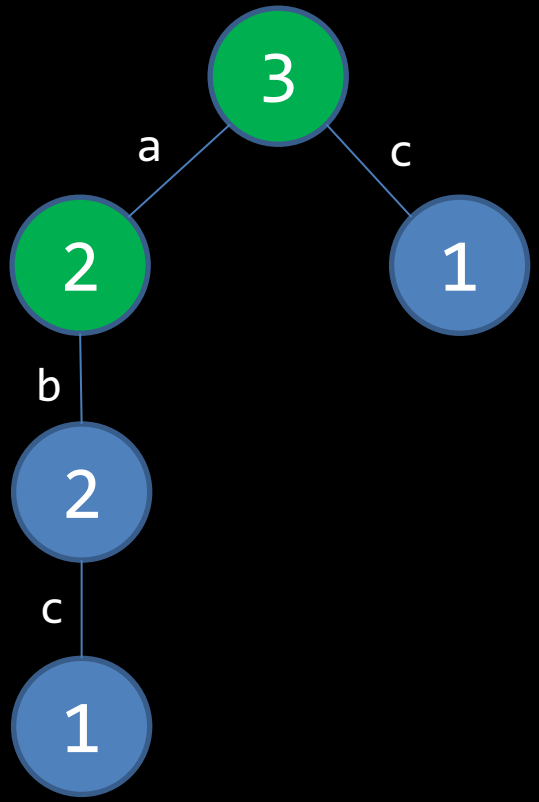
Trie

delete "ab" from {"a","ab","abc","c"}



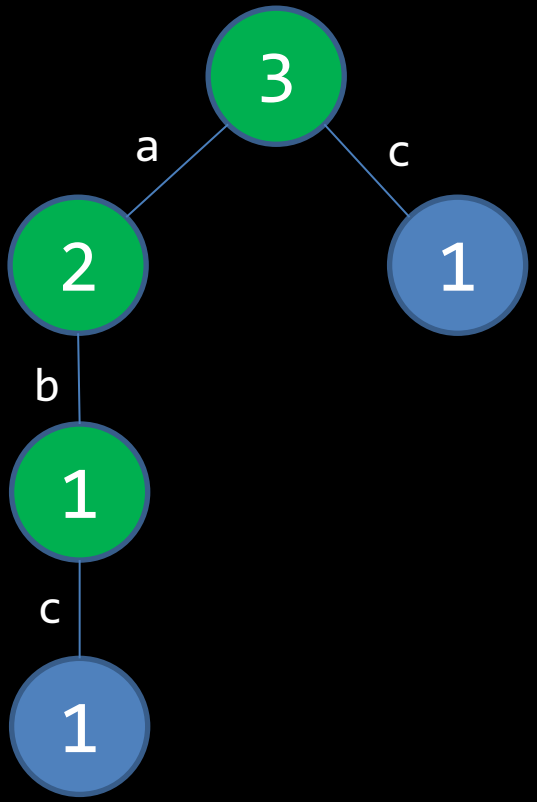
Trie

delete "ab" from {"a","ab","abc","c"}



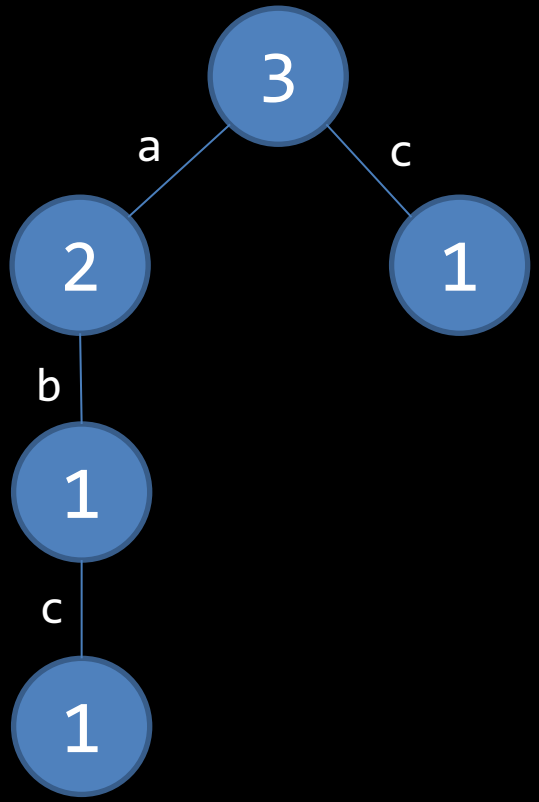
Trie

delete "ab" from {"a","ab","abc","c"}



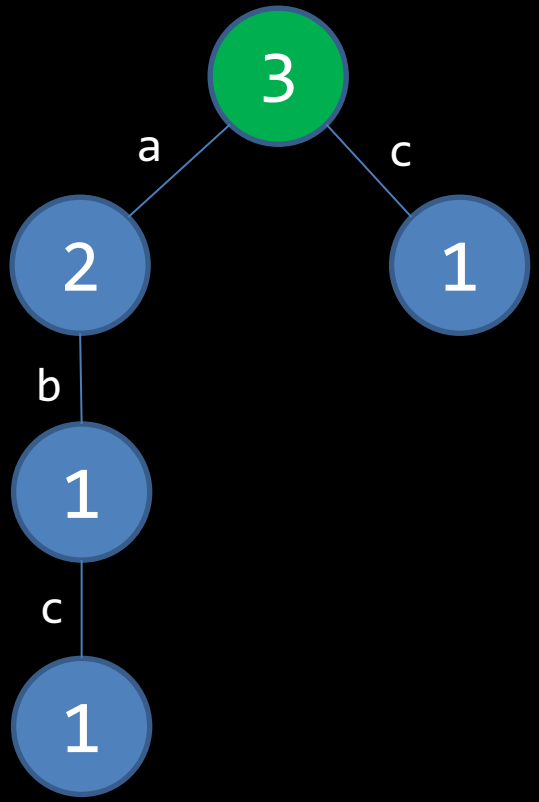
Trie

count string with the prefix "a"



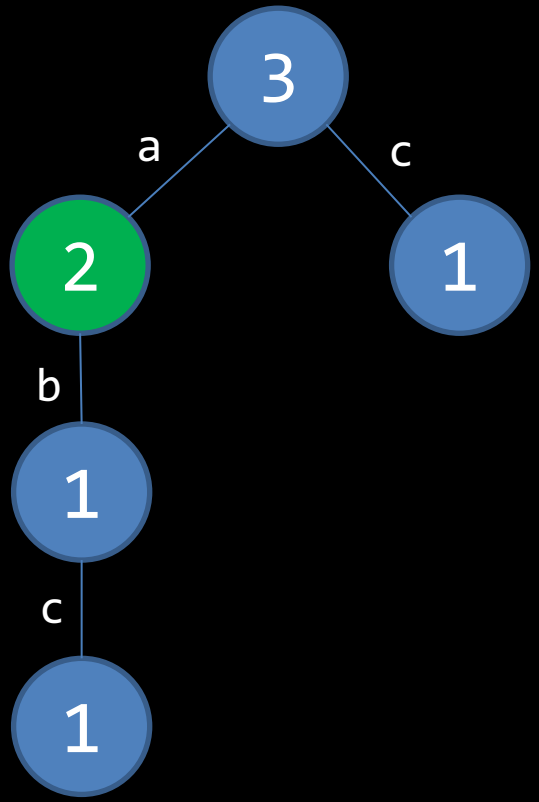
Trie

count string with the prefix "a"



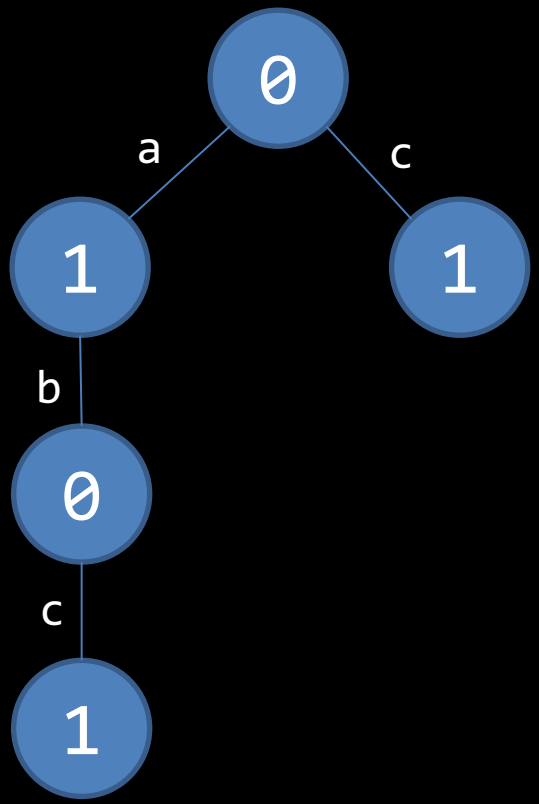
Trie

count string with the prefix "a"



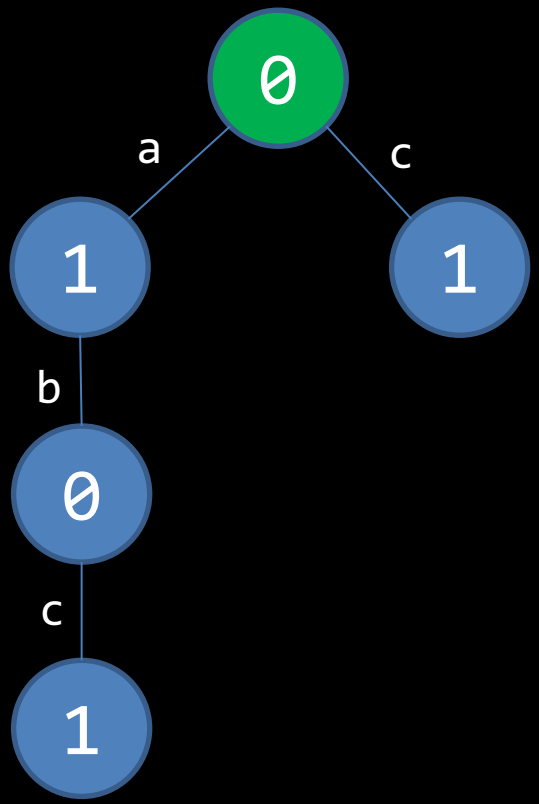
Trie

count string "a"



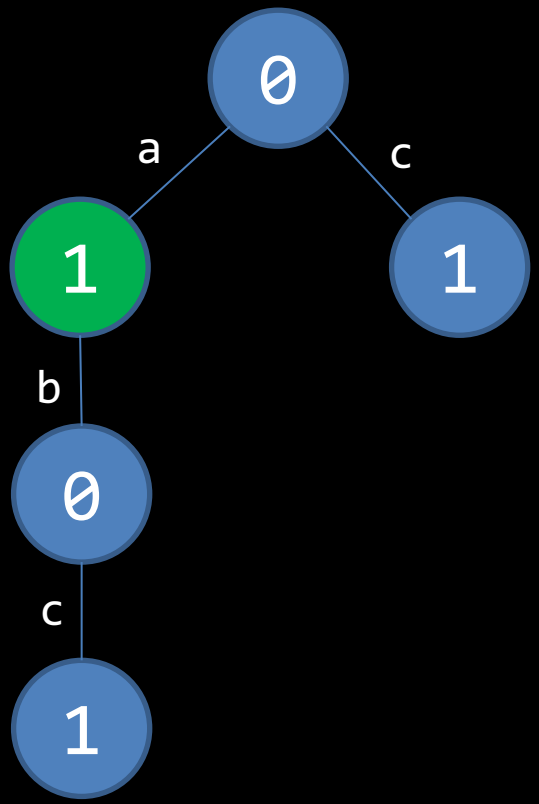
Trie

count string "a"



Trie

count string "a"



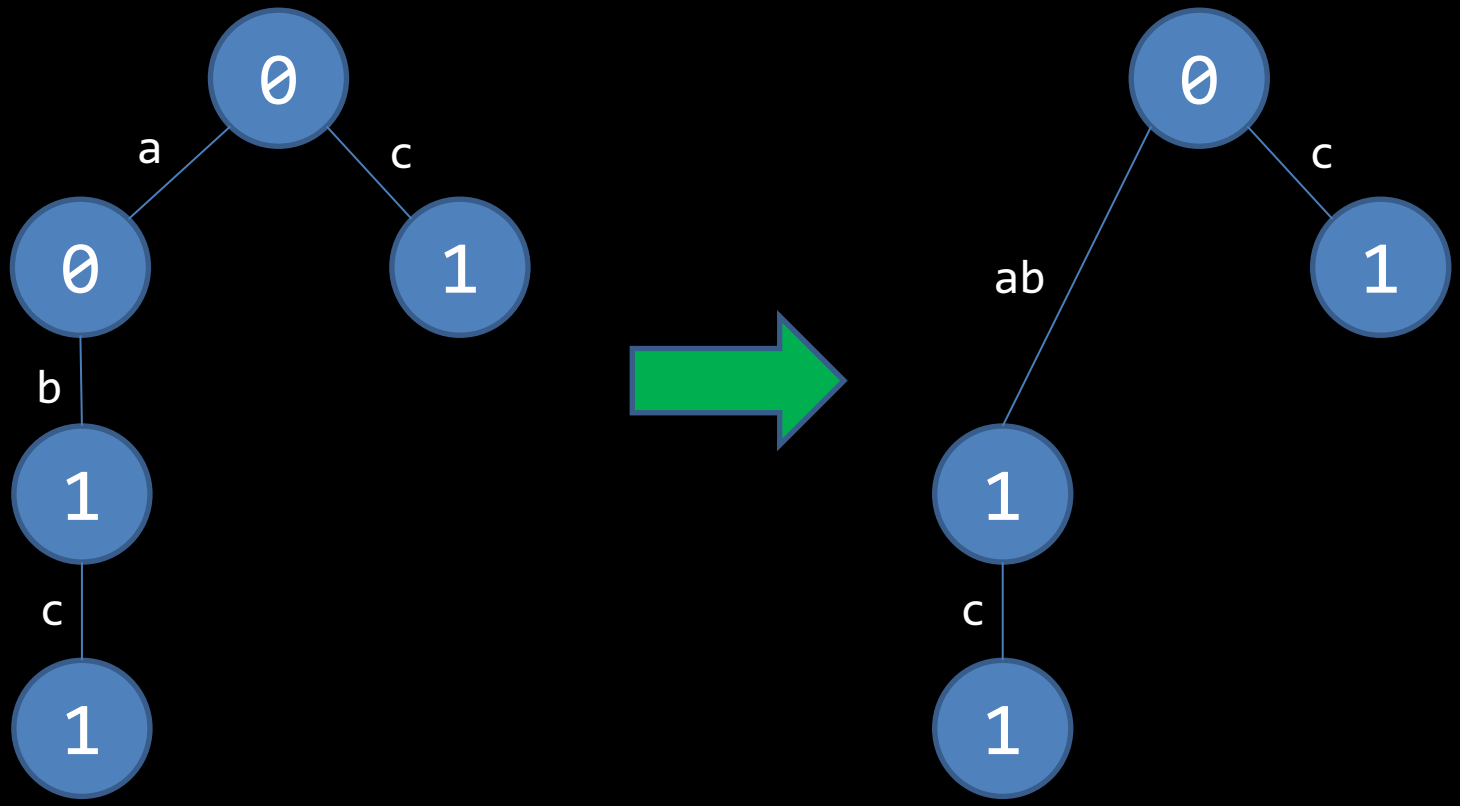
Trie

radix tree/compressed trie

- a variant (or an optimization)
- some useless nodes in trie are merged with its parent
- harder implementation

Trie

radix tree/compressed trie



Trie

- sometimes can be used as a replacement/variant of segment tree
  - by treating numbers as their binary/decimal representations
- variant: persistent trie tree
  - analog of persistent segment tree

KMP

HKOJ 01002 A Counting Problem

- given 2 strings x, y
- length <= 1000
- count the occurrences of y in x

KMP

HKOJ 01002 A Counting Problem

```
for each starting position in x
    if the |y| characters follow
    exactly match with y
        ans++
```

- O(*NM*)

KMP

~~HKOJ 01002 A Counting Problem~~

- given 2 strings x, y
- length <= ~~1000~~ 10^5
- count the occurrences of y in x

- a more efficient algorithm is needed

KMP

- KMP is a solution
  - Knuth-Morris-Pratt Algorithm
- O(N + M)
- How it works?

KMP

brute force

AHKOIHHKOIHKABC
HKOIHK

KMP

brute force

AHKOIHHKOIHKABC
HKOIHK

KMP

brute force

AHKOIHHKOIHKABC
 HKOIHK

KMP

brute force

AHKOIHHKOIHKABC
 HKOIHK

KMP

brute force

AHKOIHHKOIHKABC
 HKOIHK

KMP

brute force

AHKOIHHKOIHKABC
 HKOIHK

KMP

brute force

AHKOIHHKOIHKABC
 HKOIHK

KMP

brute force

AHKOIHHKOIHKABC
 HKOIHK

KMP

brute force

AHKOIHHKOIHKABC
HKOIHK

KMP

brute force

AHKOIHHKOIHKABC
HKOIHK

KMP

brute force

AHKOIHHKOIHKABC
HKOIHK

KMP

optimization 1

AHKOIHHKOIHKABC
 HKOIHK

KMP

optimization 1

```
-HKOIHH--------
 HKOIHK
```

KMP

optimization 1

```
-HKOIHH--------
  HKOIHK
```

KMP

optimization 1

-HKOIHH--------
HKOIHK

KMP

optimization 1

```
-HKOIHH--------
    HKOIHK
```

KMP

optimization 1

```
-HKOIHH--------
    HKOIHK
```

KMP

optimization 1

HKOIHK -> move 4 units
HKOIHK -> move 4 units

HKOIHK -> move 1 unit
HKOIHK -> move 2 units

HKOIHK -> move 1 unit

KMP

optimization 1

HKOIHK -> move 4 units
HKOIHK -> move 4 units

HKOIHK -> move 1 unit
HKOIHK -> move 2 units

HKOIHK -> move 1 unit

KMP

optimization 1

HKOIHK -> move 4 units

length of move = length of green string – length of underline / 2

KMP

optimization 1

- precompute an array of length of underline in O(M)
- simple dynamic programming

KMP

optimization 1

O(number of operations in the searching stage) =

O(number of increments of the green part + number of decrements of the green part) = O(N)

overall time complexity = O(N + M)

KMP

optimization 2

HKOIHK -> move 4 units

length of move = length of green string – length of underline / 2

KMP

optimization 2

```
---HKOI#-------
   HKOIHK
```

KMP

optimization 2

```
---HKOI#-------
   HKOIHK
```

KMP

optimization 2

```
---HKOI#-------
   1HKOIHK
```

KMP

optimization 2

```
---HKOI#-------
   12HKOIHK
```

KMP

optimization 2

```
---HKOI#-------
   123HKOIHK
```

KMP

optimization 2

```
---HKOI#-------
   1234HKOIHK
```

KMP

optimization 2

HKOI***H***K -> move 4+1 units

HKOIHKOI -> move 4+4 units

HKOIHHKI -> move 5-1 units

H***K***OIH***K***KI -> move 5-1+1 units

KMP

optimization 2

HKOIHKKI -> move 5-1+1 units

length of move = length of green string – length of underline / 2 + length of italic / 2

KMP

optimization 2

- by Knuth, the K in KMP
- no improvement on time complexity
- thus sometimes being ignored

KMP

Remarks

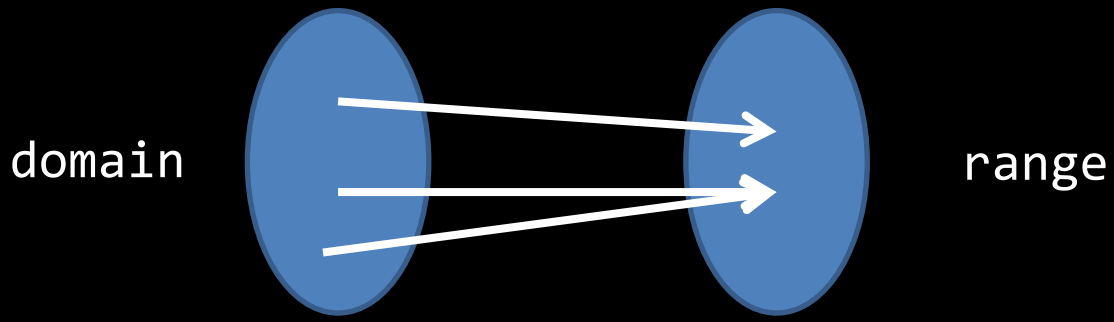
- KMP works with one target and multiple input
- e.g.: finding w from x, y, z

Take a break

Hashing

- hash table
  - taught in Data Structures (II)

- string hashing: String->Number
  - a function
  - not necessarily injective



Hashing

- advantages
  - can be stored in array
  - can be compared/searched in O(1)

- disadvantages
  - inverse may not be easily found
  - not injective -> may crash
    - e.g. output extra results in string matching

Hashing

- rolling hash
  - "sliding window" fast (in O(1))
- Rabin-Karp hash
  - common rolling hash
  - addition
    - can be replaced by xor
  - multiplication
  - mod

Hashing

Rabin-Karp hash

$$H = (c_0 a^{n-1} + c_1 a^{n-2} + \ldots + c_{n-2} a + c_{n-1}) \bmod p$$

- H = hash value
- c_i = ASCII code / A=0, B=1, …, Z=25
- a = 256 / 26
- p = a large prime, e.g. 1e9+7

Hashing

"HKOI"

$c = 7, 10, 14, 8$

$a = 26$

$p = 64997$

$H = (7*26^3+10*26^2+14*26+8) \bmod 64997$

$= 130164 \bmod 64997$

$= 170$

Hashing

"GO"

c = 6, 14

a = 26

p = 64997

H = (6*26+14) mod 64997

= 170 mod 64997

= 170

Hashing

H("HKOI") = 170 = H("GO")

"HKOI" == "GO" ???

Hashing

- assume the domain is all strings
- surjective function
- all output values with equal probability

If the input has 2 strings,

P(crash) = 1/64997 = 0.0000154

- negligible?

Hashing

If the input has 700 strings,

P(crash)

$= 1-(64997*64996*\ldots*64298)/64997^{699}$

$= 0.977$

- NOT negligible

Hashing

solution 1

- give up hashing
- use exact algorithm
- e.g. KMP for string matching

Hashing

solution 2a

- use a larger prime

p = 1e9+7
If the input has 700 strings,
P(crash) = 0.00024

Hashing

solution 2b

- use primes
- compute *H* more than once with different primes
- match <-> all *H*s match
- P(crash) decreases exponentially

Hashing

solution 2

- cannot be stored in array
- use map/set instead
- increase time complexity by logN

- not always need to store the result

Hashing

"slide"

- use previous result to speed up

$$H = (c_0 a^{n-1} + c_1 a^{n-2} + \ldots + c_{n-2} a + c_{n-1}) mod\ p$$

$$H' = (c_1 a^{n-1} + c_2 a^{n-2} + \ldots + c_{n-1} a + c_n) mod\ p$$

$$H' = ((H - c_0 a^{n-1}) \times a + c_n) mod\ p$$

$$H = ((H' - c_n) \times a^{-1} + c_0 a^{n-1}) mod\ p$$

Hashing

"slide"

- use long long (in C++) if there is possibility of overflow
- subtraction
  - avoid negative numbers

  $(a-b) \bmod p \rightarrow (a+p-b\%p)\%p$

Hashing

"slide"

- modular inverse
  - taught in Mathematics in OI (I)
  - extended Euclidean algorithm
  - Fermat's little theorem
    - p needs to be a prime first
  - power mod
    - taught in Recursion, Divide and Conquer

$a^{-1} \bmod p \rightarrow a^{p-2} \bmod p$

Hashing

hash of substring in O(1)

- precompute a partial sum array
  - taught in Optimization
- multiply a power of a accordingly

See also

- PM Session
  - Randomized Algorithm
    - The Classics
      - Hashing

- std::hash
  - http://www.cplusplus.com/reference/functional/hash/
  - used in std::unordered_map and std::unordered_set

Take a break

Suffix array

- Indices of all non-empty suffixes
- sorted

- "IOIHKGHKOI"

- "HKGHKOI"
- "HKOI"
- ...

Suffix array

- IOIHKGHKOI
- OIHKGHKOI
- IHKGHKOI
- HKGHKOI
- KGHKOI
- GHKOI
- HKOI
- KOI
- OI
- I

- 5
- 3
- 6
- 9
- 2
- 0
- 4
- 7
- 8
- 1

Suffix array

$O(N^2 \log N)$

- brute force

- user-defined compare function
- or std::sort
  std::vector< pair<vector<char>,int> >

Suffix array

$O(N \log N \log N)$

- compare suffixes by the length of the power of two

- no. of sort: $O(\log N)$
- no. of comparisons in each sort: $O(N \log N)$

Suffix array

$O(N \log N \log N)$

len=1,2,4,8...

rank[i]=the rank of the first len characters of the ith suffix

Suffix array

- IOIHKGHKOI
- OIHKGHKOI
- IHKGHKOI
- HKGHKOI
- KGHKOI
- GHKOI
- HKOI
- KOI
- OI
- I

Suffix array

- IO
- OI
- IH
- HK
- KG
- GH
- HK
- KO
- OI
- I_

- len=2

Suffix array

- IO
- OI
- IH
- HK
- KG
- GH
- HK
- KO
- OI
- I_

- 4
- 7
- 3
- 1
- 5
- 0
- 1
- 6
- 7
- 2

Suffix array

$O(N \log N \log N)$

len = 1

- trivial

Suffix array

$O(N \log N \log N)$

len > 1

- rank[] is describing the status of len/2 till the end of the sorting of len

Suffix array

$O(N \log N \log N)$

len > 1

compare(a,b) //overriding '<'

if r[a]!=r[b] return r[a]<r[b]

return r[a+len/2]<r[b+len/2]

- r[i]=-1 if i>=n

Suffix array

$O(N \log N \log N)$

len > 1

- update rank[] at the end of sorting
- handle the case of equality

  r[a]==r[b] and r[a+len/2]==r[b+len/2]

Suffix array

$O(N \log N \log N)$

len >= N

- the whole process is done after this sorting

Suffix array

faster than O($N \log N \log N$)

- it exists
- http://gagguy.blogspot.hk/2012/08/linear-time-suffix-array-dc3.html
- harder to understand/implement
- not so important unless you aim for NOI

Suffix array

longest common prefix lcp[]

- useful in many application of suffix array
  - with the aid of segment tree (RMQ)
- brute force needs $O(N^2)$
- actually $O(N)$ is enough

Suffix array

longest common prefix lcp[]

```
i   : ABC...
i+1 : BCD...
lcp[i] = 0

j   : ABC...
j+1 : ABD...
lcp[j] = 2
```

Suffix array

longest common prefix lcp[]

```
j   : ABC...
j+1 : ABD...
lcp[j] = 2

k   : BC...
k+1 : BD...
lcp[k] >= lcp[j]-1
```

Suffix array

longest common prefix lcp[]

- the 2 new suffixes obtained by deleting the first character may not be neighbors
- the property lcp[k] >= lcp[j]-1 still holds
- can be computed in $O(N)$ by the decreasing order in the lengths of the suffixes

Suffix array

string matching

- binary search
- find the suffix beginning with the target
- i.e. target is the prefix of it
- O($T \log N$)
- can be improved by lcp[]

Suffix array

other application

- longest repeated substring
- longest common substring
- longest palindromic substring

Other string algorithms

- Z algorithm
  - string matching in O(*N*)
- Manacher algorithm
  - longest palindromic substring in O(*N*)

Other string algorithms

- suffix tree
  - a tree storing all the suffixes of a string
  - compute suffix array in O(*N*)

- **A**ho-**C**orasick Algorithm
  - multi-target version of KMP
  - linear time complexity

Other string algorithms

- palindromic tree
  - solve some types of queries about palindrome efficiently

Ad hoc

- most string tasks can be solved with some well-known algorithm in your template (weapon)

- some are not
  - construct a k-length 01-string that minimizes the maximum length of palindromic substring (k >= 9)
    - hints: the maximum length is always 4

Practice

judge.hkoi.org/tag/strings
codeforces.com/problemset/tags/strings

- HKOJ 01002 A Counting Problem
- HKOJ M0932 String Rotation

- UOJ 35 後綴排序
- NOI 2015 品酒大會

Reference

- 演算法筆記
  - www.csie.ntnu.edu.tw/~u91029/

The end

remember to participate in
Mini-Competition (IV)
at 2pm