AN INTRODUCTION TO THE USA COMPUTING OLYMPIAD

Darren Yao

2020

C++ Edition

Contents

Part I

Basic Techniques

Chapter 1

The Beginning

1.1 Competitive Programming

Welcome to the world of competitive programming! If you've had some basic programming experience with C++ (perhaps at the level of an introductory course), and are interested in competitive programming, then this book is for you. (If your primary language is Java, we also have a Java edition of this book; please refer to that instead). If you currently do not know how to code, there are numerous resources available online to help you learn.

This book aims to guide you through your competitive programming journey by providing a framework in which to learn the important contest topics. From competitive programming, not only do you improve at programming, but you improve your problem-solving skills which will help you in other areas. If at any point you have questions, feedback, or notice any mistakes, please contact me at darren.yao@gmail.com. Best of luck, and enjoy the ride!

The goal of competitive programming is to write code to solve given problems quickly. These problems are not open problems; they are problems that are designed to be solved in the short timeframe of a contest, and have already been solved by the problem writer and testers. In general, each problem in competitive programming is solved by a two-step process: coming up with the algorithm, and then implementing it into working code. The degree of mathematics knowledge varies from contest to contest, but generally the level of mathematics required is relatively elementary, and we will review important topics in this book.

A contest generally lasts for several hours, and consists of a set of problems. For each problem, when you complete your code, you submit it to a grader, which checks the answers calculated by the your program against a set of predetermined test cases. For each problem, you are given a time limit and a memory limit that your program must satisfy. Grading varies between contests; sometimes there is partial credit for passing some cases, while other times grading is all-or-nothing. For those of you with experience in software development, note that competitive programming is quite different, as the goal is to write programs that compute the correct answer, run quickly, and can be implemented quickly. Note that nowhere was maintainability of code mentioned. This means that you should throw away everything you know about traditional code writing; you don't need to bother documenting your code, because it only needs to be readable to you, during the contest.

1.2 Contests and Resources

The USA Computing Olympiad is a national programming competition that occurs four times a year, with December, January, February, and US Open contests. The regular contests are four hours long, and the US Open is five hours long. Each contest contains three problems. Solutions are evaluated and scored against a set of predetermined test cases that are not visible to the student. Scoring is out of 1000 points, with each problem being weighted equally. There are four divisions of contests: Bronze, Silver, Gold, and Platinum. After each contest, students who meet the contest-dependent cutoff for promotion will compete in the next division for future contests.

While this book is primarily focused on the USACO, CodeForces is another contest programming platform that many students use for practice. CodeForces holds 2-hour contests very frequently, which are more focused on fast solving compared to USACO. However, we do think CodeForces is a valuable training platform, so many exercises and problems will come from there. We encourage you to create a CodeForces account and solve the provided problems there. CodeForces submissions are all-or-nothing; unlike USACO, there is no partial credit and you only receive credit for a problem if you pass *all* of the test cases.

We will also include some exercises from Antti Laaksonen's website CSES. It contains a selection of standard problems that you can use to learn and practice well-known algorithms and techniques. You should note that CSES's grader is very slow, so don't worry if you encounter a Time Limit Exceeded verdict; as long as you pass the majority of test cases within the time limit, and your time complexity is reasonable, you can consider the problem solved, and move on.

1.3 Competitive Programming Practice

Reaching a high level in competitive programming requires dedication and motivation. For many people, their practice is inefficient because they do problems that are too easy, too hard, or simply of the wrong type. This book aims to correct that by providing comprehensive problem sets for each topic covered on the USA Computing Olympiad, as well as an extensive selection of problems across all topics in the final chapter.

In the lower divisions, most problems use relatively elementary algorithms; the main challenge is deciding which algorithm to use, and implementing it correctly. In a contest, you should spend the bulk of your time thinking about the problem and coming up with the algorithm, rather than typing code. Thus, you should practice your implementation skills, so that during the contest, you can implement the algorithm quickly and correctly, without resorting to debugging.

On Exercises and Practice Problems

You improve at competitive programming by solving problems, so we strongly recommend that you make use of the included exercises in each section before moving on. Some of the problems will be easy, and some of them will be hard. This is because problems that you practice with should be of the appropriate difficulty. You don't necessarily need to complete all the exercises at the end of each chapter, just do what you think is right for you. A problem at the right level of difficulty should be one of two types: either you struggle with the problem for a while before coming up with a working solution, or you miss it slightly and need to consult the solution for some small part. If you instantly come up with the solution, a problem is likely too easy, and if you're missing multiple steps, it might be too hard.

In general, especially on harder problems, I think it's fine to read the solution relatively early on, as long as you're made several different attempts at it and you can learn effectively from the solution.

- On a bronze problem, read the solution after 15-20 minutes of no meaningful progress, after you've exhausted every idea you can think of.
- On a silver problem, read the solution after 30-40 minutes of no meaningful progress.

When you get stuck and consult the solution, you should not read the entire solution at once, and you certainly shouldn't look at the solution code. Instead, it's better to read the solution step by step until you get unstuck, at which point you should go back and finish the problem, and implement it yourself. Reading the full solution or its code should be seen as a last resort.

IDEs and Text Editors

Here's some IDEs and text editors often used by competitive programmers:

- Java: Visual Studio Code or IntelliJ/Eclipse
- C++: Visual Studio Code, CodeBlocks, vim/gvim, Sublime Text.
- Do not use online IDEs that display your code publicly, like the free version of ideone. This allows other users to copy your code, and you may get flagged for cheating.

1.4 About This Book

This book aims to prepare students for the Bronze and Silver division of the USACO, with the goal of qualifying for Gold. We will do this by covering all the necessary algorithms, data structures, and skills to pass the Bronze and Silver contests. Many examples and practice problems have been provided; these are the most important part of studying competitive programming, so make sure you pay careful attention to the examples and solve the practice problems, which usually come from previous USACO contests. This book is intended for those who have some programming experience – Basic knowledge of C++ at the level of an introductory class is expected. This book begins with some necessary background knowledge, which is then followed by lessons on common topics that appear on the Bronze and Silver divisions of USACO, and then examples. At the end of each chapter will be a set of problems from USACO, CodeForces, and CSES, where you can practice what you've learned in the chapter.

The primary purpose of this book is to compile all of the topics needed for a beginner in one book, and provide all the resources needed, to make the process of studying for contests easier.

Chapter 2

Elementary Techniques

2.1 Input and Output

In CodeForces and CSES, input and output are standard, meaning that using the library `<iostream>` suffices.

However, in USACO, input is read from a file called `problemname.in`, and printing output to a file called `problemname.out`. Note that you'll have to rename the `.in` and `.out` files. You will need the `<cstdio>` or the `<fstream>` library. Essentially, replace every instance of the word *template* in the word below with the input/output file name, which should be given in the problem.

In order to test a program, create a file called `problemname.in`, and then run the program. The output will be printed to `problemname.out`.

Below, we have included C++ example code for input and output. We use `using namespace std;` so that we don't have to preface standard library functions with `std::` each time we use them. These templates are kept short so that you can type them each time, as prewritten code is no longer allowed in USACO as of the 2020-2021 season.

For USACO:

If `<cstdio>` is used:

```
#include <cstdio>

using namespace std;

int main() {
    freopen("template.in", "r", stdin);
    freopen("template.out", "w", stdout);
}
```

If `<fstream>` is used (note that if you use `<fstream>`, you must replace `cin` and `cout` with `fin` and `fout`):

```
#include <fstream>

using namespace std;

int main() {
    ifstream fin("template.in");
    ofstream fout("template.out");
}
```

For CodeForces, CSES, and other contests that use standard input and output, simply use the standard input / output from `<iostream>`.

When using C++, arrays should be declared globally if at all possible. This avoids the common issue of initialization to garbage values. If you declare an array locally, you'll need to initialize the values to zero.

2.2 Data Types

There are several main data types that are used in contests: 32-bit and 64-bit integers, floating point numbers, booleans, characters, and strings.

The 32-bit integer supports values between $-2\,147\,483\,648$ and $2\,147\,483\,647$, which is roughly equal to $\pm\ 2 \times 10^9$. If the input, output, or *any intermediate values used in calculations* exceed the range of a 32-bit integer, then a 64-bit integer must be used. The range of the 64-bit integer is between $-9\,223\,372\,036\,854\,775\,808$ and $9\,223\,372\,036\,854\,775\,807$ which is roughly equal to $\pm\ 9 \times 10^{18}$. Contest problems are usually set such that the 64-bit integer is sufficient. If it's not, the problem will ask for the answer modulo m, instead of the answer itself, where m is a prime. In this case, make sure to use 64-bit integers, and take the remainder of x modulo m after every step using `x %= m;`.

Floating point numbers are used to store decimal values. It is important to know that floating point numbers are not exact, because the binary architecture of computers can only store decimals to a certain precision. Hence, we should always expect that floating point numbers are slightly off. Contest problems will accommodate this by either asking for the greatest integer less than 10^k times the value, or will mark as correct any output that is within a certain ϵ of the judge's answer.

Boolean variables have two possible states: true and false. We'll usually use booleans to mark whether a certain process is done, and arrays of booleans to mark which components of an algorithm have finished.

Character variables represent a single Unicode character. They are returned when you access the character at a certain index within a string. Characters are represented using the ASCII standard, which assigns each character to a corresponding integer; this allows us to do arithmetic with them, for example, `cout << ('f' - 'a');` will print `5`.

Strings are stored as an array of characters. You can easily access the character at a certain index and take substrings of the string. String problems on USACO are generally very easy and don't involve any special data structures.