

GB21802 - Programming Challenges

Week 8 - String Problems

Claus Aranha

caranha@cs.tsukuba.ac.jp

College of Information Science

2017-06-16,19

Last updated June 15, 2017

Results for the Previous Week

Here are the results for last week:

Week 7: Computational Geometry

Deadline: 6/16/2017, 11:59:59 PM (2 days, 00:03 hours from now)

Problems Solved -- 0P:29, 1P:7, 2P:4, 3P:14, 8P:1,

#	Name	Sol/Sub/Total	My Status
1	Sunny Mountains	22/22/55	
2	Bright Lights	4/9/55	
3	Packing polygons	1/3/55	
4	Elevator	19/20/55	
5	Soya Milk	16/17/55	
6	Trash Removal	1/1/55	
7	The Sultan's Problem	1/1/55	
8	Board Wrapping	1/1/55	

Final Week: String Problems

String manipulation is very common in real life applications:

- Pre-processing of data for analysis; – (JSON, CSV, Code)
- Bioinformatics; – (Manipulation of symbols)
- Human Interfaces; – (Text input/Output)

Characteristics of String Problems

- Many “parsing” problems: special input, special output;
- Algorithms are usually derivated from DP;
- Special data structures for substrings;

This week, we will see some of these algorithms.

Representative Ad-hoc string problems

“Pure” string problems are usually found in these forms:

- **Pre-processing/Parse:** Input data is in string, but the main problem is just numbers.
- **String Matching:** Compare two (or more) strings for similarities/differences. Find substrings.
- **Encode/Decode:** Transform encoded (encrypted) text into normal text (or vice-versa). Usually many solutions are possible (search for maximum solution).

Use of DP is very common!

String Basic Operations: A primer (1)

String Representation

```
// C/C++                                // JAVA
char[100] str; // needs \0 in the end!str;

#include<string>
str s;
```

Important to know that Java strings are immutable!

Data Input

```
// Word                                // Word
scanf("%s",&str); cin >> str;  Scanner sc = new
                                Scanner(System.in);
// Line                                str = sc.next();
gets(str);
fgets(str,1000,stdin);           // Line
getline(cin,str);                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 =
result = (str == "test");        str.equals("test");
```

String Basic Operations: A primer (3)

Combining Two or More Strings

<pre>// C/C++ strcpy(str, "hello"); strcat(str, " world"); str = "hello"; str.append(" world");</pre>	<pre>// JAVA str = "hello"; str += " world"; // Careful! // Creates new strings</pre>
---	---

Editing/Testing single characters in a string

<pre>#include <ctype.h> for (int i=0; str[i]; i++) str[i] = toupper(str[i])</pre>	<pre>// Java Strs are immutable // create a new string // or use StringBuffer</pre>
---	---

String Basic Operations: A primer (4)

String Tokenizer – Breaks a string in “words”

```
// 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 – 1

The first four problems are “ad-hoc”. What string operations are necessary to solve them?

Problem 2 – Caesar Cypher

This problem requires you to find the correct *rotational cypher*. For example:

Dictionary:

- CAT
- MOUSE
- TIGER
- GET

ECR RGECP

What is the general function needed to test a dictionary?

Discussion of Ad-hoc problems – 2

The first four problems are “ad-hoc”. What string operations are necessary to solve them?

Problem 1 – Immediate decodability

Given a set of binary symbols, you must answer the question: Are all symbols *immediately unique*?

- 001
- 100
- 11011
- 10101
- 10

What is the algorithm to test uniqueness?

- By Brute force?
- A better algorithm?

Discussion of Ad-hoc problems – 2

The first four problems are “ad-hoc”. What string operations are necessary to solve them?

Problem 3 and 4 – Parsers

Problems 3 and 4 are both parsers. They require you to parse the input and make a simple operation:

Problem 3: Parsing logical equation:

$(x \& y) \mid (x \& z) \mid (y \& \bar{y})$

Problem 4: Parsing nested function:

$(0.5 \ 3 \ (0.2 \ (0.5 \ 3 \ 3) \ (0.4 \ 2 \ 3)))$

- How do you parse each problem?
- In each case, do you calculate the problem **during** or **after** parsing?

String Matching

Many String problems include some form or [string matching](#)

Can you find a substring P inside of string T ?

- $P = \text{OBEY}$
- $T = \text{ASPBOBEBLEOLBOBEY EYBEOLBEAY}$

- The easiest solution: Use strstr from the standard library!
- But... what if the search has special conditions? (ex: $0 == O$)
- Better study algorithms for string matching!

What is the complexity of a Complete search?

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: “AAAAAAAAAAB” and “AAAAB”.

The Knuth-Morris-Pratt (KMP) Algorithm

Basic Idea

The KMP algorithm 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

COXXXX

COXXXXXXXX

CXXXXXXXX

CO-COX

CO-COX

CO-COMBO

The Knuth-Morris-Pratt (KMP) Algorithm

How it works

The KMP needs to construct a “rewind table” b . This table is used to determine where in M the algorithm should [try again](#) in case of a miss.

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

$A = \text{'ACAATCC'} = \text{'A_CAATCC'}$

$B = \text{'AGCATGC'} = \text{'AGCATG_C'}$ SCORE

$$2 - 2 - 2 - - - 2 \quad 4 * 2 + 4 * -1 = 4$$

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* $A[1..i]$, $B[1..j]$.

$\text{Score}(C_1, C_2)$ is the score of matching characters C_1 and C_2 .

Initial Conditions

- $V(0, 0) = 0$ – No points for matching empty strings
- $V(i, 0) = i * \text{Score}(A[i], _)$ – Delete all $A[i]$
- $V(0, j) = j * \text{Score}(_, B[j])$ – 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* $A[1..i]$, $B[1..j]$.

$\text{Score}(C_1, C_2)$ is the score of matching characters C_1 and C_2 .

Transition Rule:

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

- $\text{option1} = V(i-1, j-1) + C(A[i], B[j])$ // Match or mismatch
- $\text{option2} = V(i-1, j) + \text{Score}(A[i], _)$ // Delete $A[i]$
- $\text{option3} = V(i, j-1) + \text{Score}(_, B[j])$ // 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 $-\infty$, 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:
 - $\text{len}(i,j)$ - 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:
 - $\text{len}(i, j)$ - The largest palindrome found between i and j
- Start Conditions:
 - If $l = r$ then $\text{len}(l, r) = 1$.
 - If $r = l + 1$ 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:
 - $\text{len}(i, j)$ - The largest palindrome found between i and j
- Start Conditions:
 - If $l = r$ then $\text{len}(l, r) = 1$.
 - If $r = l + 1$ 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.

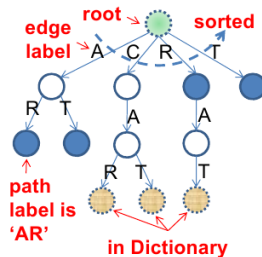
Suffix Trie for {'CAR','CAT','RAT'}

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

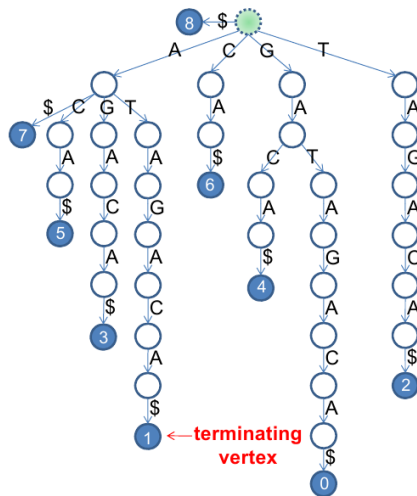
Suffix Trie ($T = \text{'GATAGACA\$'}$)

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 occurrence of substring m :

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



Suffix Trie (T='GATAGACA\$')

Compress single child nodes to obtain "Suffix Tree"

path label of this vertex is 'GA'

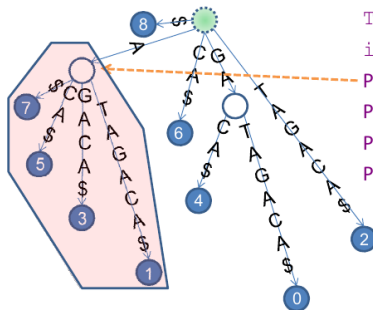
'TAGACA\$' is an edge label →

merge vertices with only 1 child

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.



$T = \text{'GATAGACA\$'}$

$i = \text{'012345678'}$

$P = \text{'A'} \rightarrow \text{Occurrences: } 7, 5, 3, 1$

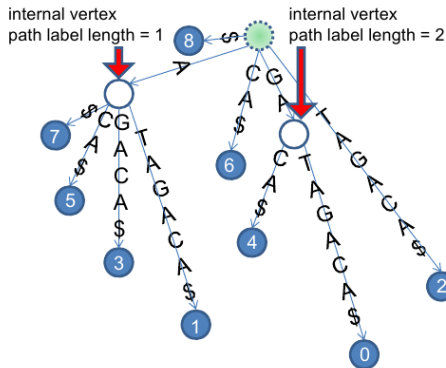
$P = \text{'GA'} \rightarrow \text{Occurrences: } 4, 0$

$P = \text{'T'} \rightarrow \text{Occurrences: } 2$

$P = \text{'Z'} \rightarrow \text{Not Found}$

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;



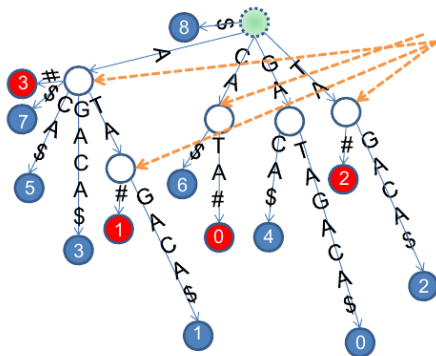
e.g. $T = \text{'GATAGACA\$'}$

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.



These are the internal vertices representing suffixes from both strings

The deepest one has path label 'ATA'

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 →

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 find 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	<u>A</u> CA\$
3	3	1	<u>A</u> GACA\$
4	1	1	<u>A</u> TAGACA\$
5	6	0	CA\$
6	4	0	GACA\$
7	0	2	<u>G</u>ATAGACA\$
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 ends;

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	<u>A</u> \$CATA#
4	5	1	1	<u>ACA</u> \$CATA#
5	3	1	1	<u>AGACA</u> \$CATA#
6	10	1	2	<u>ATA</u> #
7	1	3	1	<u>ATAGACA</u>\$CATA#
8	6	0	1	CA\$CATA#
9	9	2	2	<u>CA</u> T#
10	4	0	1	GACA\$CATA#
11	0	2	1	<u>GATAGACA</u> \$CATA#
12	11	0	2	TA#
13	2	2	1	<u>TAGACA</u> \$CATA#

This Week's Problems

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

Problem Discussion (1)

644 – Immediate Decodability

Input: A list of binary codes, each representing a different symbol.

Output: Whether any symbol in the list is a **prefix** of another symbol.

Input :

01

10

0001

00101

9

Output :

decodable (no prefixes)

001

0100

00101

01101

9

not decodable

(string 0 is a prefix of 2)

How to solve this problem?

Problem Discussion (2)

554 – Caesar Cypher

A **k-rotation cypher** replaces every symbol N with symbol $N + k$, including spaces (which are symbol 0).

- **Input:** A list of correct words, and an **encrypted text**
- **Output:** Choose the k which matches the maximum number of words in the dictionary, and output the **decrypted text**

Notes about the problem:

- Small input: is only one case, cryptotext has only 250 characters
- Only letters and spaces
- Output requirements (linebreak at 60 characters)

```
THIS      DAWN      THAT
THE       ZORRO    OTHER
AT        THING
#
BUUBDLA  PSSPABUAEBXO
```

Problem Discussion (3)

11357 – Ensuring Truth (parsing problem)

Parse the formula from the input, and evaluate if it has any possible “TRUE” evaluation.

INPUT	OUTPUT
$(a \& b \& c) \mid (a \& b) \mid (a)$	YES
$(x \& \sim x) \mid (a \& \sim a \& b)$	NO

11291 – Smeech (parsing problem)

Parse the formula from the input, and evaluate the expected value from the formula.

INPUT	OUTPUT
7	7.00
$(.5 \ 3 \ 9)$	3.00
$(.7 \ 3 \ (.5 \ 2 \ 2))$	3.80

Problem Discussion (4)

String Partition

Break a string of digits into numbers, and find the maximum possible sum.

Hints:

- The maximum integer value is important
- Treat this as a search problem

INPUT

1234554321

5432112345

000

121212121212

OUTPUT

1234554321

543211239

0

2121212124

Problem Discussion (5)

Prince and Princess

The Prince and the princess make a path through the $n * n$ grid. Both start at square 1 and end at square $n * n$, but they use different paths.

You have to eliminate steps from both to make their paths identical.

Input

```
1 7 5 4 8 3 9
1 4 3 5 6 2 8 9
```

Output

```
4 (size of common path)
(1,5,8,9)
```

```
1 3 4 2 5 8 7 10
1 5 8 9 3 2 7 10
```

```
5
(1,5,8,7,10)
```


Problem Discussion (6)

10298 – Power Strings

Imagine that a substring s^n is composed of n repetitions of s . Given s^n , find the maximum possible n .

INPUT	OUTPUT
abcd	1
abcabcabc	3
abcabcabcd	1

- This is a search problem, what would be the naive algorithm?
- Note that the size of s is up to 10^6 . So what is a better algorithm?

Problem Discussion (7)

Life Forms

You are given multiple strings and must find **the largest substring shared by more than 1/2 of the strings.**

This is a generalization of the LCS (longest common substring) algorithm.

For some reason, the time limit is 6.66 seconds!

INPUT

abcdefg

bcdefgh

cdefghi

OUTPUT

bcdefg

cdefgh

Next Week

- Final Class: Problems Remix
- Class Evaluation
- See you!