Handbook of Competitive Programming

Antti Laaksonen

December 29, 2016

Contents

Preface **ix**

I Perusasiat 1

1 Introduction **3**
1.1 Programming languages 3
1.2 Input and output 4
1.3 Handling numbers 6
1.4 Shortening code 8
1.5 Mathematics 9

2 Time complexity **15**
2.1 Laskusäännöt 15
2.2 Vaativuusluokkia 18
2.3 Tehokkuuden arviointi 19
2.4 Suurin alitaulukon summa 19

3 Sorting **23**
3.1 Järjestämisen teoriaa 23
3.2 Järjestäminen C++:ssa 27
3.3 Binäärihaku 30

4 Data structures **33**
4.1 Dynaaminen taulukko 33
4.2 Joukkorakenne 35
4.3 Hakemisto 36
4.4 Iteraattorit ja välit 37
4.5 Muita tietorakenteita 39
4.6 Vertailu järjestämiseen 42

5 Complete search **45**
5.1 Osajoukkojen läpikäynti 45
5.2 Permutaatioiden läpikäynti 47
5.3 Peruuttava haku 48
5.4 Haun optimointi 49
5.5 Puolivälihaku 52

6 Greedy algorithms **55**
6.1 Kolikkotehtävä . . . 55
6.2 Aikataulutus . . . 56
6.3 Tehtävät ja deadlinet . . . 58
6.4 Keskiluvut . . . 59
6.5 Tiedonpakkaus . . . 60

7 Dynamic programming **63**
7.1 Kolikkotehtävä . . . 63
7.2 Pisin nouseva alijono . . . 68
7.3 Reitinhaku ruudukossa . . . 69
7.4 Repunpakkaus . . . 70
7.5 Editointietäisyys . . . 71
7.6 Laatoitukset . . . 73

8 Amortized analysis **75**
8.1 Kahden osoittimen tekniikka . . . 75
8.2 Lähin pienempi edeltäjä . . . 78
8.3 Liukuvan ikkunan minimi . . . 79

9 Range queries **81**
9.1 Staattisen taulukon kyselyt . . . 81
9.2 Binääri-indeksipuu . . . 84
9.3 Segmenttipuu . . . 86
9.4 Lisätekniikoita . . . 91

10 Bit manipulation **93**
10.1 Luvun bittiesitys . . . 93
10.2 Bittioperaatiot . . . 94
10.3 Joukon bittiesitys . . . 96
10.4 Dynaaminen ohjelmointi . . . 98

II Graph algorithms 101

11 Basics of graphs **103**
11.1 Käsitteitä . . . 103
11.2 Verkko muistissa . . . 106

12 Graph search **111**
12.1 Syvyyshaku . . . 111
12.2 Leveyshaku . . . 113
12.3 Sovelluksia . . . 115

13 Shortest paths **117**
13.1 Bellman–Fordin algoritmi . . . 117
13.2 Dijkstran algoritmi . . . 120
13.3 Floyd–Warshallin algoritmi . . . 123

14 Puiden käsittely **127**
14.1 Puun läpikäynti 128
14.2 Läpimitta 129
14.3 Solmujen etäisyydet 130
14.4 Binääripuut 131

15 Spanning trees **133**
15.1 Kruskalin algoritmi 134
15.2 Union-find-rakenne 137
15.3 Primin algoritmi 139

16 Directed graphs **141**
16.1 Topologinen järjestys 141
16.2 Dynaaminen ohjelmointi 143
16.3 Tehokas eteneminen 146
16.4 Syklin tunnistaminen 147

17 Strongly connectivity **149**
17.1 Kosarajun algoritmi 150
17.2 2SAT-ongelma 152

18 Tree queries **155**
18.1 Tehokas nouseminen 155
18.2 Solmutaulukko 156
18.3 Alin yhteinen esivanhempi 159

19 Paths and circuits **163**
19.1 Eulerin polku 163
19.2 Hamiltonin polku 167
19.3 De Bruijnin jono 168
19.4 Ratsun kierros 169

20 Network flows **171**
20.1 Ford–Fulkersonin algoritmi 172
20.2 Rinnakkaiset polut 176
20.3 Maksimiparitus 177
20.4 Polkupeitteet 180

III Advanced topics 183

21 Number theory **185**
21.1 Alkuluvut ja tekijät 185
21.2 Modulolaskenta 189
21.3 Yhtälönratkaisu 192
21.4 Muita tuloksia 193

22 Combinatorics **195**
22.1 Binomikerroin 196
22.2 Catalanin luvut 198
22.3 Inkluusio-ekskluusio 200
22.4 Burnsiden lemma 202
22.5 Cayleyn kaava 203

23 Matrices **205**
23.1 Laskutoimitukset 205
23.2 Lineaariset rekursioyhtälöt 208
23.3 Verkot ja matriisit 210

24 Probability **213**
24.1 Laskutavat 213
24.2 Tapahtumat 214
24.3 Satunnaismuuttuja 216
24.4 Markovin ketju 218
24.5 Satunnaisalgoritmit 219

25 Game theory **223**
25.1 Pelin tilat 223
25.2 Nim-peli 225
25.3 Sprague–Grundyn lause 226

26 String algorithms **231**
26.1 Trie-rakenne 231
26.2 Merkkijonohajautus 232
26.3 Z-algoritmi 235

27 Square root algorithms **239**
27.1 Eräkäsittely 240
27.2 Tapauskäsittely 241
27.3 Mo'n algoritmi 241

28 Segment trees revisited **243**
28.1 Laiska eteneminen 244
28.2 Dynaaminen toteutus 247
28.3 Tietorakenteet 249
28.4 Kaksiulotteisuus 250

29 Geometry **253**
29.1 Kompleksiluvut 254
29.2 Pisteet ja suorat 256
29.3 Monikulmion pinta-ala 259
29.4 Etäisyysmitat 260

30 Sweep line **263**
30.1 Janojen leikkauspisteet 264
30.2 Lähin pistepari 265
30.3 Konveksi peite 266

Preface

The purpose of this book is to give you a thorough introduction to competitive programming. The book assumes that you already know the basics of programming, but previous background on competitive programming is not needed.

The book is especially intended for high school students who want to learn algorithms and possibly participate in the International Olympiad in Informatics (IOI). The book is also suitable for university students and anybody else interested in competitive programming.

It takes a long time to become a good competitive programmer, but it is also an opportunity to learn a lot. You can be sure that you will learn a great deal about algorithms if you spend time reading the book and solving exercises.

The book is under continuous development. You can always send feedback about the book to `ahslaaks@cs.helsinki.fi`.

Part I

Perusasiat

Chapter 1

Introduction

Competitive programming combines two topics: (1) design of algorithms and (2) implementation of algorithms.

The **design of algorithms** consists of problem solving and mathematical thinking. Skills for analyzing problems and solving them using creativity is needed. An algorithm for solving a problem has to be both correct and efficient, and the core of the problem is often how to invent an efficient algorithm.

Theoretical knowledge of algorithms is very important to competitive programmers. Typically, a solution for a problem is a combination of well-known techniques and new insights. The techniques that appear in competitive programming also form the basis for the scientific research of algorithms.

The **implementation of algorithms** requires good programming skills. In competitive programming, the solutions are graded by testing an implemented algorithm using a set of test cases. Thus, it is not enough that the idea of the algorithm is correct, but the implementation has to be correct as well.

Good coding style in contests is straightforward and concise. The solutions should be written quickly, because there is not much time available. Unlike in traditional software engineering, the solutions are short (usually at most some hundreds of lines) and it is not needed to maintain them after the contest.

1.1 Programming languages

At the moment, the most popular programming languages in contests are C++, Python and Java. For example, in Google Code Jam 2016, among the best 3,000 participants, 73 % used C++, 15 % used Python and 10 % used Java[1]. Some participants also used several languages.

Many people think that C++ is the best choice for a competitive programmer, and C++ is nearly always available in contest systems. The benefits in using C++ are that it is a very efficient language and its standard library contains a large collection of data structures and algorithms.

On the other hand, it is good to master several languages and know the benefits of them. For example, if big integers are needed in the problem, Python

[1] `https://www.go-hero.net/jam/16`

can be a good choice because it contains a built-in library for handling big integers. Still, usually the goal is to write the problems so that the use of a specific programming language is not an unfair advantage in the contest.

All examples in this book are written in C++, and the data structures and algorithms in the standard library are often used. The standard is C++11 that can be used in most contests. If you can't program in C++ yet, now it is a good time to start learning.

C++ template

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

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

using namespace std;

int main() {
    // koodi tulee tähän
}
```

The `#include` line at the beginning is a feature in the `g++` compiler that allows to include the whole standard library. Thus, it is not needed to separately include libraries such as `iostream`, `vector` and `algorithm`, but they are available automatically.

The following `using` line determines that the standard library can be used directly in the code. Without the `using` line we should write, for example, `std::cout`, but now it is enough to write `cout`.

The code can be compiled using the following command:

```
g++ -std=c++11 -O2 -Wall code.cpp -o code
```

This command produces a binary file `code` from the source code `code.cpp`. The compiler obeys the C++11 standard (`-std=c++11`), optimizes the code (`-O2`) and shows warnings about possible errors (`-Wall`).

1.2 Input and output

In most contests, standard streams are used for reading input and writing output. In C++, the standard streams are `cin` for input and `cout` for output. In addition, the C functions `scanf` and `printf` can be used.

The input for the program usually consists of numbers and strings that are separated with spaces and newlines. They can be read from the `cin` stream as follows:

```
int a, b;
string x;
cin >> a >> b >> x;
```

This kind of code always works, assuming that there is at least one space or one newline between each element in the input. For example, the above code accepts both the following inputs:

```
123 456 apina
```

```
123   456
apina
```

The `cout` stream is used for output as follows:

```
int a = 123, b = 456;
string x = "apina";
cout << a << " " << b << " " << x << "\n";
```

Handling input and output is sometimes a bottleneck in the program. The following lines at the beginning of the code make input and output more efficient:

```
ios_base::sync_with_stdio(0);
cin.tie(0);
```

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

The C functions `scanf` and `printf` are an alternative to the C++ standard streams. They are usually a bit faster, but they are 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 line from the input, possibly with spaces. This can be accomplished using the `getline` function:

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

If the amount of data is unknown, the following loop can be handy:

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

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 code reads the input from the file "input.txt" and writes the output to the file "output.txt".

1.3 Handling numbers

Integers

The most popular integer type in competitive programming is `int`. This is a 32-bit type with value range $-2^{31} \ldots 2^{31}-1$, i.e., 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, with value range $-2^{63} \ldots 2^{63}-1$, i.e., about $-9 \cdot 10^{18} \ldots 9 \cdot 10^{18}$.

The following code defines a `long long` variable:

```
long long x = 123456789123456789LL;
```

The suffix `LL` means that the type of the number is `long long`.

A typical error when using the type `long long` is that the type `int` is still used somewhere in the code. For example, the following code contains a subtle error:

```
int a = 123456789;
long long b = a*a;
cout << b << "\n"; // -1757895751
```

Even though the variable `b` is of type `long long`, both numbers in the expression `a*a` are of type `int` and the result is also of type `int`. Because of this, the variable `b` will contain a wrong result. The problem can be solved by changing the type of `a` to `long long` or by changing the expression to `(long long)a*a`.

Usually, the problems are written so that the type `long long` is enough. Still, it is good to know that the g++ compiler also features an 128-bit type `__int128_t` with value range $-2^{127} \ldots 2^{127}-1$, i.e., $-10^{38} \ldots 10^{38}$. However, this type is not available in all contest systems.

Modular arithmetic

We denote by $x \bmod m$ the remainder when x is divided by m. For example, $17 \bmod 5 = 2$, because $17 = 3 \cdot 5 + 2$.

Sometimes, the answer for a problem is a very big integer but it is enough to print it "modulo m", i.e., the remainder when the answer is divided by m (for

example, "modulo 10^9+7"). The idea is that even if the actual answer may be very big, it is enough to use the types `int` and `long long`.

An important property of the remainder is that in addition, subtraction and multiplication, the remainder can be calculated before the operation:

$$\begin{aligned}(a+b) \bmod m &= (a \bmod m + b \bmod m) \bmod m \\ (a-b) \bmod m &= (a \bmod m - b \bmod m) \bmod m \\ (a \cdot b) \bmod m &= (a \bmod m \cdot b \bmod m) \bmod m\end{aligned}$$

Thus, we can calculate the remainder after every operation and the numbers will never become too large.

For example, the following code calculates $n!$, the factorial of n, modulo m:

```
long long x = 1;
for (int i = 2; i <= n i++) {
    x = (x*i)%m;
}
cout << x << "\n";
```

Usually, the answer should be always given so that the remainder is between $0 \ldots m-1$. However, in C++ and other languages, the remainder of a negative number can be negative. An easy way to make sure that this will not happen is to first calculate the remainder as usual and then add m if the result is negative:

```
x = x%m;
if (x < 0) x += m;
```

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

Floating point numbers

The usual floating point types in competitive programming 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. The easiest way is to use the `printf` function that can be given the number of decimal places. 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, but there will be rounding errors. For example, the result of the following code is surprising:

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

Because of a rounding error, the value of x is a bit less than 1, while the correct value would be 1.

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 due to rounding errors. A better way to compare floating point numbers is to assume that two numbers are equal if the difference between them is ε, where ε is a small number.

In practice, the numbers can be compared as follows ($\varepsilon = 10^{-9}$):

```
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 be still represented accurately. For example, using `double`, it is possible to accurately represent all integers having absolute value at most 2^{53}.

1.4 Shortening code

Short code is ideal in competitive programming, because the algorithm should be implemented as fast as possible. Because of this, competitive programmers often define shorter names for datatypes and other parts of code.

Type names

Using the command `typedef` it is possible to give a shorter name to a datatype. For example, the name `long long` is long, so we can define a shorter name `ll`:

```
typedef long long ll;
```

After this, the code

```
long long a = 123456789;
long long b = 987654321;
cout << a*b << "\n";
```

can be shortened as follows:

```
ll a = 123456789;
ll b = 987654321;
cout << a*b << "\n";
```

The command `typedef` can also be used with more complex types. For example, the following code gives the name `vi` for a vector of integers, and the name `pi` for a pair that contains two integers.

```
typedef vector<int> vi;
typedef pair<int,int> pi;
```

Macros

Another way to shorten the code is to define **macros**. A macro means that certain strings in the code will be changed before the compilation. In C++, macros are defined using the command `#define`.

For example, we can define the following macros:

```
#define F first
#define S second
#define PB push_back
#define MP make_pair
```

After this, the code

```
v.push_back(make_pair(y1,x1));
v.push_back(make_pair(y2,x2));
int d = v[i].first+v[i].second;
```

can be shortened as follows:

```
v.PB(MP(y1,x1));
v.PB(MP(y2,x2));
int d = v[i].F+v[i].S;
```

It is also possible to define a macro with parameters which makes it possible to shorten loops and other structures in the code. For example, we can define the following macro:

```
#define REP(i,a,b) for (int i = a; i <= b; i++)
```

After this, the code

```
for (int i = 1; i <= n; i++) {
    haku(i);
}
```

can be shortened as follows:

```
REP(i,1,n) {
    haku(i);
}
```

1.5 Mathematics

Mathematics plays an important role in competitive programming, and it is not possible to become a successful competitive programmer without good skills in mathematics. This section covers some important mathematical concepts and formulas that are needed later in the book.

Summakaavat

Jokaiselle summalle muotoa

$$\sum_{x=1}^{n} x^k = 1^k + 2^k + 3^k + \ldots + n^k$$

on olemassa laskukaava, kun k on jokin positiivinen kokonaisluku. Tällainen laskukaava on aina astetta $k+1$ oleva polynomi. Esimerkiksi

$$\sum_{x=1}^{n} x = 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

ja

$$\sum_{x=1}^{n} x^2 = 1^2 + 2^2 + 3^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}.$$

Aritmeettinen summa on summa, jossa jokaisen vierekkäisen luvun erotus on vakio. Esimerkiksi

$$3 + 7 + 11 + 15$$

on aritmeettinen summa, jossa vakio on 4. Aritmeettinen summa voidaan laskea kaavalla

$$\frac{n(a+b)}{2},$$

jossa summan ensimmäinen luku on a, viimeinen luku on b ja lukujen määrä on n. Esimerkiksi

$$3 + 7 + 11 + 15 = \frac{4 \cdot (3+15)}{2} = 36.$$

Kaava perustuu siihen, että summa muodostuu n luvusta ja luvun suuruus on keskimäärin $(a+b)/2$.

Geometrinen summa on summa, jossa jokaisen vierekkäisen luvun suhde on vakio. Esimerkiksi

$$3 + 6 + 12 + 24$$

on geometrinen summa, jossa vakio on 2. Geometrinen summa voidaan laskea kaavalla

$$\frac{bx-a}{x-1},$$

jossa summan ensimmäinen luku on a, viimeinen luku on b ja vierekkäisten lukujen suhde on x. Esimerkiksi

$$3 + 6 + 12 + 24 = \frac{24 \cdot 2 - 3}{2 - 1} = 45.$$

Geometrisen summan kaavan voi johtaa merkitsemällä

$$S = a + ax + ax^2 + \cdots + b.$$

Kertomalla molemmat puolet x:llä saadaan

$$xS = ax + ax^2 + ax^3 + \cdots + bx,$$

josta kaava seuraa ratkaisemalla yhtälön

$$xS - S = bx - a.$$

Geometrisen summan erikoistapaus on usein kätevä kaava

$$1 + 2 + 4 + 8 + \ldots + 2^{n-1} = 2^n - 1.$$

Harmoninen summa on summa muotoa

$$\sum_{x=1}^{n} \frac{1}{x} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}.$$

Yläraja harmonisen summan suuruudelle on $\log_2(n) + 1$. Summaa voi näet arvioida ylöspäin muuttamalla jokaista termiä $1/k$ niin, että k:ksi tulee alempi 2:n potenssi. Esimerkiksi tapauksessa $n = 6$ arvioksi tulee

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} \leq 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}.$$

Tämän seurauksena summa jakaantuu $\log_2(n) + 1$ osaan (1, $2 \cdot 1/2$, $4 \cdot 1/4$, jne.), joista jokaisen summa on enintään 1.

Joukko-oppi

Joukko on kokoelma alkioita. Esimerkiksi joukko

$$X = \{2, 4, 7\}$$

sisältää alkiot 2, 4 ja 7. Merkintä $\emptyset$ tarkoittaa tyhjää joukkoa. Joukon S koko eli alkoiden määrä on $|S|$. Esimerkiksi äskeisessä joukossa $|X| = 3$.

Merkintä $x \in S$ tarkoittaa, että alkio x on joukossa S, ja merkintä $x \notin S$ tarkoittaa, että alkio x ei ole joukossa S. Esimerkiksi äskeisessä joukossa

$$4 \in X \quad \text{ja} \quad 5 \notin X.$$

Uusia joukkoja voidaan muodostaa joukko-operaatioilla seuraavasti:

- **Leikkaus** $A \cap B$ sisältää alkiot, jotka ovat molemmissa joukoista A ja B. Esimerkiksi jos $A = \{1, 2, 5\}$ ja $B = \{2, 4\}$, niin $A \cap B = \{2\}$.
- **Yhdiste** $A \cup B$ sisältää alkiot, jotka ovat ainakin toisessa joukoista A ja B. Esimerkiksi jos $A = \{3, 7\}$ ja $B = \{2, 3, 8\}$, niin $A \cup B = \{2, 3, 7, 8\}$.
- **Komplementti** $\bar{A}$ sisältää alkiot, jotka eivät ole joukossa A. Komplementin tulkinta riippuu siitä, mikä on **perusjoukko** eli joukko, jossa on kaikki mahdolliset alkiot. Esimerkiksi jos $A = \{1, 2, 5, 7\}$ ja perusjoukko on $P = \{1, 2, \ldots, 10\}$, niin $\bar{A} = \{3, 4, 6, 8, 9, 10\}$.
- **Erotus** $A \setminus B = A \cap \bar{B}$ sisältää alkiot, jotka ovat joukossa A mutta eivät joukossa B. Huomaa, että B:ssä voi olla alkioita, joita ei ole A:ssa. Esimerkiksi jos $A = \{2, 3, 7, 8\}$ ja $B = \{3, 5, 8\}$, niin $A \setminus B = \{2, 7\}$.

Merkintä $A \subset S$ tarkoittaa, että A on S:n **osajoukko** eli jokainen A:n alkio esiintyy S:ssä. Joukon S osajoukkojen yhteismäärä on $2^{|S|}$. Esimerkiksi joukon $\{2, 4, 7\}$ osajoukot ovat

$\emptyset$, {2}, {4}, {7}, {2,4}, {2,7}, {4,7} ja {2,4,7}.

Usein esiintyviä joukkoja ovat

- $\mathbb{N}$ (luonnolliset luvut),
- $\mathbb{Z}$ (kokonaisluvut),
- $\mathbb{Q}$ (rationaaliluvut) ja
- $\mathbb{R}$ (reaaliluvut).

Luonnollisten lukujen joukko $\mathbb{N}$ voidaan määritellä tilanteesta riippuen kahdella tavalla: joko $\mathbb{N} = \{0,1,2,\ldots\}$ tai $\mathbb{N} = \{1,2,3,...\}$.

Joukon voi muodostaa myös säännöllä muotoa

$$\{f(n) : n \in S\},$$

missä $f(n)$ on jokin funktio. Tällainen joukko sisältää kaikki alkiot $f(n)$, jossa n on valittu joukosta S. Esimerkiksi joukko

$$X = \{2n : n \in \mathbb{Z}\}$$

sisältää kaikki parilliset kokonaisluvut.

Logiikka

Loogisen lausekkeen arvo on joko **tosi** (1) tai **epätosi** (0). Tärkeimmät loogiset operaatiot ovat $\neg$ (**negaatio**), $\wedge$ (**konjunktio**), $\vee$ (**disjunktio**), $\Rightarrow$ (**implikaatio**) sekä $\Leftrightarrow$ (**ekvivalenssi**). Seuraava taulukko näyttää operaatioiden merkityksen:

A	B	$\neg A$	$\neg B$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
0	0	1	1	0	0	1	1
0	1	1	0	0	1	1	0
1	0	0	1	0	1	0	0
1	1	0	0	1	1	1	1

Negaatio $\neg A$ muuttaa lausekkeen käänteiseksi. Lauseke $A \wedge B$ on tosi, jos molemmat A ja B ovat tosia, ja lauseke $A \vee B$ on tosi, jos A tai B on tosi. Lauseke $A \Rightarrow B$ on tosi, jos A:n ollessa tosi myös B on aina tosi. Lauseke $A \Leftrightarrow B$ on tosi, jos A:n ja B:n totuusarvo on sama.

Predikaatti on lauseke, jonka arvo on tosi tai epätosi riippuen sen parametreista. Yleensä predikaattia merkitään suurella kirjaimella. Esimerkiksi voimme määritellä predikaatin $P(x)$, joka on tosi tarkalleen silloin, kun x on alkuluku. Tällöin esimerkiksi $P(7)$ on tosi, kun taas $P(8)$ on epätosi.

Kvanttori ilmaisee, että looginen lauseke liittyy jollakin tavalla joukon alkioihin. Tavalliset kvanttorit ovat $\forall$ (**kaikille**) ja $\exists$ (**on olemassa**). Esimerkiksi

$$\forall x(\exists y(y < x))$$

tarkoittaa, että jokaiselle joukon alkiolle x on olemassa jokin joukon alkio y niin, että y on x:ää pienempi. Tämä pätee kokonaislukujen joukossa, mutta ei päde luonnollisten lukujen joukossa.

Yllä esitettyjen merkintöjä avulla on mahdollista esittää monenlaisia loogisia väitteitä. Esimerkiksi

$$\forall x((x > 2 \wedge \neg P(x)) \Rightarrow (\exists a(\exists b(x = ab \wedge a > 1 \wedge b > 1))))$$

tarkoittaa, että jos luku x on suurempi kuin 2 eikä ole alkuluku, niin on olemassa luvut a ja b, joiden tulo on x ja jotka molemmat ovat suurempia kuin 1. Tämä väite pitää paikkansa kokonaislukujen joukossa.

Funktioita

Funktio $\lfloor x \rfloor$ pyöristää luvun x alaspäin kokonaisluvuksi ja funktio $\lceil x \rceil$ pyöristää luvun x ylöspäin kokonaisluvuksi. Esimerkiksi

$$\lfloor 3/2 \rfloor = 1 \quad \text{ja} \quad \lceil 3/2 \rceil = 2.$$

Funktiot $\min(x_1, x_2, \ldots, x_n)$ ja $\max(x_1, x_2, \ldots, x_n)$ palauttavat pienimmän ja suurimman arvoista $x_1, x_2, \ldots, x_n$. Esimerkiksi

$$\min(1,2,3) = 1 \quad \text{ja} \quad \max(1,2,3) = 3.$$

Kertoma $n!$ määritellään

$$\prod_{x=1}^{n} x = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$$

tai vaihtoehtoisesti rekursiivisesti

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot (n-1)! \end{aligned}$$

Fibonaccin luvut esiintyvät monissa erilaisissa yhteyksissä. Ne määritellään seuraavasti rekursiivisesti:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

Ensimmäiset Fibonaccin luvut ovat

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

Fibonaccin lukujen laskemiseen on olemassa myös suljetun muodon kaava

$$f(n) = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n \sqrt{5}}.$$

Logaritmi

Luvun x **logaritmi** merkitään $\log_k(x)$, missä k on logaritmin kantaluku. Logaritmin määritelmän mukaan $\log_k(x) = a$ tarkalleen silloin, kun $k^a = x$.

Algoritmiikassa hyödyllinen tulkinta on, että logaritmi $\log_k(x)$ ilmaisee, montako kertaa lukua x täytyy jakaa k:lla, ennen kuin tulos on 1. Esimerkiksi $\log_2(32) = 5$, koska lukua 32 täytyy jakaa 2:lla 5 kertaa:

$$32 \to 16 \to 8 \to 4 \to 2 \to 1$$

Logaritmi tulee usein vastaan algoritmien analyysissa, koska monessa tehokkaassa algoritmissa jokin asia puolittuu joka askeleella. Niinpä logaritmin avulla voi arvioida algoritmin tehokkuutta.

Logaritmille pätee kaava

$$\log_k(ab) = \log_k(a) + \log_k(b),$$

josta seuraa edelleen

$$\log_k(x^n) = n \cdot \log_k(x).$$

Samoin logaritmille pätee

$$\log_k\left(\frac{a}{b}\right) = \log_k(a) - \log_k(b).$$

Lisäksi on voimassa kaava

$$\log_u(x) = \frac{\log_k(x)}{\log_k(u)},$$

minkä ansiosta logaritmeja voi laskea mille tahansa kantaluvulle, jos on keino laskea logaritmeja jollekin kantaluvulle.

Luvun x **luonnollinen logaritmi** $\ln(x)$ on logaritmi, jonka kantaluku on **Neperin luku** $e \approx 2{,}71828$.

Vielä yksi logaritmin ominaisuus on, että luvun x numeroiden määrä b-kantaisessa lukujärjestelmässä on $\lfloor \log_b(x) + 1 \rfloor$. Esimerkiksi luvun 123 esitys 2-järjestelmässä on 1111011 ja $\lfloor \log_2(123) + 1 \rfloor = 7$.

Part II

Graph algorithms

Part III

Advanced topics