P THINK FAST S

Competitive Programming

From Problem 2 Solution in O(1)

String Processing Algorithms

Suffix Arrays - Examples

Mostafa Saad Ibrahim
PhD Student @ Simon Fraser University

Applications

- Suffix arrays can be solved to solve a variety of problems
- We will list some examples.
- The proper way is to freeze the video and think seriously about the solution by yourself
- Then listen to me.
- Study properly :)

Src: https://sheridanmath.wikispaces.com/file/view/GEOMETRY_04.GIF/177066721/GEOMETRY_04.GIF

Sense of the algorithm

- If you are asked about something that consider every position in the string
- If there is explicit/implicit need to consider sorting among these positions
  - Resolving ties: lexicographically
- If there is a need to compare these positions together = LCP
  - E.g. substring relationships

Src: https://sheridanmath.wikispaces.com/file/view/GEOMETRY_04.GIF/177066721/GEOMETRY_04.GIF

Searching a pattern

- [UVA 10679] Given a large text and Q queries, which string queries exist in the text
  - E.g. text: abracadabra,
  - Query 1: adabra => exist
  - Query 2: adabx => not exist
- Hint: Remember suffix array definition

Searching a pattern

- Given that suffix array is sorted, you can think of it as classical search for a number in a sorted array
- So do binary search
  - is the query string "prefix" of the middle suffix?
  - If no, either we are smaller than the middle or not
  - Based on that, search on left or right of the current suffix array

Searching a pattern

```
// Src: http://www.geeksforgeeks.org/suffix-array-set-1-introduction/
void find_patterns_queries() {
  cin >> str;
  buildSuffixArray();

  cin >> pat;
  int m = strlen(pat);  // get length of pattern, needed for strncmp()

  int l = 0, r = n - 1;
  while (l <= r) {
    // See if 'pat' is prefix of middle suffix in suffix array
    int mid = l + (r - l) / 2;
    int res = strncmp(pat, str + suf[mid], m);

    if (res == 0) {
      cout << "Pattern found at suffix: " << suf[mid] << "\n";
      return;
    }
    // Move to left or right based on string comparison results
    if (res < 0)
      r = mid - 1;
    else
      l = mid + 1;
  }
  // We reach here if return statement in loop is not executed
  cout << "Pattern not found\n";
}
```

Longest repeated substring

- [UVA 11512] Given a string, find the longest substring that repeats 2+ times. If **tie**, the smallest lexi
  - E.g. G**A**TT**A**CA => A 3
  - E.g. **GAGAG**AG => GAGAG 2
- In this type of problems you should be careful from the **overlap concern**
  - Substrings can be overlapped or not?
  - In this problem, YES, which is the direct case

Longest repeated substring

- Solution
  - We care about every position
  - Repeated substring = common prefixes = LCP
  - LCP array has all sorted suffixes with LCP between close ones
  - So LCP array tells us the longest common substring = Just find max value in this array
  - Then use this value to build the string
  - Also use it to find how manytimes it repeats

Longest repeated substring

```
int max = -1, idx = n, lcpidx = n;

for (int i = 0; i < n+1; ++i) {
    if(lcp[i] > max)
        max = lcp[i], idx = suffix[i], lcpidx = i;
}

string ans = "";
for (int j = 0; j < max; ++j) {
    ans += str[j+idx];
}

int len = 1, i = lcpidx;
if(max)
    while(i < n+1 && lcp[i] >= max) i++, len++;

if(max == 0)
    cout<<"No repetitions found!\n";
else
    cout<<ans<<" "<<len<<"\n";
```

Longest common substring between 2 strings

- [SPOJ LONGCS]: Given 2 (or K) strings, find all the **longest common substrings** between them
  - E.g. atgc tga => tg

Longest common substring between K strings

- Given 2 strings, find all the **longest common substrings** between them
  - Suffix array process a single string and tell us common things
  - If we concatentaed S1S2, we can find all common substring, but we won't know which substring belong to what?
  - Concatenate S1$S2, where $ is a delimiter not in both
  - Find the max values of LCP array where suf[i] is from S1 and suf[i-1] is from S2 or vice versa.

Smallest lexicographic rotation

- Given a string, find which rotation makes it the smallest string
- Example string alabala (n = 7)
  - Has 7 rotations: alabala, labalaa, abalaal, balaala, alaalab, laalaba, **aalabal**
  - The smallest is aalabal
  - What about baabaa? There are 2 right rotations
  - In tie, find the smallest one in # of shifts

Smallest lexicographic rotation

- Solution:
  - Whenever asked for **rotations**, think about **<u>concatenating</u>** string to itself
  - E.g. aalabal + aalabal = aalabalaalabal
  - Compute suffix array on this string
  - Now, your turn again, how to find the solution?

Smallest lexicographic rotation

- Solution:
  - All our generated suffixes of the first n letters has n+ letters
  - Among these ones we need to find the smallest suffix
  - So iterate on **all** 2n+1 suffixes
    - For ONLY the ones in the first string (e.g. sufix[i] < n), find the first one.
    - Now, if there are several correct ones, they will be consecutive in the array. So using LCP, get the one with **lowest** suf[i]
    - In fact, the **last** on in LCP array with common length >= n is the right one as ***larger suffix has lower index***

Smallest lexicographic rotation

- Solution:
  - Easy idea? Yes. But tricky to code correctly
  - 1) you should go through the 2n sorted suffixes NOT only the first N ones in the array
  - Why? For string aa, the first string suffix will be in the end of the sorted suffix array: 4 3 2 **1 0**
  - 2) Once you found the first instance, iterate carefully to get the smallest one or you will get WA
  - Many Similar problems to solve(UVA [719, 1314], LIVEARCHIVE 2755, SPOJ [BEADS, WINMOVE]
  - **SPOJ WINMOVE** seems the strongest test cases

Smallest lexicographic rotation

```
cin >> str;
int n = strlen(str);

for (int i = 0; i < n; ++i)  // Concatenate
  str[i + n] = str[i];
str[2 * n] = 0;

buildSuffixArray();
buildLcp();

// cases: aa is tricky. Fail if used <= n NOT <= 2*n
  // Ass the first string suffixes are all in 2nd part of array
for (int i = 0; i <= 2*n; ++i) {
  if (suf[i] < n) { // from suffixes of 1st string
    while (lcp[i + 1] >= n)
      ++i;  // last one of common prefix will have smallest index
    cout << suf[i] << "\n";
    break;
  }
}
```

Smallest lexicographic rotation

- This problem has several algorithms specific for it that are O(n)
- One example is **minimum expression algo**
  - Code. Explantation. Feel free to ignore.
- One nice algorithm is based on **Lyndon Factorization Algorithm**
  - I like it. Looks like good utililty in your library
  - I never used in other things :(
  - So feel free to ignore too

Lyndon decomposition

- Given string S of length N it finds in O(N) time and O(1) memory its Lyndon decomposition:
  - A decomposition into substrings S1,S2,...,Sk such that:
  - S1+...+Sk = S and S1>=S2>=...>=Sk and each Si is a simple string
  - **Simple string** = Every suffix of Si is lexicographically strictly smaller than Si itself.
  - For every string S there exists a **unique** Lyndon decomposition.
  - (ZAA -> Z, A, A), (ZAB -> Z, AB), (CBA -> CBA)

Lyndon decomposition

- In short, the string S is called Lyndon word iff it's the smallest rotation of itself.
- For example, "ab" is a Lyndon word, but "ba" is not.
- "aa" also not a Lyndon word because there is a rotation which equals to the string.

Lyndon decomposition

```
void DuvalLyndonDecompse(string s) {
    int n = sz(s), i = 0;
    while (i < n) {
        int j=i+1, k=i;
        while (j < n && s[k] <= s[j]) {
            if (s[k] < s[j])
                k = i;
            else
                ++k;
            ++j;
        }
        while (i <= k) {
            cout << s.substr (i, j-k) << ' ';
            i += j - k;
        }
    }
}
```

Smallest lexicographic rotation

```
//Modification of Lyndon decomposition:
// The concatenation will have in between one of si
//that is less than reminder
int min_cyclic_shift(string &s) {
    s += s;
    int n = sz(s), i = 0, ans = 0;
    while (i < n/2) {
        ans = i;
        int j=i+1, k=i;
        while (j < n && s[k] <= s[j]) {
            if (s[k] < s[j])
                k = i;
            else
                ++k;
            ++j;
        }
        while (i <= k)  i += j - k;
    }
    //solution is in s.substr (ans, n/2)
    return ans;  //0-based idx of position where
                 //rotating string cause it lexico
}
```

Finally

- There are many interesting problems to solve
- Here is one: [SPOJ SUBST1] Given a string, count how many distinct substrings exist?
  - Try and then See problem 4 solution
- More examples
  - Stanford doc with many examples
  - Topcoder Forum suffix arrays

تم بحمد الله

علمكم الله ما ينفعكم

ونفعكم بما تعلمتم

وزادكم علماً