Contest Strategy

Ryan Jian

November 15, 2013

Pieced together mostly from past lectures by Alex and Andre and some of my own advice.

1 Introduction

SCT focuses on preparation for the USA Computing Olympiad (USACO) Contest programming is very different from "regular" programming in many ways. For one, contest programming focuses on algorithms and data structures. Hence, a significant portion of a contest should be devoted to thinking about the problems. How much depends on time constraints, but on a three hour contest it is usually advisable to spend a good 30-45 minutes thinking.

2 The Contest

2.1 Reading

- The problems are not intended to be sorted by increasing difficulty. Therefore the **FIRST** thing you should do is carefully **READ ALL OF THE PROBLEMS**. Work out the sample cases on paper if you need to, just make sure you understand all the details of each problem. Afterwards, look for problems that are similar to those you have solved before or problems that you can easily see how to solve.

2.2 Solving

- What is the naive/brute force solution?
- Look at the bounds on the size of the input along with the runtime of the naive/brute force solution. These can be good clues for a correct solution to the problem. A good rule of thumb is that your program can perform around 10^8 operations per second.
- What is the simplest algorithm you can come up with that will run within the time and memory constraints?
- If you're stuck on a problem, ask yourself the following questions: Have you seriously tried to use the different techniques that you know (DP, greedy, binary search, graph theory, network flow, divide and conquer, line-sweep, string matching, math, etc.). Can you solve a more restricted version of the problem? Does the problem have an invariant (some property that doesn't change)? Can you restate the problem in another way (i.e. a mathematical expression). Does the problem have an inverse that's easier to compute? Have you fully exploited small bounds? (not necessarily explicitly stated in the problem).
- Come up with tricky cases to try and break your algorithm.
- Try to solve problems completely; one full solution and two hacks are better than three bad solutions. That being said, if you can get partial credit solutions, by all means submit them!

2.3 Coding

- The goal is to code a CORRECT program as quickly as possible. As a result there are a number of coding guidelines you should follow. Good contest code is:
  - **legible**: There is nothing more frustrating than debugging a program that you cannot comprehend. Name your variables logically, use white-space strategically, and add comments when necessary.
  - **concise**: This doesn't mean squeezing all your into as few lines as possible. However, you should avoid coding more than you have to. USACO solutions tend to be very short and pretty.
  - **modular**: Modular code has more subroutines, which can making debugging easier.

  The best way to improve on these three aspects is to read other people's solutions.
- It is a good idea to test code as you're writing it - it's easier to check small parts than to debug a huge multi-part program.
- Check if the input can cause integer overflow (Does your program ever work with numbers larger or smaller than what an `int` can hold?)?
- Check to make sure your arrays are large enough.
- Java Tips
  - ~~Switch to C++~~
  - Try to handle exceptions without try-catch statements.
  - Use `BufferedReader` and `StringTokenizer` over Scanner.

    ```
    BufferedReader input = new BufferedReader (new FileReader ("filename.in"));
    StringTokenizer st = new StringTokenizer (input.readLine(), " ");
    String string = st.nextToken();
    int number = Integer.parseInt(st.nextToken());
    ```
  - Use `PrintWriter` for output. Remember to close the output file.
  - Remember to have `System.exit(0)` at the end of your program.
  - Do not bother adhering to the principles of object oriented programming, that is, avoid polymorphism, abstract classes, interfaces, etc.
  - Learn how to use the functions and data structures in `java.util`.
- C/C++ Tips
  - Avoid pointers and performing your own dynamic memory allocation (use STL instead).
  - Initialize your arrays globally so they are automatically set to default values rather than to random garbage.
  - Use `sort()` from `<algorithm>`.

2.4 Testing

- Try to leave the last 30-45 minutes of the contest to test your solutions.
- Write a slow but obviously correct solution to the problem, and write a script to generate test cases.
- Test your fast program against your slow program's output for a few thousand test cases or so with `diff`. For example:

```
#!/bin/bash
for i in ‘seq 1 10000‘
do
    ./prog.generate
    ./prog.slow
```

```
    mv prog.out prog.slow.out
    ./prog
    if diff prog.out prog.slow.out
    then
        echo $i OKAY
    else
        echo $i FAIL
        exit
    fi
done
```

- It is usually faster to code test case generators in higher level programming languages like Python because they require significantly less code, especially because they tend to have many built in functions for string manipulation (generating an input file is just generating some ascii string and writing it to a file).
- Test edge cases!

2.5 Debugging

- Ideally you don't want to debug at all, so try not to create bugs by testing components of your program as you write them.
- For C/C++ use **assert()** and consider gdb for segmentation faults.
- If you've spent 45 minutes debugging a program, look at the clock and see if it will be more beneficial for you to keep going or to cut your losses and move on.

3 Additional Tips

- Keep a log of each contest - this allows you to see how much time you've spent in each area, and you can refer back to the log to see how you could have improved your strategy.
- The rules allow you to use code from books, the Tnternet, and other resources, **SO LONG AS YOU CITE IT**. Just add a comment with a link to the website you got the snippet of code from and a short explanation.
- Don't cheat. Unlike at school they frequently catch people.

4 Sample Programs

Below are sample solutions to (SCT Grader, easyprob): Given N and a list of N numbers, find and print their sum.

4.1 C++

```
#include <iostream>
#include <fstream>
// #include <algorithm> // sorting functions and binary search
// #include <vector> // dynamic arrays
using namespace std;

ifstream fin("easyprob.in");
ofstream fout("easyprob.out");

int N;
int sum = 0;
int main() {
    fin >> N;
    for(int i = 0; i < N; i++) {
        int x;
        fin >> x;
        sum += x;
    }
    fout << sum << endl;
    return 0;
}
```

4.2 Java

```
// The two standard imports that cover most useful classes.
import java.util.*;
import java.io.*;
// import java.math.*; // BigInteger

public class easyprob { // always use the problem name here
    // throw an IOException here so there is no need to handle errors later
    public static void main(String[] args) throws IOException {
        // These two classes are rather fast for input and output.
        BufferedReader input = new BufferedReader(new FileReader("easyprob.in"));
        PrintWriter output = new PrintWriter(new BufferedWriter(new FileWriter("easyprob.out")));
        int N = Integer.parseInt(input.readLine());
        int sum = 0;
        for(int i = 0; i < N; i++) {
            int next = Integer.parseInt(input.readLine());
            sum += next;
        }
        output.println(sum);
        output.close();
        System.exit(0);
    }
}
```