GB21802 - Programming Challenges

Week 6 - String Problems

Claus Aranha
caranha@cs.tsukuba.ac.jp

College of Information Science

2019-05-31,06-03

Last updated May 30, 2019

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

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;

Ad-hoc (one-of-a-kind) string problems

Many String Problems in programming contests are **"Ad-hoc"** (one-of-a-kind). They are usually a cover for problems of other types.

- 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++
char[100] str;
// ends with '\0'

#include<string>
str s;
```

```
// JAVA
String str;

// JAVA strings
// are imutable!
```

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 – Separates a string based on a character

```
// 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

Problem 1 – Immediate decodability

Given a set of **2 to 8 binary words**, of length between **1 and 10**, decide if the set is **immediately decodable.**

A set is immediately decodable if **no word is a prefix of another word.**

Input example 1

- 001
- 110
- 10101
- 01101
- 100

Input example 2

- 001
- 110
- 10101
- 01101
- 10

What is the brute force algorithm? What is the complexity? What is a smarter algorithm?

Discussion of Ad-hoc problems

Problem 2 – Caesar Cypher

A **rotational cypher** transforms *plaintext* to *cyphertext* by adding a constant value "k" to every character.

Example: I LOVE YOU + ($k = 3$) $\rightarrow$ LCORYHCARY

Given a dictionary of plaintext, find the best translation of the cyphertext.

THIS
DAWN
THAT
THE
ZORRO
OTHER
AT
THING
#
BUUBDLA PSSPABUAEBXO

Output:
ATTACK ZORRO AT DAWN

Discussion of Ad-hoc problems

Problem 3 – Ensuring Truth

Given a boolean formula in the following format, is the formula **satisfiable**?

$$(x_1 \wedge \hat{x}_2 \wedge \ldots \wedge x_n) \vee (x_i \wedge \hat{x}_j \wedge \ldots) \vee \ldots$$

Examples:

```
(a&b&c)|(a&b)|(a)        <--- Satisfiable;
(x&~x)                   <--- Not satisfiable;
```

- A big part of the program is to build a function to read a string with size over 5000 in the right format.
- SAT is a very hard problem, but for this particular string format, is there a simple way to calculate satisfiability?

String Matching

Many String problems include some form of string matching

Find a substring P inside of string T.

- P = OBEY
- T = ASPBOBEBLEOLBOBEYEYBEOLBEAY

- The easiest solution: Use strstr from the standard library!
- But... what if the search has special conditions?
  - **example:** O and 0 are the same character
- Let's study how to make a string matching algorithm!

First: What is the complexity of a Complete search?

String Matching: Naive Algorithm

Complete search approach: For every character *n*[*i*], test if substring *m* begins 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("Found at index %d\n",i)
```

- Average case: $O(n)$
- Worst case: $O(mn)$
  M = AAAAB, N = AAAAAAAAAAAAAAAAAB
- Why is this case bad?

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
    COXXXX
          COXXXXXXX
                   CXXXXXXXX
                            CO-COX
                               CO-COX
                                  CO-COMBO
```

The Knuth-Morris-Pratt (KMP) Algorithm

How it works

The KMP needs to construct a “M rewinded table” *b*. When a mismatch happens, the substring *M* is rewinded, but the *N* index always moves forward.

```
M =   0  1  2  3  4  5  6  7        <-- Substring index
      C  O  -  C  O  M  B  O
b = -1  0  0  0  1  2  0  0        <-- M "rewind" index
```

If a miss happens at $m = 5$ (M), the algorithm will return the M counter (j) to $j = b[5] = 2$.

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

Some string problems can be explained as a search problem. In this case, we can solve them using Dynamic Programming.

- String Alignment/Edit Distance
- Longest Common Subsequence

String DP: Edit Distance

The **Edit Distance**, **String Alignment** or **Levenhstein Distance**, consists of measuring how many spaces are needed to **minimize the difference between two strings.**

```
S1: ACAATCC -> A_CAATCC -> A_CAATCC
S2: AGCATGC -> AGCATGC_ -> AGCA_TGC
Diff:       -> +.++--+. -> +.++.+-+
```

To maximise score, we want to avoid **letter mismatches.**

Uses

- Finding similar words.
- Identifying Mispellings.

String DP: Edit Distance

Score Maximization Problem

Align two strings, *A* and *B*, with the maximum alignment score.
For each pair of characters, we have three choices:

- A[i] and B[i] are the same character ('+': +2 score)
- A[i] and B[i] are diff character ('-': -1 score)
- Add a space to A[i] or B[i] ('_': -1 score)

```
S1: ACAATCC -> A_CAATCC           -> A_CAATCC
S2: AGCATGC -> AGCATGC_           -> AGCA_TGC
Diff:       -> 2-22--2- = +4      -> 2-22-2-2 = +7
```

Trying all combinations: ($O(3^n)$). Let's try DP.

Edit Distance: Bottom Up DP Approach – Setup

State table

The state table V has dimensions *size(A)* by *size(B)*

$V[i][j]$ is the maximum score for matching *substrings* A[1..i], B[1..j].

Initial Conditions

- $V(0, 0) = 0$ – Empty Strings
- $V(i, 0) = i * -1$ – Fill "B" with "_"
- $V(0, j) = j * -1$ – Fill "A" with "_"

Edit Distance: Bottom Up DP Approach – Update

State table

The state table V has dimensions *size(A)* by *size(B)*

$V[i][j]$ is the maximum score for matching *substrings* A[1..i], B[1..j].

Transition Rule:

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

Update: $V(i, j) = \max(\mathit{option1}, \mathit{option2}, \mathit{option3})$

- option1 = V(i-1,j-1) + Score(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]

Edit Distance: Bottom Up DP Approach – Example

Match ACAATCC and AGCATGC with a table.

Problem 2 – 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:
  - 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 $O(n^2)$

Longest Palindrome

Longest Palindrome DP: Diagonal Table Top Down

len(l,r) final state

	R	A	C	E	F	1	C	A	R
R	1	1	1	1	1	1	3	5	7
A		1	1	1	1	1	3	5	5
C			1	1	1	1	3	3	3
E				1	1	1	1	1	1
F					1	1	1	1	1
1						1	1	1	1
C							1	1	1
A								1	1
R									1

len(l,r) initial state

	R	A	C	E	F	1	C	A	R
R	1	1							
A		1	1						
C			1	1					
E				1	1				
F					1	1			
1						1	1		
C							1	1	
A								1	1
R									1

```
transition:
- A[l] == A[r]?
  len(l+1,r-1)+2
- A[1] != A[r]?
  max(left,down)
```

Suffix Trie: Definition

Definition

Data structure used to find matching suffixes of multiple strings.

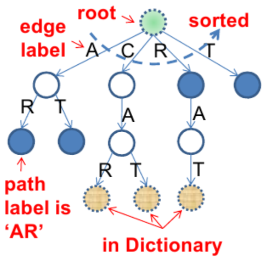
Suffix Trie for {'CAR','CAT','RAT'}

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

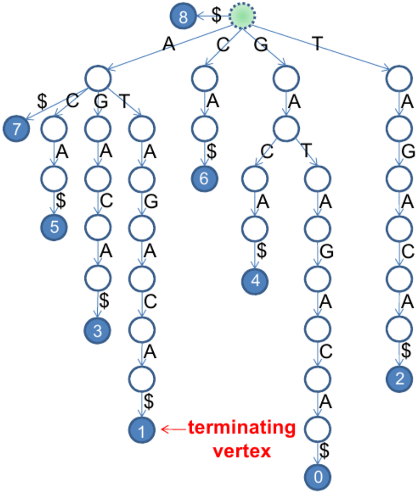
Suffix Trie (T='GATAGACA$')

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 occurence of substring *m*:

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

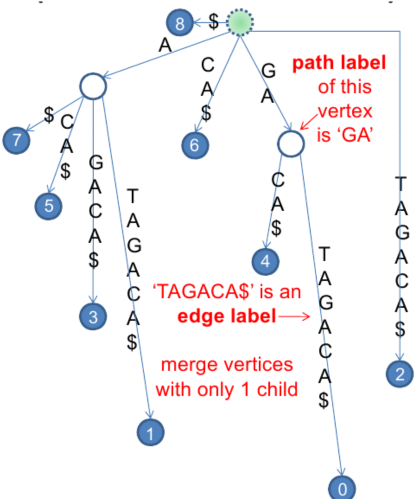


Suffix Trie: Suffix Tree

Suffix Trie (T='GATAGACA$')

Compress single child nodes to obtain “Suffix Tree”

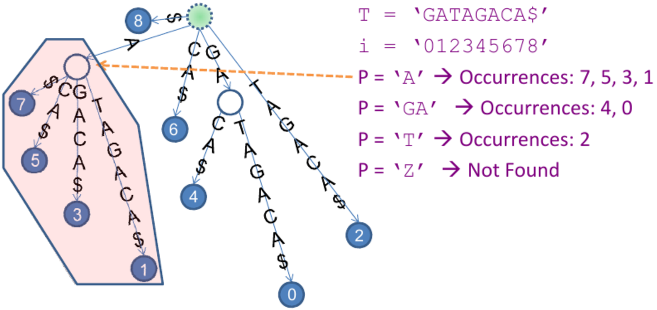
i	suffix
0	GATAGACA$
1	ATAGACA$
2	TAGACA$
3	AGACA$
4	GACA$
5	ACA$
6	CA$
7	A$
8	$



With the suffix tree, many algorithms become faster.

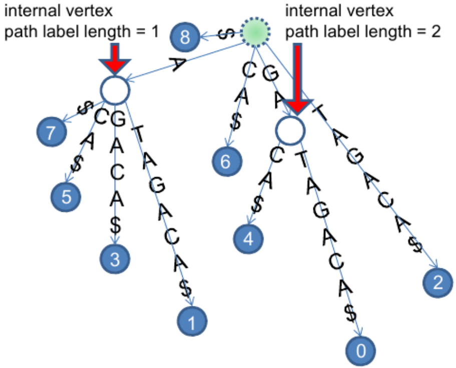
Uses of a Suffix Tree 1: String Matching

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.



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;



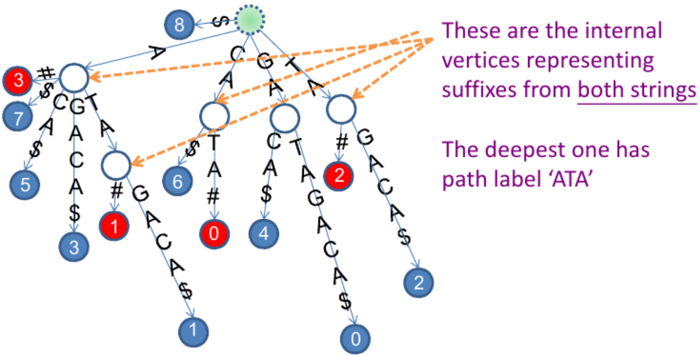
e.g. T = ‘GATAGACA$’

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#

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:	Output:
01	decodable (no prefixes)
10	
0001	
00101	
9	
001	not decodable
0100	(string 0 is a prefix of 2)
00101	
01101	
9	

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, crypotext 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)|(a&b)|(a)	YES
(x&~x)|(a&~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	OUTPUT
1234554321	1234554321
5432112345	543211239
000	0
121212121212	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                        Output
1 7 5 4 8 3 9                4 (size of common path)
1 4 3 5 6 2 8 9              (1,5,8,9)

1 3 4 2 5 8 7 10             5
1 5 8 9 3 2 7 10             (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	OUTPUT
abcdefg	bcdefg
bcdefgh	cdefgh
cdefghi	