ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-3-319-72546-8 ISBN 978-3-319-72547-5 (eBook)
https://doi.org/10.1007/978-3-319-72547-5

Library of Congress Control Number: 2017960923

Printed on acid-free paper

This Springer imprint is published by the registered company is Springer International Publishing AG part of Springer Nature.
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

The purpose of this book is to give you a comprehensive introduction to modern competitive programming. It is assumed that you already know the basics of programming, but previous background in algorithm design or programming contests is not necessary. Since the book covers a wide range of topics of various difficulty, it suits both for beginners and more experienced readers.

Programming contests already have a quite long history. The *International Collegiate Programming Contest* for university students was started during the

Contents

This chapter shows what competitive programming is about, outlines the contents of the book, and discusses additional learning resources.

Section 1.1 goes through the elements of competitive programming, introduces a selection of popular programming contests, and gives advice on how to practice competitive programming.

Section 1.2 discusses the goals and topics of this book, and briefly describes the contents of each chapter.

Section 1.3 presents the CSES Problem Set, which contains a collection of practice problems. Solving the problems while reading the book is a good way to learn competitive programming.

Section 1.4 discusses other books related to competitive programming and the design of algorithms.

1.1 What is Competitive Programming?

Competitive programming combines two topics: the design of algorithms and the implementation of algorithms.

Design of Algorithms The core of competitive programming is about inventing efficient algorithms that solve well-defined computational problems. The design of algorithms requires problem solving and mathematical skills. Often a solution to a problem is a combination of well-known methods and new insights.

Mathematics plays an important role in competitive programming. Actually, there are no clear boundaries between algorithm design and mathematics. This book has been written so that not much background in mathematics is needed. The appendix of the book reviews some mathematical concepts that are used throughout the book,

A. Laaksonen, *Guide to Competitive Programming*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-72547-5_1

such as sets, logic, and functions, and the appendix can be used as a reference when reading the book.

Implementation of Algorithms In competitive programming, the solutions to problems are evaluated by testing an implemented algorithm using a set of test cases. Thus, after coming up with an algorithm that solves the problem, the next step is to correctly implement it, which requires good programming skills. Competitive programming greatly differs from traditional software engineering: programs are short (usually at most some hundreds of lines), they should be written quickly, and it is not needed to maintain them after the contest.

At the moment, the most popular programming languages used in contests are C++, Python, and Java. For example, in Google Code Jam 2017, among the best 3,000 participants, 79% used C++, 16% used Python, and 8% used Java. Many people regard C++ as the best choice for a competitive programmer. The benefits of

type of problems.

Another important observation is that most programming contest problems can be solved using simple and short algorithms, but the difficult part is to invent the algorithm. Competitive programming is not about learning complex and obscure algorithms by heart, but rather about learning problem solving and ways to approach difficult problems using simple tools.

Finally, some people despise the implementation of algorithms: it is fun to design algorithms but boring to implement them. However, the ability to quickly and correctly implement algorithms is an important asset, and this skill can be practiced. It is a bad idea to spend most of the contest time for writing code and finding bugs, instead of thinking of how to solve problems.

1.2 About This Book

The *IOI Syllabus* [15] regulates the topics that may appear at the International Olympiad in Informatics, and the syllabus has been a starting point when selecting topics for this book. However, the book also discusses some advanced topics that are (as of 2017) excluded from the IOI but may appear in other contests. Examples of such topics are maximum flows, nim theory, and suffix arrays.

While many competitive programming topics are discussed in standard algorithms textbooks, there are also differences. For example, many textbooks focus on implementing sorting algorithms and fundamental data structures from scratch, but this knowledge is not very relevant in competitive programming, because standard library functionality can be used. Then, there are topics that are well known in the competitive programming community but rarely discussed in textbooks. An example of such a topic is the segment tree data structure that can be used to solve a large number of problems that would otherwise require tricky algorithms.

One of the purposes of this book has been to *document* competitive programming techniques that are usually only discussed in online forums and blog posts. Whenever possible, scientific references have been given for methods that are specific to competitive programming. However, this has not often been possible, because many techniques are now part of competitive programming *folklore* and nobody knows who has originally discovered them.

The structure of the book is as follows:

- Chapter 2 reviews features of the C++ programming language, and then discusses recursive algorithms and bit manipulation.
- Chapter 3 focuses on efficiency: how to create algorithms that can quickly process large data sets.
- Chapter 4 discusses sorting algorithms and binary search, focusing on their applications in algorithm design.
- Chapter 5 goes through a selection of data structures of the C++ standard library, such as vectors, sets, and maps.
- Chapter 6 introduces an algorithm design technique called dynamic programming,

This problem is a simple simulation problem, which does not require much thinking. Here is a possible way to solve the problem in C++:


```
#include <iostream>

using namespace std;

int main() {
    int n;
    cin >> n;
    while (true) {
        cout << n << " ";
        if (n == 1) break;
        if (n%2 == 0) n /= 2;
        else n = n*3+1;
    }
    cout << "\n";
}
```

The code first reads in the input number n, and then simulates the algorithm and prints the value of n after each step. It is easy to test that the algorithm correctly handles the example case $n = 3$ given in the problem statement.

Now is time to *submit* the code to CSES. Then the code will be compiled and tested using a set of test cases. For each test case, CSES will tell us whether our code passed it or not, and we can also examine the input, the expected output, and the output produced by our code.

After testing our code, CSES gives the following report:

test	verdict	time (s)
#1	ACCEPTED	0.06 / 1.00
#2	ACCEPTED	0.06 / 1.00
#3	ACCEPTED	0.07 / 1.00
#4	ACCEPTED	0.06 / 1.00
#5	ACCEPTED	0.06 / 1.00
#6	TIME LIMIT EXCEEDED	- / 1.00
#7	TIME LIMIT EXCEEDED	- / 1.00
#8	WRONG ANSWER	0.07 / 1.00
#9	TIME LIMIT EXCEEDED	- / 1.00
#10	ACCEPTED	0.06 / 1.00

Programming Techniques

2

This chapter presents some of the features of the C++ programming language that are useful in competitive programming, and gives examples of how to use recursion and bit operations in programming.

Section 2.1 discusses a selection of topics related to C++, including input and output methods, working with numbers, and how to shorten code.

Section 2.2 focuses on recursive algorithms. First we will learn an elegant way to generate all subsets and permutations of a set using recursion. After this, we will use backtracking to count the number of ways to place n non-attacking queens on an $n \times n$ chessboard.

Section 2.3 discusses the basics of bit operations and shows how to use them to represent subsets of sets.

2.1 Language Features

A typical C++ code template for competitive programming looks like this:

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    // solution comes here
}
```

The `#include` line at the beginning of the code is a feature of the g++ compiler that allows us to include the entire standard library. Thus, it is not needed to separately

A. Laaksonen, *Guide to Competitive Programming*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-72547-5_2

Note that the newline "\n" works faster than endl, because endl always causes a flush operation.

The C functions scanf and printf are an alternative to the C++ standard streams. They are usually slightly faster, but also more difficult to use. The following code reads two integers from the input:

```
int a, b;
scanf("%d %d", &a, &b);
```

The following code prints two integers:

```
int a = 123, b = 456;
printf("%d %d\n", a, b);
```

Sometimes the program should read a whole input line, possibly containing spaces. This can be accomplished by using the getline function:

```
string s;
getline(cin, s);
```

If the amount of data is unknown, the following loop is useful:

```
while (cin >> x) {
    // code
}
```

This loop reads elements from the input one after another, until there is no more data available in the input.

In some contest systems, files are used for input and output. An easy solution for this is to write the code as usual using standard streams, but add the following lines to the beginning of the code:

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```

After this, the program reads the input from the file "input.txt" and writes the output to the file "output.txt".

2.1.2 Working with Numbers

Integers The most used integer type in competitive programming is int, which is a 32-bit type[1] with a value range of $-2^{31} \ldots 2^{31}-1$ (about $-2 \cdot 10^9 \ldots 2 \cdot 10^9$). If the type int is not enough, the 64-bit type long long can be used. It has a value

However, this is only needed when there are subtractions in the code, and the remainder may become negative.

Floating Point Numbers In most competitive programming problems, it suffices to use integers, but sometimes floating point numbers are needed. The most useful floating point types in C++ are the 64-bit `double` and, as an extension in the g++ compiler, the 80-bit `long double`. In most cases, `double` is enough, but `long double` is more accurate.

The required precision of the answer is usually given in the problem statement. An easy way to output the answer is to use the `printf` function and give the number of decimal places in the formatting string. For example, the following code prints the value of x with 9 decimal places:

```
printf("%.9f\n", x);
```

A difficulty when using floating point numbers is that some numbers cannot be represented accurately as floating point numbers, and there will be rounding errors. For example, in the following code, the value of x is slightly smaller than 1, while the correct value would be 1.

```
double x = 0.3*3+0.1;
printf("%.20f\n", x); // 0.99999999999999988898
```

It is risky to compare floating point numbers with the == operator, because it is possible that the values should be equal but they are not because of precision errors. A better way to compare floating point numbers is to assume that two numbers are equal if the difference between them is less than ε, where ε is a small number. For example, in the following code $\varepsilon = 10^{-9}$:


```
if (abs(a-b) < 1e-9) {
    // a and b are equal
}
```

Note that while floating point numbers are inaccurate, integers up to a certain limit can still be represented accurately. For example, using `double`, it is possible to accurately represent all integers whose absolute value is at most 2^{53}.

2.1.3 Shortening Code

Type Names The command `typedef` can be used to give a short name to a data type. For example, the name `long long` is long, so we can define a short name `ll` as follows:

```
typedef long long ll;
```

2.2 Recursive Algorithms

Recursion often provides an elegant way to implement an algorithm. In this section, we discuss recursive algorithms that systematically go through candidate solutions to a problem. First, we focus on generating subsets and permutations and then discuss the more general backtracking technique.

2.2.1 Generating Subsets

Our first application of recursion is generating all subsets of a set of n elements. For example, the subsets of $\{1, 2, 3\}$ are $\emptyset$, $\{1\}$, $\{2\}$, $\{3\}$, $\{1, 2\}$, $\{1, 3\}$, $\{2, 3\}$, and $\{1, 2, 3\}$. The following recursive function `search` can be used to generate the subsets. The function maintains a vector


```
vector<int> subset;
```

that will contain the elements of each subset. The search begins when the function is called with parameter 1.

```
void search(int k) {
    if (k == n+1) {
        // process subset
    } else {
        // include k in the subset
        subset.push_back(k);
        search(k+1);
        subset.pop_back();
        // don't include k in the subset
        search(k+1);
    }
}
```

When the function `search` is called with parameter k, it decides whether to include the element k in the subset or not, and in both cases, then calls itself with parameter $k+1$. Then, if $k = n+1$, the function notices that all elements have been processed and a subset has been generated.

Figure 2.1 illustrates the generation of subsets when $n = 3$. At each function call, either the upper branch (k is included in the subset) or the lower branch (k is not included in the subset) is chosen.

2.2.2 Generating Permutations

Next we consider the problem of generating all permutations of a set of n elements

2.2.3 Backtracking

A *backtracking* algorithm begins with an empty solution and extends the solution step by step. The search recursively goes through all different ways how a solution can be constructed.

As an example, consider the problem of calculating the number of ways n queens can be placed on an $n \times n$ chessboard so that no two queens attack each other. For example, Fig. 2.2 shows the two possible solutions for $n = 4$.

The problem can be solved using backtracking by placing queens on the board row by row. More precisely, exactly one queen will be placed on each row so that no queen attacks any of the queens placed before. A solution has been found when all n queens have been placed on the board.

For example, Fig. 2.3 shows some partial solutions generated by the backtracking algorithm when $n = 4$. At the bottom level, the three first configurations are illegal, because the queens attack each other. However, the fourth configuration is valid, and it can be extended to a complete solution by placing two more queens on the board. There is only one way to place the two remaining queens.

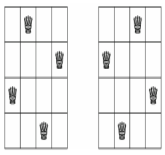

Fig. 2.2 The possible ways to place 4 queens on a 4×4 chessboard