P THINK FAST S

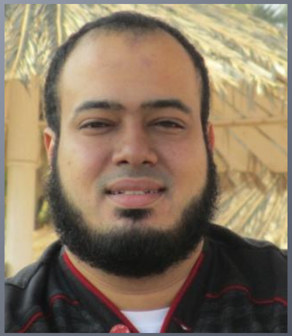
Competitive Programming

From Problem 2 Solution in O(1)

String Processing Algorithms

Suffix Arrays - O(nlognlogn)

Mostafa Saad Ibrahim
PhD Student @ Simon Fraser University



Prefix and Suffix of a string

Let the string is $abbab$

Prefixes	Suffixes
λ	$abbab$
a	$bbab$
ab	bab
abb	ab
$abba$	b
$abbab$	λ

$$w = uv$$

prefix

suffix

- For our purpose, we will consider empty string
- So, for length **n = 5**, we generate **6 suffixes**
- Remember: 6 suffixes of different lengths

Src: http://images.slideplayer.com/25/7605252/slides/slide_11.jpg

Generate suffixes and sort

Let S = **abracadabra** (length = 11)

1- **Generate** 12 suffixes
2- **Sort** based on string (alphabetically)
3- The new **indices** ordering is called **suffix array**

```
int suffix_array[] = {
10, 7, 0, 3, 5, 8, 1, 4, 6, 9, 2, 11
};
```

0	abracadabra
1	bracadabra
2	racadabra
3	acadabra
4	cadabra
5	adabra
6	dabra
7	abra
8	bra
9	ra
10	a
11	

11	
10	a
7	abra
0	abracadabra
3	acadabra
5	adabra
8	bra
1	bracadabra
4	cadabra
6	dabra
9	ra
2	racadabra

Suffix Arrays

- A suffix array is a sorted array of all suffixes of a string.
- Given that it considers every position in the string, it can be used in several string processing tasks such as queries on all available substrings or pattern search

Brute Force Approach

- Generate the suffixes
- Sort them
  - nlogn for sorting algorithm
  - comparing 2 strings is O(n)
  - total O(n^2 logn) = So slow
- Code
  - Generate N suffixes put in vector
  - Create map from suffix to its original index
  - Sort the vector
  - Now we can use map to know idx of ith sorted suffix

Brute Force Approach

```
void buildSuffixArraySlow(string str) {
  map<string, int> suffix_idx_map;
  vector<string> suffixes;

  for (int i = 0; i <= (int) str.size(); i++) {
    string suffix = str.substr(i, str.size() - i);
    suffix_idx_map[suffix] = i;
    suffixes.push_back(suffix);
  }
  sort(suffixes.begin(), suffixes.end());
  for (int i = 0; i < (int) suffixes.size(); i++)
    cout << suffixes[i] << "\t" << suffix_idx_map[suffixes[i]] << "\n";
}
```

Faster approaches

- **Main observation:**
  - They are suffixes of ONE string, not random strings
  - How to use this fact to build efficient solutions?
- **O(Nlogn^2) solution**
- **O(Nlogn) solution improvement**
- Algorithms based on Suffix tree
- O(N) algorithms (e.g. SA-IS algorithm)
- I will cover the first 2
  - It is not easy topic, but not so hard (especially 2nd one)
  - tracing examples & debugging code = Full understanding

Incremental Sortings

- Assume suffixes are sorted based on the first 2 letters
- Can we sort it efficiently based on 4 letters?
- Then, sort it based on 8 letters?
- Then sort it based on 16 letters?
- And so on?
- This O(logn) steps * ordering first **h letters**

Sorted suffixes on first 2 letters

Suffix (2)	Index	Group
	11	0
a	10	1
abracadabra	0	2
abra	7	2
acadabra	3	3
adabra	5	4
bracadabra	1	5
bra	8	5
cadabra	4	6
dabra	6	7
racadabra	2	8
ra	9	8

- A **group** is a new array that group equal same length prefixes
- E.g. index 0 and 7 starts with ab
- So both assigned same group = 2

- This can be trivially computed
- Your group = previous group + 1 if different prefixes <u>*at first h=2 letters*</u>
-
- E.g. group(**ac**adabra) = group(**ab**ra) + (ac != ab) = 2 + 1 = 3

Sorting 4 letters from 2 letters

Suffix (2)	Index	Group
	11	0
a	10	1
abracadabra	0	2
abra	7	2
acadabra	3	3
adabra	5	4
bracadabra	1	5
bra	8	5
cadabra	4	6
dabra	6	7
racadabra	2	8
ra	9	8

Compare(abra, bra)

- g(abra) = 2, group(bra) = 5
- Actually on 2 letters, they are different
- So in new ordering abra < bra (4 letters)

Compare(bracadabra, bra)

- g(abra) = 5, group(bra) = 5
- Same group (= first 2 letters)
- We need to compare **next 2 letters**
- How to do that fast?
- Remember next 2 letters are suffixes

Sorting 4 letters from 2 letters

Suffix (2)	Index	Group
	11	0
a	10	1
abracadabra	0	2
abra	7	2
acadabra	3	3
adabra	5	4
bracadabra	1	5
bra	8	5
cadabra	4	6
dabra	6	7
racadabra	2	8
ra	9	8

br**ac**adabra

- We need to ignore first 2 letters
- **ac**adabra, find its group
- group(**ac**adabra) = 3

br**a**

- We need to ignore first 2 letters
- group(**(ra)**) = 9

Then Compare(bracadabra, bra)

- compare 3 vs 9
- 3 first => bracadabra < bra

Sorting 4 letters from 2 letters

Suffix (2)	Index	Group
	11	0
a	10	1
abracadabra	0	2
abra	7	2
acadabra	3	3
adabra	5	4
bracadabra	1	5
bra	8	5
cadabra	4	6
dabra	6	7
racadabra	2	8
ra	9	8

br**ac**adabra

- We need to ignore first 2 letters
- **ac**adabra, find its group
- group(**ac**adabra) = 3

How to get the **ac**adabra efficiently?

- index(br**ac**adabra) = 1
- index(r**ac**adabra) = 2
- index(**ac**adabra) = 3 (1 + h = 2)
- index(_**c**adabra) = 4 .. and so on

Then group[idx + h] is h shift from group[idx]

From 2 => 4 => 8 first letters

Suffix (2)	Index	Group		Suffix (4)	Index	Group		Suffix (8)	Index	Group
	11	0			11	0			11	0
a	10	1		**a**	10	1		**a**	10	1
abracadabra	0	2		**abra**cadabra	0	2		**abra**	7	2
abra	7	2		**abra**	7	2		**abracada**bra	0	3
acadabra	3	3		**acad**abra	3	3		**acadabra**	3	4
adabra	5	4		**adab**ra	5	4		**adabra**	5	5
bracadabra	1	5		**bra**	8	5		**bra**	8	6
bra	8	5		**brac**adabra	1	6		**bracadab**ra	1	7
cadabra	4	6		**cada**bra	4	7		**cadabra**	4	8
dabra	6	7		**dabr**a	6	8		**dabra**	6	9
racadabra	2	8		**ra**	9	9		**ra**	9	10
ra	9	8		**raca**dabra	2	10		**racadabr**a	2	11

Observe: Sorted suffix never go up. Either same **position** or lower. Same for its **group**
Observe: At h = 8, every suffix has a **different group**. We can stop processing.

Overall

- Initialization
  - At 1st iteration (h = 1), we need to sort on first letter
  - Then we should depend on ascii letter
  - Create length+1 suffixes
  - Assign group of suffix = ascii of first letter
  - Sort in O(nlogn)
- Process for h = {1, 2, 4, 8, 16…}
  - Sort **2h letters** based on h letters => O(nlogn)
  - Comparing now is 2 checkings on the group index only
- Order: O(logn) * O(nlogn)

Data Structures

```
const int MAXLENGTH = 10 * 0000;

char str[MAXLENGTH + 1];      //the string we are building its suffix array
int suf[MAXLENGTH + 1];       //the sorted array of suffix indices
int group[MAXLENGTH + 1];     //In ith iteration: what is the group of the suffix index
int sorGroup[MAXLENGTH + 1];  //temp array to build grouping of ith iteration

struct comp  //compare to suffixes on the first 2h chars
{
  int h;
  comp(int h) : h(h) {}

  bool operator()(int i, int j) {
    if (group[i] != group[j])     // previous h-groups are different
      return group[i] < group[j];
    return group[i + h] < group[j + h];
  }
};
```

Algorithm 2: Snapets

- Assume We sorted based on h letters
  - Sort based on h letters using 2h values
  - Linearly generate the new groups (first group id = 0)
  - Let n = # suffixes

```
sort(suf, suf + n, comp(h));  //sort the array using the first 2h chars

for (int i = 1; i < n; i++)  //compute the 2h group data given h group data
  sorGroup[i] = sorGroup[i - 1] + comp(h)(suf[i - 1], suf[i]);
```

  - Now, we need to reassign the groups of suffixes

```
for (int i = 0; i < n; i++)  //copy the computed groups to the group array
  group[suf[i]] = sorGroup[i];
```

Algorithm 2

```
void buildSuffixArray() {
  int n;  //number of suffixes = 1+strlen(str)
  //Initially assume that the group index is the ASCII
  for (n = 0; n - 1 < 0 || str[n - 1]; n++)
    suf[n] = n, group[n] = str[n];  //code of the first char in the suffix

  sort(suf, suf + n, comp(0));  //sort the array the suf on the first char only
  sorGroup[0] = sorGroup[n-1] = 0;

  //loop until the number of groups=number of suffixes
  for (int h = 1; sorGroup[n - 1] != n - 1; h <<= 1) {

    sort(suf, suf + n, comp(h));  //sort the array using the first 2h chars

    for (int i = 1; i < n; i++)  //compute the 2h group data given h group data
      sorGroup[i] = sorGroup[i - 1] + comp(h)(suf[i - 1], suf[i]);

    for (int i = 0; i < n; i++)  //copy the computed groups to the group array
      group[suf[i]] = sorGroup[i];
  }
}
```

i+h vs i-h observations

Let 2h = 8
For suffix at i = 14

i-h = 10
Longer suffix

i+h = 18
Shorter suffix

Bottom up perspective

Suffix i (already sorted) is part of a longer suffix (to be sorted)

Observation on the **longer suffix** leads to O(nlogn)

Top down perspective

Suffix i (to be sorted) includes a shorter suffix (already sorted)

Observation on the **shorter suffix** leads to O(nlogn^2)

i-h

0	abracadabra
1	bracadabra
2	racadabra
3	acadabra
4	cadabra
5	adabra
6	dabra
7	abra
8	bra
9	ra
10	a
11	

i = 8
h = 2
i-h = 6
bra part of dabra

i = 8
h = 4
i-h = 4
bra part of cadabra

Improving the algorithm

- FYI, O(nlognlogn) other explanation.
- In next time, O(nlogn) will be explained
  - Followed by LCP Algorithm
  - Then Some examples for applying these 2 algorithms
- Most of time, one can use this algorithm is a **black box** and solve complex problems
- Codes in this session and next ones from my coach wahab (aka fegla) library

About Suffix Tree

- Suffix Tree is a **compressed trie** of all **suffixes** of the given text.
- However, the efficient algorithms are not trivial to explain/implement
  - But much fun to study and understand
  - One can understand **the tree** and use it as black box
- Suffix array [O(nlogn)] is space efficient and most probably will be enough for most of the competitions problems.

تم بحمد الله

علمكم الله ما ينفعكم

ونفعكم بما تعلمتم

وزادكم علماً