GB21802 - Programming Challenges

Week 9 - String Problems

Claus Aranha
caranha@cs.tsukuba.ac.jp

College of Information Science

2015-06-24,27

Last updated June 27, 2016

Suffix Trie pictures from “Competitive Programming 3” by Steven Halim

Last Week Results

Week 8 - Computational Geometry

- Sunny Mountains – 14/30
- Bright Lights – 4/30
- Rope Crisis – 7/30
- Bounding Box – 6/30
- Soya Milk – 22/30
- SCUD Busters – 1/30
- Trash Removal – 0/30
- Sultan's Problem – 0/30

Special Notes (1): Course Evaluation

Please take 20-30 minutes to complete the course evaluation.

Special Notes (2): Final Deadlines

- FINAL deadline for late submissions: July 8th, 00:01
- Submissions after this grade will not be considered.
- Grades will be announced on Manaba after this deadline.
- Deadline for questions/comments on grades: July 14th.

Final Week: String Problems

In Programming Contests, string-based problems have become less popular. Some reasons:

- String problems are often troublesome to code, with many special checks;
- Not as many “beautiful” algorithms as in math probems

On the other hand, string problems still often turn up in real life applications, such as:

- Pre-processing of data for analysis;
- Bioinformatics;
- Human Interfaces;

This week, we will see a few representatives of string problems in programming contests.

Representative Ad-hoc string problems

String problems in programming contests usually take one of the following forms:

- Pre-processing: Input data is in string form, but the underlying problem is of another type.
- String Matching: Compare two strings for similarities/differences. We will see more of this towards the end of the class.
- Encode/Decode: Find rules to transform encrypted text into normal text (or vice-versa). Usually requires comparing multiple possible transformations, and deciding which one is correct.
- Parsing: A grammar (sometimes recursive) is defined, and you must parse the input following this grammar rules. Sometimes you have to discover the minimal grammar based on the input.
- Substring: Find a substring with certain properties inside a larger strings;

The last three types of problems are usually solved using some sort of search, recursion, or DP.

String Basic Operations: A primer (1)

String Representation

```
// C/C++
char[100] str; //last is \0

#include<string>
str s;
```

```
// JAVA
String str;
```

Data Input

```
// Word
scanf("%s",&str); cin >> str;

// Line
gets(str);
fgets(str,1000,strdin);
getline(cin,str);
```

```
// Word
Scanner sc = new
       Scanner(System.in);
str = sc.next();

// Line
str = sc.nextLine();
```

String Basic Operations: A primer (2)

String Output and formatting

```
// C/C++
printf("s = %s, l = %d\n",
   str, (int) strlen(str));
cout << "s = " << str <<
   ", l= " << str.length()
   << endl;
```

```
// JAVA
System.out.print("..."); OR
System.out.println(); OR
System.out.printf(
   "s = %s, l= %d\n", str,
   str.length();)
```

Testing Two Strings for Equality

```
result = strcmp(str,"test");
result = (str == "test");
```

```
result =
   str.equals("test");
```

String Basic Operations: A primer (3)

Combining Two or More Strings

```
// C/C++
strcpy(str,"hello");
strcat(str," world");
str = "hello";
str.append(" world");
```

```
// JAVA
str = "hello";
str += " world";
// Careful!
// Creates new strings
```

Editing/Testing single characters in a string

```
#include <ctype.h>
for (int i=0;str[i];i++)
   str[i] = toupper(str[i])
```

```
// Java Strs are immutable
// create a new string
// or use StringBuffer
```

String Basic Operations: A primer (4)

String Tokenizer

```
// C/C++
#include <string.h>
for (char *p.strtok(str," ");
     p; p=strtok(NULL," "))
   printf("%s",p)

#include <sstream>
stringstream p(str);
while (!p.eof()) {
  string token;
  p >> token;
  }
```

```
// JAVA
import java.util.*;
StringTokenizer st = new
  StringTokenizer(str," ");
while (st.hasMoreTokens())
  System.out.println(
     st.nextToken());
```

String Basic Operations: A primer (5)

Finding a Substring in a String

```
// C/C++
char *p=strstr(str,substr);
if (p) printf("%d",p-str-1);

int pos=str.find(substr);
if (pos!=string::npos)
  cout << pos-1 << endl;
```

```
// JAVA
int pos =
  str.indexOf(substr);
if (pos != -1)
  System.out.println(pos);
```

Sorting Characters in a string

```
#include <algorithm>
sort(s, s+(int)strlen(s));
sort(s.begin(),s.end());
```

```
//Immutable, break the
//string using
//toCharArray()
```

String Basic Operations: A primer (6)

Sorting an array of strings or characters

```
// C/C++
#include <algorithm>
#include <string>
#include <vector>
vector<string> s;
// strings are put into s
sort(s.begin(), s.end())
```

```
// JAVA
Vector<String> s =
   new Vector<String>();
Collections.sort(S);
```

Discussion of Ad-hoc problems

- Immediate Decodability: Goal is to make sure no binary string is a prefix of another one. Full search by string. Is it possible to find a mathematical solution? Why/Why not?
- Caesar Cypher: This was a real cypher used by Ancient Romans. Also complete search: where is the loop?
- Ensuring Truth: A SAT problem is NP-complex, but this problem is much easier. What do you have to evaluate?
- Smeech: What is the formula of the expected values? Problem requires recursive parsing of the expression.

String Matching

Problem Definition

Given a pattern string P, can P be found in the longer string T?

OBEY

ASPBOBEBLEOLBOBEYEYBEOLBEAY

- Easiest solution: use the string library as we described before;
- However, sometimes the matching has to be modified for special cases;
- Useful to know efficient matching algorithms;

String Matching: Naive Algorithm

A basic approach for string matching is the complete search: For every position $i \in n$, test if you can find the substring m there.

```
int naiveMatching() {
  for (int i = 0; i < n; i++)
    bool found = true;
    for (int j = 0; j < m && found; j++)
      if (i+j >= n || M[j] != N[i+j]) found = false;
    if (found) printf("M is found at index %d in N\n",i);}
```

For regular natural text, this runs around $O(n)$, but for the worst case the cost becomes $O(mn)$. Example: “AAAAAAAAAB” and “AAAAB”.

The Knuth-Morris-Pratt (KMP) Algorithm

Basic Idea

The KMP algorith will never re-match a character in M that was matched in N.

If KMP finds a mismatch, it will skip n to $m+1$, and rewind m to the appropriate value to continue the match.

```
N = COLIN COMBWELL CALLED A CO-CO-CO-COMBO BREAKER
M = CO-COMBO
    CO....
          CO.......
                   C........
                            CO-CO.
                               co-CO.
                                  co-COMBO
```

The Knuth-Morris-Pratt (KMP) Algorithm

How it works

To determine to where in M the algorithm should rewind in case of a MISS, the KMP prepares and keeps a “back table”.

```
m =  0 1 2 3 4 5 6 7
     C O - C O M B O
b = -1 0 0 0 1 2 0 0
```

If a miss happens at $m = 5$ (M), the algorithm will try to rematch the string at $m = b[5] = 2$ (-).

The Knuth-Morris-Pratt (KMP) Algorithm – Code

```
char N[MAX_N], M[MAX_N];
int b[MAX_N], n, m;

void kmpPreprocess() {
  int i = 0, j = -1; b[0] = -1;
  while (i < m) {
     while (j >= 0 && M[i] != M[j]) j = b[j];
     i++; j++;
     b[i] = j; }}

void kmpSearch() {
  int i = 0, j = 0;
  while (i < n) {
     while (j >= 0 && N[i] != M[j]) j = b[j];
     i++; j++;
     if (j == m) {
        printf("M is found at index %d in N\n", i-j);
        j = b[j]; }}}
```

String Processing with Dynamic Programming

Many String problems can be reduced to a search problem and, as such, can be sped-up by the use of Dynamic Programming.

Let's discuss a few of them.

- String Alignment/Edit Distance
- Longest Common Subsequence

Note: For strings and DP, usually the indexes of the DP table are the *start/end indexes* of the string/substring, and not the string/substring itself. Easier to work with numbers.

String DP: Edit Distance

Problem Definition

Align two strings, *A* and *B*, with the maximum alignment score (or the minimum number of edit operations).

- A[i] and B[i] is a match (+2 score)
- Need to replace A[i] with B[i] (-1 score)
- Need to add a space to A[i] (-1 score)
- Need to delete a character from A[i] (-1 score)

```
Non-optimal example
A = 'ACAATCC' = 'A_CAATCC'
B = 'AGCATGC' = 'AGCATG_C'   SCORE
                 2-22---2    4*2 + 4*-1 = 4
```

Trying to solve this by brute force would quickly get TLE ($O(3^n)$).

Edit Distance: Bottom Up DP Approach (1)

State table

Given two strings A[1..n] B[1..m],

$V(i, j)$ is the best cost for matching the *substrings* A[1..i], B[1..j].

Score(C_1, C_2) is the score of matching characters C_1 and C_2.

Initial Conditions

- $V(0, 0) = 0$ – No points for matching empty strings
- $V(i, 0) = i *$ Score($A[i]$,_) – Delete all A[i]
- $V(0, j) = j *$ Score(_,$B[j]$) – Insert all B[j] in A

Edit Distance: Bottom Up DP Approach (2)

State table

Given two strings A[1..n] B[1..m],

$V(i, j)$ is the best cost for matching the *substrings* A[1..i], B[1..j].

Score(C_1, C_2) is the score of matching characters C_1 and C_2.

Transition Rule:

$V(i, j) = \max(option1, option2, option3)$

- option1 = V(i-1,j-1) + C(A[i],B[j]) // Match or mismatch
- option2 = V(i-1,j) + Score(A[i],_) // Delete A[i]
- option3 = V(i,j-1) + Score(_,B[j]) // Insert B[j]

Longest Common Subsequence

Problem Definition

Given two strings *A* and *B*, what is the longest common subsequence between them?

Example:

```
String A:    'ACAATCC'      - A_CAAT_CC
String B:    'AGCATGC'      - AGCA_TGC_
Longest Common Subsequence: A_CA_T_C_

LCS: ACATC
```

- The LCS problem is similar to the String Alignment problem;
- The same DP algorithm presented before can be used;
- Set cost of Mismatch to $-$inf, the cost of insert/deletion to 0, and the cost of matching to 1;

Longest Palindrome

Problem Description

Given a string S of size up to $N = 1000$ characters, what is the longest palindrome that you can make by deleting characters from S?

Examples

- ADAM – ADA
- MADAM – MADAM
- NEVERODDOREVENING – NEVERODDOREVEN
- RACEF1CARFAST – RACECAR

Longest Palindrome

Problem Description

Given a string S of size up to $N = 1000$ characters, what is the longest palindrome that you can make by deleting characters from S?

DP Solution:

- State Table:
- Start Conditions:
- Transition:

This DP has complexity $O(n^2)$

Longest Palindrome

Problem Description

Given a string S of size up to $N = 1000$ characters, what is the longest palindrome that you can make by deleting characters from S?

DP Solution:

- State Table:
  - len(i,j) - The largest palindrome found between i and j
- Start Conditions:
- Transition:

This DP has complexity $O(n^2)$

Longest Palindrome

Problem Description

Given a string S of size up to $N = 1000$ characters, what is the longest palindrome that you can make by deleting characters from S?

DP Solution:

- State Table:
  - len(i,j) - The largest palindrome found between i and j
- Start Conditions:
  - If $l = r$ then $\text{len}(l, r) = 1$.
  - If $r = l + 1 \text{and} S[l] = S[r]$, $\text{len}(l, r) = 2$, else $\text{len}(l, r) = 1$.
- Transition:

This DP has complexity $O(n^2)$

Longest Palindrome

Problem Description

Given a string S of size up to $N = 1000$ characters, what is the longest palindrome that you can make by deleting characters from S?

DP Solution:

- State Table:
  - len(i,j) - The largest palindrome found between i and j
- Start Conditions:
  - If $l = r$ then $\text{len}(l, r) = 1$.
  - If $r = l + 1 \text{and} S[l] = S[r]$, $\text{len}(l, r) = 2$, else $\text{len}(l, r) = 1$.
- Transition:
  - If $S[l] = S[r]$, then $\text{len}(l, r) = 2 + \text{len}(l + 1, r - 1)$;
  - else $\text{len}(l, r) = \max(\text{len}(l + 1, r), \text{len}(l, r - 1))$

This DP has complexity $O(n^2)$

Suffix Trie: Definition

Definition

Data structure used to find matching suffixes of multiple strings.

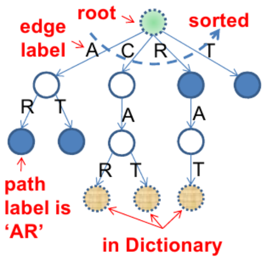
Suffix Trie for {'CAR','CAT','RAT'}

All Suffixes

1. CAR
2. AR
3. R
4. CAT
5. T
6. RAT
7. AT
8. T

Sorted, Unique Suffixes

1. AR
2. AT
3. CAR
4. CAT
5. R
6. RAT
7. T



Suffix Trie: Using it for a single, long string

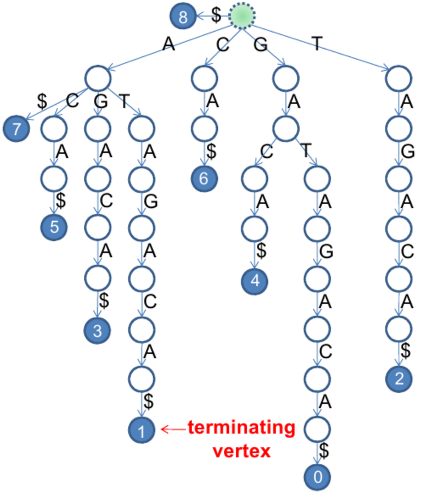
Create all *n* suffixes:

i	suffix
0	GATAGACA$
1	ATAGACA$
2	TAGACA$
3	AGACA$
4	GACA$
5	ACA$
6	CA$
7	A$
8	$

Count the occurence of substring *m*:

- 'A': 4 times
- 'GA': 2 times
- 'AA': 0 times

Suffix Trie (T='GATAGACA$')

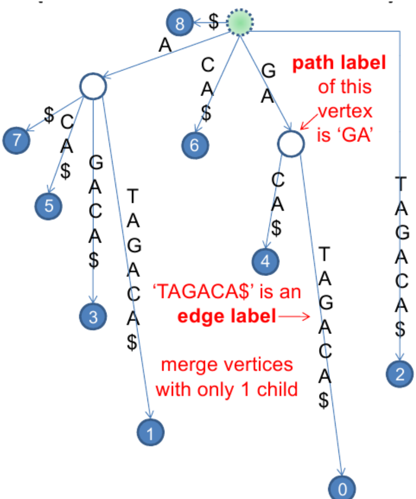


Suffix Trie: Suffix Tree

Suffix Trie (T='GATAGACA$')
Compress single child nodes to obtain "Suffix Tree"

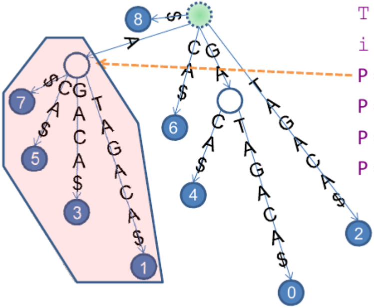
i	suffix
0	GATAGACA$
1	ATAGACA$
2	TAGACA$
3	AGACA$
4	GACA$
5	ACA$
6	CA$
7	A$
8	$

With the suffix tree, many algorithms become faster.



Uses of a Suffix Tree 1: String Matching

Assuming that we have the Suffix Tree already built, we can find all occurrences of substring m in T in time $O(m + \text{occ})$, where occ is the number of occurrences.



T = 'GATAGACA$'

i = '012345678'

P = 'A' → Occurrences: 7, 5, 3, 1

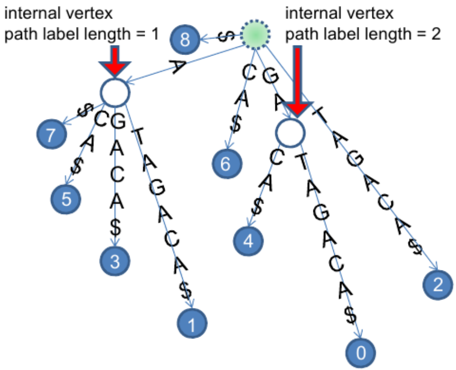
P = 'GA' → Occurrences: 4, 0

P = 'T' → Occurrences: 2

P = 'Z' → Not Found

Uses of a Suffix Tree 2: Longest Repeated Substring

- The LRS is the longest substring with number of occurrences > 2;
- The LRS is the deepest internal node in the tree;

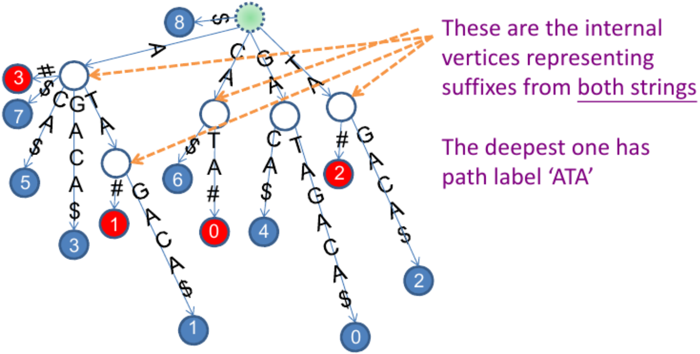


e.g. T = 'GATAGACA$'
The longest repeated substring is 'GA' with path label length = 2

The other repeated substring is 'A', but its path label length = 1

Uses of a Suffix Tree 3: Longest Common Substring

- We can find the common substring of M and N by making a combined Suffix Tree. Each string has a different ending character.
- The common substring is the deepest node that has both characters.



Suffix Trie: Suffix Array (1)

- The algorithms in previous slides are very efficient...
  ... if you have the suffix tree
- The suffix tree can be built in $O(n)$...
  ... but implementation is rather complex;
- In this course, we will see the Suffix Array;
- The Suffix Array is built in $O(n \log n)$...
  ... but the implementation is very simple!

I encourage you to study the implementation of the suffix tree by yourself!

Suffix Trie: Suffix Array (2)

- To make a Suffix array, make an array of all possible suffixes of T, and sort it;
- The order of the suffix array is the visit in preorder of the suffix tree;
- We can adapt all algorithms accordingly;

i	suffix
0	GATAGACA$
1	ATAGACA$
2	TAGACA$
3	AGACA$
4	GACA$
5	ACA$
6	CA$
7	A$
8	$

Sort $\rightarrow$

i	SA[i]	suffix
0	8	$
1	7	A$
2	5	ACA$
3	3	AGACA$
4	1	ATAGACA$
5	6	CA$
6	4	GACA$
7	0	GATAGACA$
8	2	TAGACA$

Suffix Array: Implementation (1)

Simple Implementation

```
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;
char T[MAX_N]; int SA[MAX_N],i,n;

bool cmp(int a, int b) { return strcmp(T+a, T+b) < 0; }
// O(n)

int main() {
  n = (int) strlen (gets(T));
  for (int i = 0; i < n; i++) SA[i] = i;
  sort (SA, SA+n, cmp); // O(n^2 log n) }
```

This implementation is too slow for strings bigger than 1000 characters.

Suffix Array: Implementation (2.1)

O(n log n) implementation using “ranking pairs/radix sort”

```
char T[MAX_N]; int n; int c[MAX_N];
int RA[MAX_N], tempRA[MAX_N], SA[MAX_N], tempSA[MAX_N];

void countingSort(int k) {
  int i, sum, maxi = max(300,n); //255 ASCII chars or n
  memset(c, 0, sizeof(c));
  for (i = 0; i < n; i++) c[i+k<n? RA[i+k] : 0]++
  for (i = sum = 0; i < maxi; i++)
    { int t = c[i]; c[i] = sum; sum += t;} //frequency
  for (i = 0; i < n; i++)
    tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];
  for (i = 0; i < n; i++) // update suffix array
    SA[i] = tempSA[i];
}

// ... continues next slide
```

Suffix Array: Implementation (2.2)

O(n log n) implementation using “ranking pairs/radix sort”

```
// ... continued from last slide

void constructSA() {
  int i, k, r;
  for (i = 0; i < n; i++) { RA[i] = T[i]; SA[i] = i;}
  for (k = 1; k < n; k <<=1) {
    countingSort(k); countingSort(0);
    tempRA[SA[0]] = r = 0;
    for (i = 1; i < n; i++) tempRA[SA[i]] =
          (RA[SA[i]] == RA[SA[i-1]] &&
           RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;
    for (i = 0; i < n; i++)
      RA[i] = tempRA[i];
    if (RA[SA[n-1]] == n-1) break;
}}
```

Suffix Array: Using Suffix Array (1)

String Matching: Finding ’GA’

- Do a binary search once to find the lower bound;
- Do a binary search once to fint the upper bound;

Finding lower bound

i	SA[i]	Suffix
0	8	$
1	7	A$
2	5	ACA$
3	3	AGACA$
4	1	ATAGACA$
5	6	CA$
6	4	GACA$
7	0	GATAGACA$
8	2	TAGACA$

Finding upper bound

i	SA[i]	Suffix
0	8	$
1	7	A$
2	5	ACA$
3	3	AGACA$
4	1	ATAGACA$
5	6	CA$
6	4	GACA$
7	0	GATAGACA$
8	2	TAGACA$

Suffix Array: Using Suffix Array (2)

Longest Repeated Substring

Find the longest common prefix between suffix i and $i+1$

i	SA[i]	LCP[i]	Suffix
0	8	0	$
1	7	0	A$
2	5	1	ACA$
3	3	1	AGACA$
4	1	1	ATAGACA$
5	6	0	CA$
6	4	0	GACA$
7	**0**	**2**	**GATAGACA$**
8	2	0	TAGACA$

Suffix Array: Using Suffix Array (3)

Longest Common Substring

- Create Suffix Array for appended strings *MN*;
- Find the longest common prefix that has both string enders;

i	SA[i]	LCP[i]	Owner	Suffix
0	13	0	2	#
1	8	0	1	$CATA#
2	12	0	2	A#
3	7	1	1	A$CATA#
4	5	1	1	ACA$CATA#
5	3	1	1	AGACA$CATA#
6	10	1	2	ATA#
7	**1**	**3**	**1**	**ATAGACA$CATA#**
8	6	0	1	CA$CATA#
9	9	2	2	CATA#
10	4	0	1	GACA$CATA#
11	0	2	1	GATAGACA$CATA#
12	11	0	2	TA#
13	2	2	1	TAGACA$CATA#

Problem Discussion

- Immediate Decodability
- Caesar Cypher
- Ensuring Truth
- Smeech
- String Partition
- Prince and Princess
- Power Strings
- Life Forms

Final Message

Thank you for participating in this class!
I hope your programming abilities have improved!

Remember: Your brain is like a muscle, it needs constant practice to keep itself smart