GB21802 - Programming Challenges

Week 0B - Solving Problems

Claus Aranha
caranha@cs.tsukuba.ac.jp

Department of Computer Science

2017/4/17
(last updated: April 17, 2017)

Survey Results

About 35 respondents:

- Students: Regular: 26, Tech School: 9
- Contest Experience: 9 (30%)/ 5 (55%)
- Why do you Program?
  - Fun/Personal: 19; Money: 15; Save the world: 5;
  - More people are worried about saving the world than last year! :-)
- Languages:
  - C/C++: 15; Java: 14; Python: 10;
  - Very balanced.
- Extra: 5 people said they loved cats! I will try to put more cats in the course.

Submission Results (Sunday)

Week 0: Introduction and Problem Solving

Deadline: UCT 2017-04-20T15:00:01 (3 days, 22:27 hours from now)
Problems Solved -- 0P:21, 2P:6, 3P:9, 5P:2, 6P:2, 8P:2,

#	Name	Sol/Sub/Total	My Status
1	The 3n + 1 problem	14/20/42	
2	Cost Cutting	21/21/42	
3	Event Planning	11/15/42	
4	Horror Dash	13/13/42	
5	Y3K Problem	6/7/42	
6	Stack 'em Up	5/5/42	
7	Traffic Lights	3/3/42	
8	Population Explosion	4/4/42	

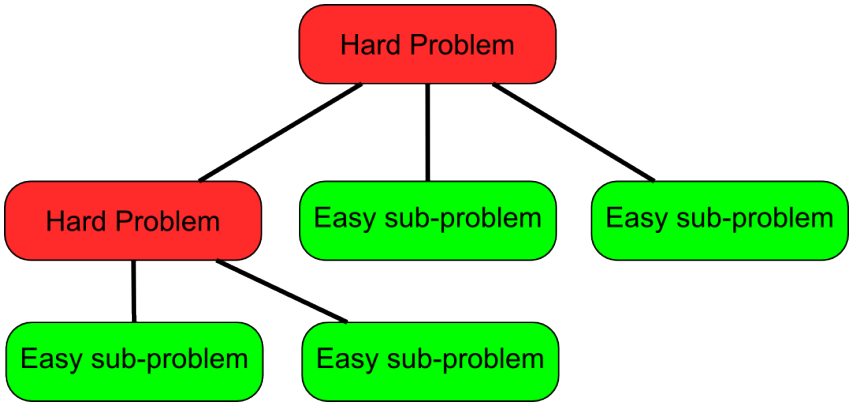
Very Good! It is going to get harder from here on.

Today's Class: Solving Problems

Hints and ideas on how to solve programming challenges

- Reading Problems
- Considerations when programming
- Input/Output
- Debugging
- Types of Problems

A Programming Challenge Workflow



First trick of solving a hard problem: break it into simpler ones:

- Task 1: Understand the problem description;
- Task 2: Understand the input/output;
- Task 3: Choose the Algorithm;
- Task 4: Write the Code;
- Task 5: Test the program on example data;
- Task 6: Test the program on hidden data;

Task 1: Understanding the Problem Description

The English description is so hard!

Don't Worry:

1. Separate the text into flavor and rules;
2. Sometimes it is easy to read the input/output first, and then the text;
3. Problems with a lot of flavor are usually not very hard.;

Example: Problem 11559 – Event Planning

Flavor:

As you didn't show up to the yearly general meeting of the Nordic Club of Pin Collectors, you were unanimously elected to organize this years excursion to Pin City. You are free to choose from a number of weekends this autumn, and have to nd a suitable hotel to stay at, preferably as cheap as possible.

rules

You have some constraints: The total cost of the trip must be within budget, of course. All participants must stay at the same hotel, to avoid last years catastrophe, where some members got lost in the city, never being seen again.

Keywords: constraints, minimum, maximum, cost, rules, number, etc...

Extreme example: UVA 1124, Celebrity Jeopardy

It's hard to construct a problem that's so easy that everyone will get it, yet still difficult enough to be worthy of some respect. Usually, we err on one side or the other. How simple can a problem really be?

Here, as in Celebrity Jepoardy, questions and answers are a bit confused, and, because the participants are celebrities, there's a real need to make the challenges simple. Your program needs to prepare a question to be solved — an equation to be solved — given the answer. Specifically, you have to write a program which finds the *simplest* possible equation to be solved given the answer, considering all possible equations using the standard mathematical symbols in the usual manner. In this context, simplest can be defined unambiguously several different ways leading to the same path of resolution. For now, find the equation whose transformation into the desired answer requires the least effort.

For example, given the answer $X = 2$, you might create the equation $9 - X = 7$. Alternately, you could build the system $X > 0$; $X^2 = 4$. These may not be the simplest possible equations. Solving these mind-scratchers might be hard for a celebrity.

Input

Each input line contains a solution in the form $< symbol > \blacksquare < value >$

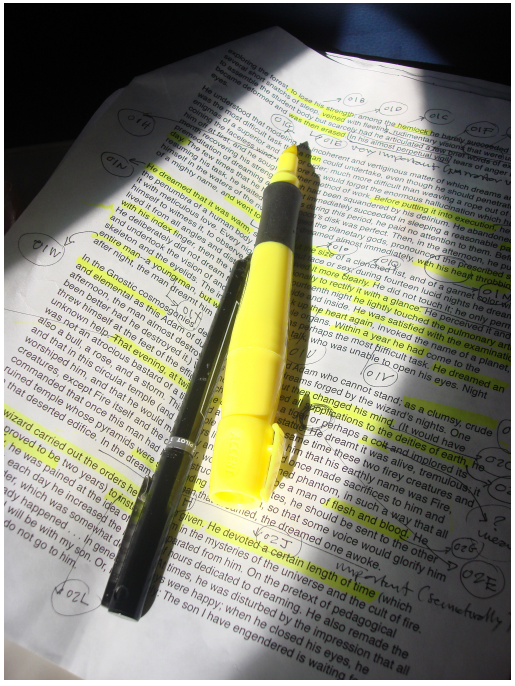
Output

For each input line, print the simplest system of equations which would to lead to the provided solution, respecting the use of space exactly as in the input.

Real Problem: Copy the input into the output. (look at the examples)

Hints for hard to read problems

- If it is hard, look at the examples;
- If it is really hard, print on paper;
- First read: mark keywords;
- Second read: cut flavor;
- Re-write the important parts in another file;
- Do not begin programming until you understand the problem!



Task 2: Understanding the Input

When you read the input, pay attention to:

- What is the problem size?
- What is the end condition?
- What type of data is it?
- What is the format of the data?

Task 2: Input Size

The input size shows how big the problem gets.

- If the problem is too small, you can try a brute force algorithm;
- If the problem is too big, you must use a more complex algorithm;

Keep in Mind!

- The average time limit in UVA is 1-3 seconds.
- Assume around 10.000.000 operations per second (in the judge).
- Use this as a rough measure of time your program spends.

Task 2: Input Size – Examples

n < 24

Exponential algorithms will work ($O(2^n)$).
Or sometimes you can just calculate all solutions.

n = 500

Cubic algorithms don't work anymore ($O(n^3) = 125.000.000$)
Maybe $O(n^2 \log n)$ will still work.

n = 10.000

A square algorithm ($O(n^2)$) might still work.
But beware any big constants!

n = 1.000.000

$O(n \log n) = 13.000.000$
We might need a linear algorithm!

Task 2: Input Types

You should pay attention to the type and format of the input.

- Reading A fixed number of tokens;
- Reading Until a condition;
- Reading until the end of the input;

Task 2: Input Types

Reading a fixed number of tokens

The first line of the input says the total number of tokens/lines/cases to read;

```
#include <iostream>
using namespace std;

int main()
{
    int n;
    cin >> n;

    for (; n > 0; n--)
    {
       // Do something
    }
}
```

Task 2: Input Types

Conditional Ending

The problem specifies a special case to end the input.

Example – Request for Proposal:

The input ends with a line containing two zeroes.

```
int main()
{
    cin >> n >> p;
    cin.ignore(1000, '\n');
    // throws away the \n for a future getline
    while (n!=0 || p!=0)
    {
        // do something!
        cin >> n >> p;
        cin.ignore(1000, '\n');
    }
}
```

Task 2: Input Types

Reading until the end of file

Put your reading function (getline, cin) inside a conditional. The input goes on until the end of the file. No explicit terminating value.

Example: 100 – The N+1 Problem

```
#include<string>
int main()
{
    int a, b;
    while (cin >> a >> b;)
    {
        // Do something!
    }
}
```

Reading the Output – Correctness

The UVA judge will evaluate your code based on a simple *diff*.
Be very careful to write your output exactly like stated.

The Judge is like an angry client. It wants the output EXACTLY how it stated.

Easy to Miss errors in the output

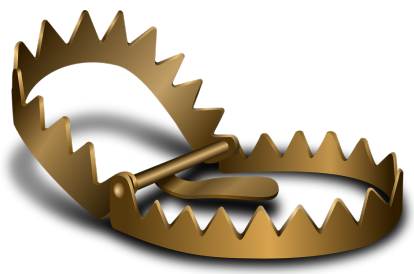
- Are the words uppercase, or lower case? (Case or case)
- Are there any mistakes in the words? (case or caes)
- Singular or plural? (1 hours or 1 hour)
- What is the precision of floating numbers? (3.051 or 3.05)
- Do you round up or round down? (3.62 $\rightarrow$ 3 or 4)
- Is there more than one answer?

Reading the problem: Identify Traps!

Be very careful to find traps in the problem:

Example: 3n+1 Problem

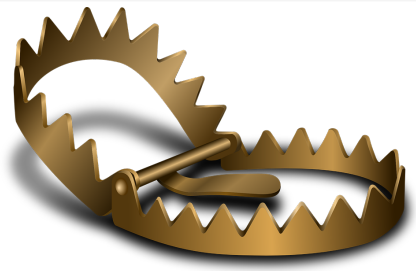
- The two numbers *i* and *j* in the input can come in any order.
- But the output must be in the same order as the input!



Reading the problem: Identify Traps!

Common Traps

- Negative numbers, zeros;
- Duplicated input, empty input;
- No solutions, multiple solutions;
- Other special cases;



Task 3: Choosing the algorithm

3.1 What type of problem it is?

- **Full Search:** Tests all possible solutions;
- **Greedy:** Break the solution, and try each best subsolution;
- **Simulation:** Execute a series of steps, record the results;
- **Dynamic Programming:** Table of partial solutions;
- **Counting:** Calculate the number of possible solutions;
- **Ad-hoc:** No fixed category;
- And many others;

First understand the problem; Then think; lastly code.

Task 3: Choosing the algorithm

Basic considerations for recursive and iterative algorithms:

- An algorithm with k-nested loops of about n iterations each has O(nk) complexity;
- A recursive algorithm with b recursive calls per level, and L levels, it should have O(bL) complexity;
- Use pruning to reduce the complexity of algorithms.
- An algorithm processing a $n * n$ matrix in O(k) per cell runs in $O(kn^2)$ time.

Task 3: Choosing the algorithm

Example of Pruning: The 8-queen problem

Pruning with a good data structure

- 8x8 matrix, with 1 for each queen: 64*63*62... = 1.78e+14
- 1 queen per column: 8*8*8*... = 1.67+e8
- 1 queen per column/row: 8*7*6... = 40320

You can prune even more by removing simmetries!

Task 4: Coding

Do you understand The Problem and The Algorithm?

NOW you can start writing your program.

If you start your program before you understand the solution, you will create many more bugs.

Task 4: Coding

Hint 1: “The Library”

Create a file with code examples that you often use.

- Input/output functions;
- Common data structures;
- Difficult algorithms;

Hint 2: Use paper

- Writing your idea on paper help you visualize;
- Sometimes you can find bugs this way;

Task 4: Coding

Hint 3: Programmer Efficiency

Everyone knows about CPU efficiency or Memory efficiency.

But Programmer Efficiency is very important too: Don't get tired/confused!

- Use standard library and macros;
- Your program just need to solve THIS problem;
- Use simple structures and algorithms;
- TDD mindset;

Task 5,6: Test and Hidden Data

The example data **is not** all the data!

Example Data

- Useful to test input/output;
- Read the example data to understand the problem;

Hidden Data

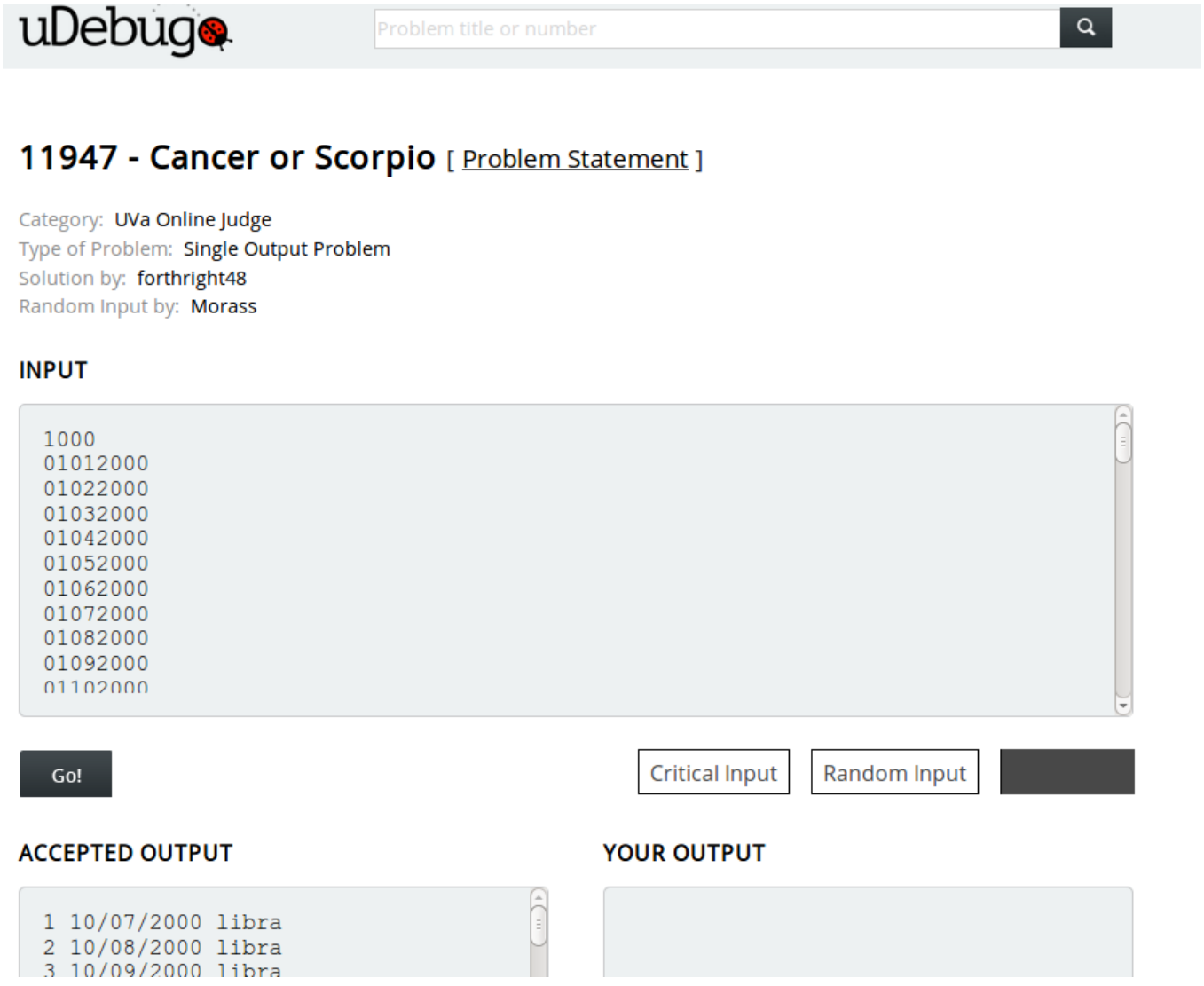
- Used by the UVA judge;
- Contain bigger data sets;
- Includes special cases;

Generate your own set of hidden data before submitting!

What data to generate?

- Datas with multiple entries (to check for initialization);
- Datas with maximum size (can be trivial cases);
- Random data;
- Border cases (maximum and minimum values in input range);
- Worst cases (depends on the problem);

The uDebug Website



To Summarize

Mental framework to solve problems:

1. Read the problem carefully to avoid traps;
2. Think of the algorithm and data structure;
3. Keep the size of the problem in mind;
4. Keep your code simple;
5. Create special test cases;

Now go solve the other problems!

Thanks for Listening!

Any questions?

BONUS I – The most expensive program ever?

Ackermann's Function

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0, n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0, n > 0 \end{cases}$$

- $A(0,n) = n+1$
- $A(1,n) = n+3$
- $A(2,n) = 2n+3$
- $A(3,n) = 2^{n+3} - 3$!
- $A(4,n) = 2^{2^{\cdots^{2}}} - 3$!!! (exponential tower of n+3)
- $A(5,n) =$!!!!!!!!!!

BONUS II – Typing Fast

Practice typing in one of many online typing challenges.

- `www.typingtest.com`
- `https://typing.io` – for programmers!