Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A Short Introduction to C++ for the Algorithms Lab

Miloš Trujić

based on a version by

Florian Jug, Christoph Krautz, and Thomas Rast

Department of Computer Science, ETH Zürich

September, 2018

Acknowledgements

Several people have contributed immensely towards improving the presentation of this booklet with their ideas, feedback, corrections, and proofreading: Michael Hoffmann, Anders Martinsson, Felix Weissenberger, Manuel Wettstein, and Xun Zou. Thank you!

Contents

Chapter 1

Introduction

Welcome readers! In this booklet we highlight some C++ concepts, classes, objects, and functions that should be useful when implementing solutions to Algorithms Lab problems. We also discuss a few technical issues whose importance ranges from nice-to-know to absolutely essential in some cases. Therefore, as a student in the Algorithms Lab you should study this guide carefully. We tried to keep the material concise and focused. For more details, follow the links provided. If you find an error (even minor) or if you find some useful information to be missing, please do not hesitate to report to `algolab@lists.inf.ethz.ch`.

1.1 Brief Description

What you will find here

- An example of an exercise and the Judge feedback
- Input/output (streams), strings
- Fundamental data types
- Basic concepts, data structures and algorithms from the C++ standard library
    1. Containers
    2. Iterators
    3. Functions/working with iterators
- Pointers
- How to compile, run, and debug
- How to estimate the runtime of your algorithm

What you will *not* find here

- Introduction to C++ programming
- Control structures (`if`, `switch`, `for`, `while`, . . .)
- Memory management
- Object Oriented Programming in C++

1.2 Further Literature

- C++ Tutorial: `http://www.cplusplus.com/doc/tutorial/`
- C++ Reference: `https://en.cppreference.com/w/`
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT press, 2009.

1.3 Credits

The following references have been used while designing this booklet. Give credit where credit is due.

- The previous version of this booklet by Florian Jug, Christoph Krautz, and Thomas Rast
- `https://en.cppreference.com/w/`
- `https://github.com/gibsjose/cpp-cheat-sheet`

Chapter 2

The Basics

2.1 The Beginning: C++'s Adam & Eve

Let us begin by looking at a very simple problem.

Exercise 2.1.1 (Sum It!). Given $n \geq 1$ integers $a_0, a_1, \ldots, a_{n-1}$, calculate the sum $\sum_{i=0}^{n-1} a_i$.

Input The first line of the input contains the number $t \leq 10$ of test cases. Each of the t test cases is described as follows.

- It starts with a line that contains an integer `n`, denoting the number of integers to sum up, such that $0 \leq n \leq 10$.
- The following line contains n integers $\mathtt{a}_0 \ \ldots \ \mathtt{a}_{n-1}$, separated by a space, such that $-1000 \leq a_i \leq 1000$, for every $i \in \{0, \ldots, n-1\}$.

Output For each test case output one line with a single integer that denotes the required sum.

Simple enough? Let us look at an example of a valid solution.

Example 2.1.2 (Solution for 'Sum It!').

```cpp
#include <iostream> // We will use C++ input/output via streams

void testcase() {
    int n; std::cin >> n; // Read the number of integers to follow
    int result = 0; // Variable storing the result
    for (int i = 0; i < n; ++i) {
        int a; std::cin >> a; // Read the next number
        result += a; // Add it to the result
    }

    std::cout << result << std::endl; // Output the final result
}

int main() {
    std::ios_base::sync_with_stdio(false); // Always!

    int t; std::cin >> t; // Read the number of test cases
    for (int i = 0; i < t; ++i)
        testcase(); // Solve a particular test case
    return 0; // Signals successful completion to the operating system
}
```

IMPORTANT! The first line of the `main` function

```
std::ios_base::sync_with_stdio(false);
```

probably deserves an explanation. You should **always** include this command into your solutions, put it before the first I/O operation, and use C++-style stream I/O only—unless you really know what you do and insist to use C-style I/O instead.

This command disables the synchronisation between C-style `cstdio` operations (such as `scanf` and `printf`) and C++ standard streams (such as `std::cin`, `std::cout`, and `std::cerr`). Such a synchronisation is enabled by default, but it noticeably slows down C++ stream I/O. The amount of slowdown depends on the amount of input data to read, which is substantial for some problems. For other problems with smaller inputs, the slowdown may be less noticeable. Regardless, as your code is run in a timed environment, you really want to avoid any slowdown. Hence, just get used to include this command to be on the safe side.

2.2 The Judge

After you have tested your program locally, submit it to *the Judge* (`https://judge.inf.ethz.ch`). As seen in Exercise 2.1.1, you are given a problem description and the description of the input format. The input is given on `stdin`, which means you can just read it from the C++ standard input stream `std::cin`. Additionally, you get a description of the output format that has to be written on `stdout` via `std::cout`.

The Judge takes your source code, compiles it, and runs it with some input data—designed by us—to test your solution. The output of your program is then collected and compared to the expected (correct) output. The outcome of your trial can be seen (almost) immediately, as the Judge returns one of the following:

CORRECT Congratulations! Your code solved all instances correctly and within the given time and resource constraints.

WRONG-ANSWER Your program gave a wrong answer. This indicates a bug in your code. Or maybe you implemented a partial solution that was intended to solve a subset of the test sets only.

ASSERTION-FAILURE Your program terminated with a SIGABRT. This indicates an assertion failure, which among other things can be caused by wrong use of library routines or memory (malloc arena) corruption.

SEGMENTATION-FAULT Your program terminated with a SIGSEGV. This indicates it attempted to access memory that it is not allowed to, and is the result of a programming error, such as an out-of-bounds array access or dereferencing a `null` pointer.

RUN-ERROR Your program exited with nonzero exit status. Most probably your `main()` function returns a nonzero value. (This status also includes all signal exits not covered by the two preceding items.)

TIMELIMIT Your program has exceeded the allowed running time.

FORBIDDEN Your program tried to make a forbidden system call not on the whitelist or it tried to break out of the valgrind environment.

Partial solutions. Most problems come with several groups of test sets, typically three to five. These groups differ in complexity, for instance, in the size of the input. Some groups may also specify additional properties of the input or output that make the problem easier to solve. In this way, a problem can be solved partially and partial points can be obtained for such solutions. It is very important to carefully read the specifications of the different groups of test sets. Even if you do not see how to solve a problem in its entirety, obtaining an easier partial solution should always be possible. Often, but not always, a partial solution can hint in direction of a full solution.

In case your code yields the same result on all groups of test sets, exactly this result is shown. If your code yields different results for the various groups of test sets, the Judge shows a sequence of the corresponding result strings in an abbreviated format. For instance, a result of **OK OK WA TL** indicates that your code solved the first two groups of test sets correctly, gave a wrong answer on the third, and exceeded the time limit on the fourth.

Groups of test sets. Each group of test sets consists of several different test cases or instances. *All of these instances* have to be solved correctly and within the given time and resource constraints in order to obtain a **CORRECT** result from the Judge and accordingly score the listed amount of points.

2.3 Namespaces

Namespaces are a C++ mechanism to ensure that code that originates from different sources can safely be combined within a single program. Every library or project defines its own namespace, where all of its classes, objects, and functions are declared and defined. In this way, the same name can be used for different entities, as long as they reside in different namespaces, and name clashes are avoided. For instance, the C++ standard library defines all its classes, objects, and functions within the namespace `std`, the Boost libraries use a namespace `boost`, and the CGAL library uses a namespace `CGAL`.

There exist ways to effectively disable namespaces by so-called *using directives*. While it may seem convenient at first sight to not have to type the namespace, we advise you to embrace the benefits that namespaces provide rather than throwing them away.

A main benefit is *code readability*. By spelling out the namespace it is immediately explicit where a class, object, or function originates from. In particular when using several libraries in combination—as we frequently will within the Algorithms Lab—it helps to distinguish between parts from the standard library, from Boost, from CGAL, and locally defined names, which can be recognised by not having a namespace qualification.

A second benefit is the obvious one: *avoid name clashes*. C++ allows overloading functions and the rules for name lookup and argument matching are nontrivial. By calling a function with its namespace qualifier, you greatly reduce the possibility of name clashes, which may lead to compiler errors due to ambiguity or very nasty runtime errors, in case the function actually called is not the one you expected.

2.4 Fundamental Data Types

For a more detailed description of C++ data types we refer the reader to the online C++ documentation: `https://en.cppreference.com/w/cpp/language/types`. Nevertheless, we point out several of the most used data types and their sizes (see Table 2.1).

- `bool`, to represent one of two values: `true` or `false`;
- `char`, to represent a character, such as `'a'`, `'Z'`, or `'@'`;
- `std::string`, to represent a sequence of characters;
- `int` and `long` to represent integer numbers, such as `-23` and `7L` (the appended `L` denotes a `long` literal);
- `double`, IEEE-754 double-precision binary floating-point representation for rational/real numbers, such as `.5` or `-12E-3`; there are also some special symbolic values such as `Infinity`, *negative zero* `-0.0`, and *not-a-number* `NaN`.

Type specifier	Standard	Judge	Min Integer	Max Integer
`int`	≥ 32 bits	32 bits	-2^{31}	$2^{31}-1$
`long`	≥ 64 bits	64 bits	-2^{63}	$2^{63}-1$
`double`	64 bits	64 bits	$-(2^{54}-1)$	$2^{54}-1$

Table 2.1: Built-in C++ number types and some of their characteristics. For `double`*, the last two columns refer to the mantissa, which has* 53 *bits, so that* all *integers in the listed range can be represented exactly.*

Most of the mentioned types are *built-in* types, which are part of the C++ core language. Only `std::string` is not a built-in type. It is part of the C++ standard library—hence the `std::` qualification. To use `std::string`, you have to include the `string` library.

```
#include <string>
```

Initialisation. In many scenarios, variables of built-in type are *not* initialised by default for efficiency reasons. Thus, wherever a specific value is required, use an explicit initialisation.

```
int x; // The value of x is undefined for now
int y = 0; // Explicit initialisation to set the value of y to zero
```

Other built-in types. There exist other built-in number types, such as `short`, `long long`, `float`, or `long double`. But within the scope of this course there is no reason to use them.

Limited precision. All built-in number types have a limited, fixed precision. If the result of an operation (for instance, a multiplication) is outside of the representable range of the corresponding type, it cannot be represented and is truncated or rounded. For efficiency reasons, there is no warning, error, or exception during runtime to indicate such a loss of precision. It is up to the programmer to handle it, for instance, by making sure that the size of the operands is so that the result can always be represented exactly and correctly.

As an example, consider the code in Example 2.1.2, where the `result` is stored as an `int`. The $n \leq 10$ input numbers of absolute value at most 10^3 sum up to a result r with

$$|r| \leq 10 \cdot 10^3 = 10^4 < 2^{14} < 2^{31} - 1,$$

which comfortably fits within an `int`. If n or the size of the input numbers would be considerably larger, this might not be the case, and in such a setting the code from Example 2.1.2 may not work correctly.

IMPORTANT! When working with limited precision data types, make sure that they can represent all necessary values that may occur during the execution of your program.

2.5 Input and Output

We outline basic ways of reading the input from `stdin` and formatting the output on `stdout`.

- Read n integers and output their sum (see, Exercise 2.1.1)

```
int sum = 0;
int n; std::cin >> n;
for (int i = 0; i < n; ++i) {
    int a; std::cin >> a;
    sum += a;
}
std::cout << "The sum is: " << sum << std::endl;
```

- Read a string

```
std::string name;
std::cout << "Write your name: ";
std::cin >> name; // This reads a string until the first empty space
std::cout << "Hello " << name << "!" << std::endl;
```

- Write something with a specified fixed precision (requires the `iomanip` library):

```
double a, b, c;
a = 3.1415926534;
b = 2006.0;
c = 1.0e-10;
std::cout << std::setprecision(5); // Sets the precision of the out stream to exactly 5
std::cout << a << '\t' << b << '\t' << c << std::endl;
std::cout << std::fixed << a << '\t' << b << '\t' << c << std::endl;
std::cout << std::scientific << a << '\t' << b << '\t' << c << std::endl;
```

The snippet above creates the following output:

```
3.1416          2006            1e-10
3.14159         2006.00000      0.00000
3.14159e+00     2.00600e+03     1.00000e-10
```

IMPORTANT! For some problems with large inputs, reading this input is a noticeable part of the overall execution time. Whenever reading an integer, adhere to the following rule: If the number fits into an `int`, read it as an `int`; otherwise, read it as a `long`.

Note that `std::endl` does two things: It inserts a newline character `'\n'` into the output stream and then flushes the stream. Flushing the stream is comparatively expensive, so you do not want to do it too frequently. If only a newline is needed, just write `'\n'`. On the other hand, if your program crashes, then the characters on an unflushed stream are usually lost. So keep this in mind when debugging.

Limits. The `limits` library is useful to test for specifics of number types (given that they are platform dependent). It provides, among other things, the minimum and maximum values of fundamental data types; see Example 2.5.1 below.

Example 2.5.1.

```cpp
#include <limits>
#include <iostream>

int main() {
    std::cout << "type\tmin()\t\t\tmax()\t\t\t#binary digits\n";
    std::cout << "int\t"
              << std::numeric_limits<int>::min() << "\t\t"
              << std::numeric_limits<int>::max() << "\t\t"
              << std::numeric_limits<int>::digits << "\n";
    std::cout << "long\t"
              << std::numeric_limits<long>::min() << "\t"
              << std::numeric_limits<long>::max() << "\t"
              << std::numeric_limits<long>::digits << "\n";
    std::cout << "double\t"
              << std::numeric_limits<double>::min() << "\t\t"
              << std::numeric_limits<double>::max() << "\t\t"
              << std::numeric_limits<double>::digits << "\n";
    return 0;
}
```

Possible output (compare with the entries in Table 2.1):

```
type        min()                   max()                   #binary digits
int         -2147483648             2147483647              31
long        -9223372036854775808    9223372036854775807     63
double      2.22507e-308            1.79769e+308            53
```

Exercise 2.5.2 (Basic Data Types)**.** In this exercise you have to read several basic C++ data types and output them to the standard output in the required format.

Input The first line of the input contains the number $t \leq 10$ of test cases. Each of the t test cases is described as follows.

- It consists of four values, separated by a space. These values have different types, as given by the following order: `int`, `long`, `std::string`, `double`.

Output For each test case output one line containing all input data types, separated by a space. Floating point numbers should be rounded to 2 decimal digits.

2.6 Strings

As mentioned before, strings represent a sequence of characters. In contrast to Java strings, the C++ class `std::string` is mutable, that is, strings can be changed directly without having to create another new object. Strings are ordered lexicographically, and the `<` operator is defined accordingly.

Initialisation:

```
std::string s1(10, ' '); // A string of ten spaces
std::string s2("string");
std::string s3 = "string";
```

Read, write, and concatenate:

```
std::string name;
std::cin >> name; // Reads only one word, not the whole line!
std::string line = "Hello " + name + "!";
std::cout << line << std::endl;
std::cout << name[0] << std::endl; // Outputs the first character of the name variable
```

Sometimes it is useful to map characters into integers by subtracting −`'a'`, or in the case of capital letters −`'A'`.

```
char c = 'g';
int index = c - 'a';
std::cout << index << std::endl; // Outputs '6' to the standard output
```

For further reference we refer the reader to the online C++ documentation: https://en.cppreference.com/w/cpp/string/basic_string.

Exercise 2.6.1 (Strings)**.** In this exercise you practise some operations on strings.

Input The first line of the input contains the number $t \leq 10$ of test cases. Each of the t test cases is described as follows.

- It consists of a single line that contains two strings `a b`, separated by a space, and such that their individual length is at most 10.

Output For each test case output three lines.

- The first line contains two integers, separated by a space, representing the length of a and b, respectively.
- The second line contains a string obtained by concatenating the given strings a and b.
- The third line contains two strings `c` and `d`, where c and d are obtained by first reversing the strings a and b and then swapping the first characters of the newly obtained strings.

2.7 Pairs, Tuples, Custom Structures

Pairs. The class `std::pair` represents a pair of two objects of possibly different type as a single unit. In order to use it, one must include the `utility` library (which may also be indirectly included by some other library, such as `iostream`).

```
#include <utility>
...
std::pair<int, int> p1(0, 1); // A pair of integers (0, 1);
std::cout << p1.first << " " << p1.second << std::endl; // Outputs '0 1'
std::pair<int, std::string> p2(3, "three"); // A pair of an integer and a string
```

Similar to strings, pairs are ordered lexicographically, and the `<` operator is defined accordingly.

Tuples. As a generalization of `std::pair`, the class `std::tuple` represents a fixed-size collection of values of possibly different type. In order to use it, one must include the `tuple` library.

```
#include <tuple>
...
std::tuple<double, std::string, int, char> t(3.14, "Pi", 1, 'c');
std::cout << std::get<0>(t) << " " << std::get<1>(t) << " "
          << std::get<2>(t) << " " << std::get<3>(t) << std::endl;
```

The snippet above creates the following output:

```
3.14 Pi 1 c
```

Custom Structures. Usually it is more convenient to have a custom structure which serves as a collection of values instead of using an `std::tuple`. One of the reasons being that one may give names to the values and then access them accordingly, instead of using the `std::get()` function.

```
#include <iostream>

struct Edge {
  int start, end;
  double weight;
  int visited;

  // This constructor is convenient for a concise initialization. It can also
  //  be omitted and the member variables be set manually.
  Edge(int s, int e, double w, int v) :
    start(s), end(e), weight(w), visited(v)
  {}
};

int main() {
  Edge e(0, 1, 0.5, 0);
  std::cout << e.start << " " << e.end << " " << e.weight << " " << e.visited << std::endl;
}
```

The snippet above creates the following output:

```
0 1 0.5 0
```

2.8 Typedefs

When using data types from libraries, it is quite common that the corresponding typenames appear several times in the code. In such a case it is very convenient to define a local alias/synonym for such a type. The benefits are twofold: For once, a local alias is usually shorter, so there is less to type and the lines in the source code do not get so long. In addition, it is much easier to change the type if desired because only the definition of the local alias needs to be changed rather than every single occurrence of the typename. The C++ construct that allows to define such an alias for a typename is called `typedef`.

```
typedef std::tuple<double, std::string, int, char> DSIC;
// from here on we can use DSIC as a synonym for std::tuple<double, std::string, int, char>
DSIC t(3.14, "Pi", 1, 'c');
```

As with all names you define, for the sake of code readability it is advisable to choose them in a meaningful way and not make them too cryptic just for brevity.

Chapter 3

Data Structures

In this chapter we give a brief overview over the most important data structures from the C++ standard library for the Algorithms Lab.

3.1 Arrays, Vectors, Deque, Lists

Builtin arrays, `T[]`

We recommend to not use them and always use `std::array` or `std::vector` instead. For all intents and purposes of the Algorithms Lab, the difference in performance is negligible.

Arrays, `std::array`

The class `std::array` is a container to represent a sequence of arbitrary but fixed length. It has all the benefits of builtin arrays `T[]`, such as efficiency, and then some additional benefits that builtin arrays do not provide, such as consistent copy semantics. To use `std::array`, you need to include the `array` library.

```
#include <array>
```

Initialization:

```
typedef std::array<int, 5> Aint5; // array of five ints
Aint5 a; // The elements are not initialized => values are undefined.
Aint5 b = {}; // The elements are initialized to the default value, which for int is zero.
Aint5 c = {1, 2, 3, 4, 5}; // initialization as specified
```

Element access works as you would expect, numbered from zero and *without any bounds checking* (be careful with your indices!). There is also a highly useful `at` function, which checks bounds.

```
// element access using square brackets:
int d = c[1]; // initializes d to 2 (recall that indices always start from zero)
// element access using the at member function (checks for out-of-bounds):
d = c.at(3); // sets d to 4
```

One can iterate through an `array` using a 'for-each' loop[1].

```
#include <iostream>
#include <array>

typedef std::array<int, 5> Aint5;

std::ostream& operator<<(std::ostream& o, const Aint5& a) {
  for (int i : a) o << i << " ";
  return o;
}

int main() {
    Aint5 a = {1, 3, 5, 7, 9};
    std::cout << a << std::endl;
}
```

For further reference we refer the reader to the online C++ documentation: https://en.cppreference.com/w/cpp/container/array.

Vectors, `std::vector`

The class `std::vector` is a container to represent a sequence whose length may change dynamically. All memory management is done automatically. As for `std::array`, the elements are stored contiguously, allowing for constant time random access to elements. In order to use `std::vector`, one must include the `vector` library.

```
#include <vector>
```

Initialisation:

```
typedef std::vector<int> VI;
VI a; // a starts out empty
VI b(10); // b starts out with 10 elements each set to default 0
VI c(10, 42); // c starts out with 10 elements each set to 42

int n = 100, m = 50;
std::vector<VI> dp(n, VI(m, -1)); // A two-dimensional vector -> n x m matrix
```

Element access works the same as for `std::array`.

```
b[5] = 1; // Element number 5 set to 1.
a[0] = 1; // This will most likely segfault because a is empty.
b[10] = 1; // This will most likely NOT segfault and cause trouble later!
b.at(10) = 1; // This will terminate with SIGABRT.
std::cout << dp[17][33] << std::endl; // Outputs '-1' to the standard output.
```

Insertion and deletion at/from the end of the sequence is very efficient (amortized constant time). But inserting at or deleting from any other position is rather expensive. Effectively, all elements following in the sequence have to be moved around, which leads to a linear worst-case runtime.

[1]This syntax was introduced in C++11; some older compilers may not support it or require a specific language option to be set.

```
b.clear(); // b is now empty
a.push_back(5); // Appends 5 to a
c.pop_back(); // Removes the last element of vector c
```

Below is a table with the time complexity for the most important operations with vectors, where n denotes the current size (number of elements).

Operation	**Worst-Case Complexity**
Insert/delete an element to/from the front	$O(n)$
Insert/delete an element at/from a given index	$O(n)$
Insert an element to the back	$O(n)$ ($O(1)$ amortized)
Delete an element from the back	$O(1)$
Access an element at a given index	$O(1)$

Let us comment on the `push_back` function, which inserts an element at the back of the sequence. A single `push_back` call has *worst-case complexity* $O(n)$ and *amortized complexity* $O(1)$. An *amortized analysis* averages the time required to perform a sequence of operations over the number of operations performed. Here, a sequence of n `push_back` calls takes $O(n)$ time, leaving a single call with amortized complexity $O(1)$. Knowing the maximum size of a vector beforehand, one may call the `reserve` function to reserve space in advance. Then `push_back` becomes a worst-case constant time operation, as long as the reserved space suffices.

For further reference we refer the reader to the online C++ documentation: `https://en.cppreference.com/w/cpp/container/vector`.

Easy Rule of Thumb: Use `std::array` over `T[]` and `std::vector` over both!

Exercise 3.1.1 (Vectors)**.** In this exercise you are supposed to do some operations with vectors.

Input The first line of the input contains the number $t \leq 10$ of test cases. Each of the t test cases is described as follows.

- It starts with a line that contains an integer `n`, such that $0 \leq n \leq 10$.
- The following line contains n integers $\mathtt{a}_0 \ \ldots \ \mathtt{a}_{n-1}$, separated by a space, such that $-1000 \leq a_i \leq 1000$, for all $i \in \{0, \ldots, n-1\}$.
- The following line contains an integer `d`, denoting the index of an element that is to be removed from the vector, and such that $0 \leq d \leq n-1$.
- The following line contains two integers `a b`, separated by a space, denoting the range of indices of the elements that should be removed from the *remaining* vector (both inclusive), and such that $0 \leq a \leq b \leq n-2$.

Output For each test case output one line with the remaining elements of the vector separated by a space. If there are no elements remaining in the vector output '`Empty`'.

Deque, `std::deque`

Deque (pronounced as 'deck', stands for **D**ouble **E**nded **Que**ue) `std::deque` is a vector-like indexed sequence container allowing fast insertion and deletion to both the beginning and the end. The main difference to `std::vector` is that elements of a deque are not stored contiguously and allow quick insertion at the beginning. They still allow for constant time random access to elements. In order to use it, one must include the `deque` library.

```
#include <deque>
```

Initialisation:

```
std::deque<int> d = {3, 5, 7};
d.push_front(1); // Adds 1 to the front of deque d
d.push_back(9); // Adds 9 to the back of deque d
```

The table with time complexities of most important operations with deques is given below (n denotes the current size of the container).

Operation	Worst-Case Complexity
Insert/delete an element to the front	$O(1)$
Insert/delete an element at the given index	$O(n)$
Insert/delete an element to the back	$O(1)$
Access an element at the given index	$O(1)$
Find an element	$O(n)$

For further reference we refer the reader to the online C++ documentation: https://en.cppreference.com/w/cpp/container/deque.

Lists, `std::list`

List `std::list` is a container allowing constant time insertion and deletion of elements from any position (with a caveat, see below). On the down side, random access to elements is not supported. Internally, lists are usually implemented as *doubly-linked lists*. In order to use them, one must include the `list` library.

```
#include <list>
```

Initialisation:

```
std::list<int> l = {3, 5, 7};
l.push_back(9);
l.push_front(1);
```

In order to perform insertion at a specific position one needs to have an iterator pointing to the position. This is a significant drawback and might not behave as expected. Namely, in order to get an iterator pointing to the i-th element of the list given an iterator to the beginning has complexity $O(n)$.

```
// Insert an integer before 5 by searching
auto it = std::find(l.begin(), l.end(), 5);
if (it != l.end()) { // If element '5' exists in the list
  // The iterator points to the found element, i.e. to '5'
  l.insert(it, 4); // Inserts '4' before '5'
  // The iterator points to the element after the inserted one, i.e. to '5'
}
```

```
++it; // The iterator now points to '7'
l.insert(it, 6); // Inserts '6' before '7'

for (int i : l) {
  std::cout << i << " ";
}
```

The snippet above creates the following output:

```
1 3 4 5 6 7 9
```

The table with time complexities of most important operations with lists is given below (n denotes the current size of the container).

Operation	**Worst-Case Complexity**
Insert/delete an element to the front	$O(1)$
Insert/delete an element at the given index	$O(1)$
Insert/delete an element to the back	$O(1)$
Access an element at the given index	$O(1)$
Find an element	$O(n)$

The complexity is expressed with respect to a *given iterator*. However, obtaining an iterator from an index has time complexity $O(n)$ as mentioned before.

For further reference we refer the reader to the online C++ documentation: `https://en.cppreference.com/w/cpp/container/list`.

3.2 Sets and Maps

Set, `std::set`

Set `std::set` is a container that contains a sorted set of unique objects of keys. The sorting is performed using the `<` operator of the key. Hence, in order to have a set of custom data structures one needs to define the `<` operator (see, Section 4.1). Internally, sets are usually implemented as *red-black trees*. In order to use it, one must include the `set` library.

```
#include <set>
```

Initialisation:

```
std::set<int> s;
s.insert(25); // Inserts 25 into the set
s.clear(); // Erases the set's contents
```

Erasing an element:

```
std::set<int> s = {1, 3, 5, 7, 9};
auto it = s.find(3); // First one needs an iterator pointing to the element
s.erase(it); // The iterator points to the element '3'!

s = {1, 3, 5, 7, 9};
it = s.find(3);
it = s.erase(it); // The iterator points to the element after, that is '5'!
```

Note that finding an iterator to an element of a set has worst-case complexity $O(\log n)$. The delete operation has worst-case complexity $O(\log n)$ and amortised $O(1)$ (see, Section 3.1).

As the elements of the set are sorted, they prove to be highly useful when one needs to find a minimum/maximum of elements quickly.

```
int min_e = *s.begin(); // Returns the minimum element of the set s
int max_e = *s.rbegin(); // Returns the maximum element of the set s
```

Additionally, one may easily perform a *binary search* on the elements of a set by using its `std::set::lower_bound` and `std::set::upper_bound` functions. They return an iterator to the first element that is *not smaller* than key and that is *larger* than key, respectively. Both have time complexity of $O(\log n)$ where n denotes the size of the set (cf. Section 4.1).

```
std::set<int> s = {1, 3, 5, 7, 9};
std::cout << *s.lower_bound(3) << " " << *s.upper_bound(5) << std::endl;
```

The snippet above creates the following output:

```
3 7
```

The table with time complexities of most important operations with sets is given below (n denotes the current size of the container).

Operation	**Worst-Case Complexity**
Insert an element	$O(\log n)$
Delete an element	$O(\log n)$
Find an element	$O(\log n)$

One can also use `std::multiset` which allows storing keys with the same value.

For further reference we refer the reader to the online C++ documentation: `https://en.cppreference.com/w/cpp/container/set`.

Exercise 3.2.1 (Sets)**.** In this exercise you are supposed to do some operations with sets.

Input The first line of the input contains the number $t \leq 10$ of test cases. Each of the t test cases is described as follows.

- It starts with a line that contains an integer `q`, such that $0 \leq q \leq 10$.
- The following q lines each contain two integers `a b`, separated by a space, such that $a \in \{0, 1\}$ and $-1000 \leq b \leq 1000$, in one of the following forms:
  - `0 b`: add the element b to the set;
  - `1 b`: delete the element b from the set.

Output For each test set output one line with the remaining elements of the set separated by a space. If there are no elements remaining in the set output '`Empty`'.

Unordered set, `std::unordered_set`

Unordered set `std::unordered_set` is a container that contains a set unique objects of keys. However, unlike `std::set`, the elements are *not sorted*. The elements are organised into buckets depending on the hash of its key. In order to use it, one must include the `unordered_set` library.

The table with time complexities of most important operations with unordered sets is given below (n denotes the current size of the container).

Operation	**Worst-Case Complexity**
Insert an element	$O(n)$ ($O(1)$ on average)
Delete an element	$O(n)$ ($O(1)$ on average)
Find an element	$O(n)$ ($O(1)$ on average)

Note that all three operation have complexity $O(1)$ on average.

IMPORTANT! The performance of an `std::unordered_set` highly depends on the hash of its key. Namely, having a 'bad' hash function in case you want to store custom structures would put several (maybe all) keys into the same bucket and thus increase the complexity of insert/access operations to $O(n)$.

For further reference we refer the reader to the online C++ documentation: https://en.cppreference.com/w/cpp/container/unordered_set.

Map, `std::map`

Map `std::map` is a sorted container that contains key-value pairs with unique keys. The sorting is performed using the `<` operator of the key. Hence, in order to have a set of custom data structures one needs to define the `<` operator (see, Section 4.1). Internally, maps are usually implemented as *red-black trees*. In order to use it, one must include the `map` library.

```
#include <map>
```

Initialisation:

```
std::map<std::string, int> m;
m.insert(std::make_pair("One", 1));
m.insert(std::make_pair("Two", 2));

std::cout << m.at("One") << std::endl; // Outputs '1' to the standard output
std::cout << m["Ten"] << std::endl; // Outputs '0' to the standard output
std::cout << m.at("Zero") << std::endl; // Causes an error
```

`std::map::operator[]` gives a reference to the value that is mapped to the key or performs an insertion if the key does not exist (with the default value of the data type).

The table with time complexities of most important operations with maps is given below (n denotes the current size of the container).

One can also use `std::multimap` which allows storing keys with the same value.

For further reference we refer the reader to the online C++ documentation: https://en.cppreference.com/w/cpp/container/map.

Operation	**Worst-Case Complexity**
Insert an element	$O(\log n)$
Access an element by key	$O(\log n)$
Delete an element by key	$O(\log n)$
Find/remove a value	$O(\log n)$

Exercise 3.2.2 (Maps)**.** In this exercise you are supposed to do some operations with maps.

Input The first line of the input contains the number $t \leq 10$ of test cases. Each of the t test cases is described as follows.

- It starts with a line that contains an integer `q`, such that $0 \leq q \leq 10$.
- The following q lines each contain an integer and a string `a b`, separated by a space, such that $0 \leq a \leq 1000$ and b is of length at most 10, in one of the following forms:
  - `0 b`: erase all entries with b as the key;
  - `x b`, for $x > 0$: add an entry with key b and value x to the map.
- The following line contains a string s consisting of length at most 10.

Output For each test case output all the values with the key s in a non-decreasing order, separated by a space. If there are no elements with the key s output `Empty`.

Note It may be tempting to use `std::map` or `std::unordered_map` as a dynamic programming table. We recommend you to avoid this especially if the DP state consists of integers only. Using a d-dimensional `std::vector` instead is a much faster option, that is $O(1)$ access vs. $O(\log n)$ access (or depending on the hash function, see below). However, `std::map` becomes useful once the DP state consists of several different data types, not only integers.

Unordered map, `std::unordered_map`

Unordered map `std::unordered_map` is a container that contains key-value pairs with unique keys. However, unlike `std::map`, the elements are *not sorted*. The elements are organised into buckets depending on the hash of its key. Internally, unordered maps are usually implemented as *hash tables*. In order to use it, one must include the `unordered_map` library.

The table with time complexities of most important operations with unordered maps is given below (n denotes the current size of the container).

Operation	**Worst-Case Complexity**
Insert an element	$O(n)$ ($O(1)$ on average)
Access an element by key	$O(n)$ ($O(1)$ on average)
Delete an element by key	$O(n)$ ($O(1)$ on average)
Find/remove a value	$O(n)$ ($O(1)$ on average)

Note that all four operation have complexity $O(1)$ on average.

IMPORTANT! The performance of an `std::unordered_map` highly depends on the hash of its key. Namely, having a 'bad' hash function in case you want to store custom structures would put several (maybe all) keys into the same bucket and thus increase the complexity of insert/access operations to $O(n)$.

For further reference we refer the reader to the online C++ documentation: https://en.cppreference.com/w/cpp/container/unordered_map.

3.3 Stack, Queue, Priority Queue

Stack, `std::stack`

Stack `std::stack` is a FILO (first-in-last-out) data structure. Internally, it is usually implemented as a `std::deque`. In order to use it, one must include the `stack` library.

```
std::stack<int> s;
s.push(17); // Adds an element to the top of the stack
int top = s.top(); // Returns the top element of the stack
s.pop(); // Removes the top element from the stack
```

It proves useful for graph algorithms, in particular an iterative implementation of the Depth-First Search.

The table with time complexities of most important operations with a stack is given below.

Operation	Worst-Case Complexity
Push (insert) an element	$O(1)$
Pop (delete) an element	$O(1)$
Access the top element	$O(1)$

For further reference we refer the reader to the online C++ documentation: https://en.cppreference.com/w/cpp/container/stack.

Exercise 3.3.1 (DFS)**.** Compute the DFS timestamps of discovery and finishing of all vertices starting from a given vertex. The order in which the DFS traversal visits the vertices should be such that it *always* visits the unvisited neighbor of the current vertex *with the smallest identifier.*

Input The first line of the input contains the number $t \leq 10$ of test cases. Each of the t test cases is described as follows.

- It starts with a line that contains three integers `n m v`, separated by a space, denoting the number of vertices, the number of edges, and the starting vertex, and such that $0 \leq n \leq 10^3$, $0 \leq m \leq \binom{n}{2}$, and $0 \leq v \leq n-1$.
- The following m lines each contain two integers `a b`, separated by a space, indicating that $\{a, b\}$ is an edge of the graph.

Output For each test case you should output two lines: the first containing the timestamps of discovery separated by a space and ordered by increasing labels; the second containing timestamps of finishing separated by a space and ordered by increasing labels. If a vertex cannot be reached, both of its timestamps are -1.

Queue, `std::queue`

Queue `std::queue` is a FIFO (first-in-first-out) data structure. Internally, it is usually implemented as a `std::deque`. In order to use it, one must include the `queue` library.

```
std::queue<int> q;
q.push(17); // Adds an element to the end of the queue
int front = q.front(); // Returns the first element added to the queue
int back = q.back(); // Returns the last element added to the queue
q.pop(); // Removes the first element from the queue
```

It proves useful for graph algorithms, in particular the Breadth-First Search.

The table with time complexities of most important operations with a queue is given below.

Operation	**Worst-Case Complexity**
Push (insert) an element	$O(1)$
Pop (delete) an element	$O(1)$
Access the top element	$O(1)$

For further reference we refer the reader to the online C++ documentation: `https://en.cppreference.com/w/cpp/container/queue`.

Exercise 3.3.2 (BFS)**.** Compute the distances of all vertices from a given starting vertex using BFS.

Input The first line of the input contains the number $t \leq 10$ of test cases. Each of the t test cases is described as follows.

- It starts with a line that contains three integers `n m v`, separated by a space, denoting the number of vertices, the number of edges, and the starting vertex, and such that $0 \leq n \leq 10^3$, $0 \leq m \leq \binom{n}{2}$, and $0 \leq v \leq n-1$.
- The following m lines each contain two integers `a b`, separated by a space, indicating that $\{a, b\}$ is an edge of the graph.

Output For each test case you should output one line containing the distance of the vertices from v, ordered by increasing labels. If a vertex cannot be reached, its distance is -1.

Priority queue, `std::priority_queue`

Priority queue `std::priority_queue` is a data structure where the elements are sorted by *priority* and not by their arrival time. By default, the elements are sorted such that the largest element appears at the `std::priority_queue::top`. Internally, it is usually implemented as a `std::vector`. In order to use it, one must include the `queue` library.

```
std::priority_queue<int> q;
q.push(17);
int top = q.top();
q.pop();
```

It is often used as a min-heap or a max-heap.

```
#include <iostream>
#include <queue>

typedef std::priority_queue<int> max_heap; // Default behaviour
typedef std::priority_queue<int, std::vector<int>, std::greater<int> > min_heap;

int main() {
    max_heap max_q;
    min_heap min_q;

    for (int i = 0; i < 5; ++i) {
        max_q.push(i);
        min_q.push(i);
    }

    std::cout << max_q.top() << " " << min_q.top() << std::endl;

    return 0;
}
```

The snippet above creates the following output:

```
4 0
```

The table with time complexities of most important operations with a priority queue is given below (n denotes the current size of the container).

Operation	**Worst-Case Complexity**
Push (insert) an element	$O(n)$ ($O(1)$ amortised)
Acces the top element	$O(1)$
Pop (delete) an element	$O(\log n)$

Since `std::priority_queue` is internally implemented as an `std::vector`, the amortised complexity of the `push` is $O(\log n)$, however the worst-case complexity is $O(n)$.

Sometimes it is useful to have a priority queue which is capable of handling custom structures. In order to do this, one needs to define the `<` operator of the structure or a functor defining the `()` operator and pass it to the priority queue (see, Section 4.1).

For further reference we refer the reader to the online C++ documentation: https://en.cppreference.com/w/cpp/container/priority_queue.

Chapter 4

Pearls of Wisdom

4.1 Algorithms

The C++ standard library provides many reusable algorithms. To use them, you need to include the `algorithm` library.

```
#include <algorithm>
```

Sorting. One of the most useful algorithms is probably `std::sort`, which sorts any linearly ordered random access container with n elements in worst-case $O(n \log n)$ time.

```
std::vector<std::string> v;
// here we fill v with strings... and then:
std::sort(v.begin(), v.end());
```

This works with any data type that is ordered, that is, for which the `<` operator is defined. In oder to sort in descending order, one can pass the `std::greater` functor as a third (optional) argument to `std::sort`.

```
std::vector<std::string> v;
// Fill vector with strings...
std::sort(v.begin(), v.end(), std::greater<std::string>());
```

Finding values in sorted sequences. Another pair of functions worth mentioning are `std::lower_bound` and `std::upper_bound`. The function `std::lower_bound` computes the first element in a given range $[f, \ell)$ that is *not smaller* than a given value. If no such element exists, ℓ is returned instead. Similarly, `std::upper_bound` computes the first element *strictly greater* than a given value. Both functions assume that the given range is sorted in a nondecreasing order. In case that the underlying container provides random access (e.g. `std::vector`), the time complexity is $O(\log n)$, where n is the length of the range. For non-random access containers (e.g. `std::set`), the time complexity is $O(n \log n)$.

```
std::vector<int> v = {1, 3, 5, 5, 7, 9, 9, 9};

auto lower = std::lower_bound(v.begin(), v.end(), 5);
auto upper = std::upper_bound(v.begin(), v.end(), 5);
```

```
std::cout << *lower << " " << *upper << std::endl; // Outputs '5 7' to the standard output
```

IMPORTANT! When searching among the elements of an `std::set` or an `std::map` always use their respective `lower_bound` and `upper_bound` members functions instead of the here mentioned global functions `std::lower_bound` and `std::upper_bound`. This ensures that the time complexity of the operation is $O(\log n)$ rather than $O(n \log n)$.

Random shuffle. Sometimes the input data is given in an adversarial way, that is such that the algorithm which is supposed to be performed on it works in the 'worst case' scenario rather than in the 'average case'. In order to get around such a scenario it can help to randomly permute the input using the `std::random_shuffle` function. The time complexity of the function is linear in the size of the container.

```
std::vector<int> v = {1, 3, 5, 5, 7, 9, 9, 9};

std::random_shuffle(v.begin(), v.end());

for (int i : v) std::cout << i << " ";
std::cout << std::endl;
```

One possible output of the snippet above is:

```
7 1 9 3 5 9 9 5
```

Lastly, the `algorithm` library also contains several useful functions for working with sets such as `std::set_intersection`, `std::set_difference`, and `std::set_union`.

For further reference we refer the reader to the online C++ documentation: https://en.cppreference.com/w/cpp/algorithm.

4.2 Predicates for Custom Data Types

In case you want to sort (or compare) custom data structures, you have at least three options.

The less-than operator. As C++ allows operator overloading, you can define the operator < for custom data types. This also work in connection with `std::set` or `std::map` for custom data types.

```
#include <iostream>
#include <vector>
#include <algorithm>

struct Edge {
  int start, end;
  double weight;
  int visited;

  Edge(int s, int e, double w, int v) : start(s), end(e), weight(w), visited(v) {}
};
```

```
// linearly order edges by increasing weight
bool operator<(const Edge& e1, const Edge& e2) {
  return (e1.weight < e2.weight);
}

int main() {
    Edge e1(0, 1, 0.5, 0);
    Edge e2(1, 2, 0.75, 0);
    Edge e3(2, 3, 0.25, 0);

    std::vector<Edge> edges;
    edges.push_back(e1); edges.push_back(e2); edges.push_back(e3);
    std::sort(edges.begin(), edges.end());

    for (Edge e : edges)
      std::cout << e.weight << "\n";

    return 0;
}
```

The snippet above creates the following output:

```
0.25
0.5
0.75
```

A custom functor. Alternatively, you can define a custom functor, and give it to the algorithm or data structure as an argument. This is particularly useful if the data type you want to sort has the `<` operator already defined differently, or in case that you want to sort separately according to different criteria.

For example, `std::pair` objects are by default sorted by lexicographic order. But maybe you want them to be sorted by the sum of their entries instead.

```
#include <iostream>
#include <vector>
#include <algorithm>

typedef std::pair<int, int> PII;

// functor to (less-than) compare the sum of entries
struct SumLessThan {
  bool operator()(const PII& p1, const PII& p2) const {
    return p1.first + p1.second < p2.first + p2.second;
  }
};

int main() {
    PII p(3, 17), q(5, 2);
    std::vector<PII> pairs;
    pairs.push_back(p); pairs.push_back(q);
    std::sort(pairs.begin(), pairs.end(), SumLessThan());

    for (PII i : pairs)
        std::cout << "(" << i.first << ", " << i.second << ") ";
    std::cout << std::endl;
    return 0;
}
```

The snippet above creates the following output:

```
(5, 2) (3, 17)
```

Use a lambda expression. Since C++11 it is possible to define so-called lambda expressions, which are unnamed temporary functors. They provide a way to wrap some inline code into an object that then can be passed as an argument to a function, such as `std::sort`. This is particularly useful if the functor in question appears only once in your code. As another benefit, the definition of the functor appears locally right where it is used, which can improve code readability. Ideally, the code fragment in question is short.

For comparison, here is the example from above again. But this time, the comparison operator is handed to `std::sort` as a lambda.

```
#include <iostream>
#include <vector>
#include <algorithm>

typedef std::pair<int, int> PII;

int main() {
  PII p(3, 17), q(5, 2);
  std::vector<PII> pairs;
  pairs.push_back(p); pairs.push_back(q);
  std::sort(pairs.begin(), pairs.end(),
            [](const PII& p1, const PII& p2) -> bool {
              return p1.first + p1.second < p2.first + p2.second;
            });

    for (PII i : pairs)
        std::cout << "(" << i.first << ", " << i.second << ") ";
    std::cout << std::endl;
    return 0;
}
```

Exercise 4.2.1 (Sort)**.** In this exercise you are supposed to do some sorting of integers.

Input The first line of the input contains the number $t \leq 10$ of test cases. Each of the t test cases is described as follows.

- It starts with a line that contains an integer `n`, such that $0 \leq n \leq 10^5$.
- The following line contains n integers $\mathtt{a}_0 \ \dots \ \mathtt{a}_{n-1}$, separated by a space, such that $-1000 \leq a_i \leq 1000$, for all $i \in \{0, \dots, n-1\}$.
- The following line contains an integer `x`, denoting whether the numbers should be sorted in a non-decreasing order or in a non-increasing order, and such that $x \in \{0, 1\}$.

Output For each test case output one line with the numbers sorted in a non-decreasing order if $x = 0$, and in a non-increasing order otherwise.

4.3 Iterators

Iterators are used to point to memory addresses of STL containers. They are mostly used to iterate through (hence the name) the elements of a container or to find a particular value. Vast number of algorithms, functions, and data structures rely on iterators in this or that way. Most common ones are `begin` and `end`, pointing to the first element and the element *after the last element* in a container.

- Sorting

```
std::vector<int> v = {1, 3, 5, 7, 9};
std::sort(v.begin(), v.end());
```

- Iterating through elements of a set

```
std::set<int> s = {1, 3, 5, 7, 9};
for (std::set<int>::iterator it = s.begin(); it != s.end(); it++) {
    std::cout << *it << " ";
}
std::cout << std::endl;
```

- Searching for an element in a container

```
std::vector<int> v = {1, 3, 5, 7, 9};
if (std::find(v.begin(), v.end(), 42) != v.end()) {
    // Do something
}
```

- Filling a vector with values

```
std::vector<int> v = {1, 3, 5, 7, 9};
std::fill(v.begin(), v.end(), 42);
```

- Summing up the values in a vector (requires the `numeric` library)

```
std::vector<int> v = {1, 3, 5, 7, 9};
std::cout << std::accumulate(v.begin(), v.end(), 0) << std::endl;
```

4.4 Pointers and References

IMPORTANT! Vectors and other large data structures that are constant across the invocation of a function should be passed as a `const` reference.

```
// This function does not modify the out-edge lists
void depth_first_search(int start, const std::vector<std::vector<int> > &outedges) {
    // Do something
}
```

This avoids the *huge* overhead associated with copying, especially when the data structures are nested.

Chapter 5

Compiling, Running, and Debugging

All tools necessary to solve the exercises for the Algorithms Lab are installed on the student lab computers. We suggest using `gedit` or `vim`/`emacs` to edit your source code.

If you want to use your own machines you have to install the GNU compiler collection. (Linux and Mac users should already have it on their machines, Windows users could install Cygwin or comparable tools. Also working remote on the student-lab machines is a possibility.)

Note For the second and third part of the Algorithms Lab you will have to use the Boost Graph Library and CGAL. In order to use your own computer you will have to install also these libraries later on.

IMPORTANT! Also if you use your own computer you should get used to the setup on the student lab machines. You will have to write the exam on them!

5.1 How to Compile

We are using `g++` to compile your C++ code. In order to compile your source, type the following command in the directory containing your source code:

```
g++ -Wall -O3 <SOURCE> -o <BINARY>
```

`g++` This is the traditional nickname of `GNU C++`, a freely redistributable C++ compiler. It is part of `GCC`, the GNU compiler collection.

`-Wall` Turns on all optional warnings which are desirable for normal code.

`-O3` Turns on many compiler optimisations. Your program will run faster! (Note: 'O' is not a zero but the capital character 'O' like in 'Optimization'!)

`-o <BINARY>` Place output in file `<BINARY>`.
(Something like '`funnyexercise`' or '`funnyexercise.exe`' on Windows systems.)

`<SOURCE>` Indicates the file containing the source.
(Something like '`funnyexercise.cpp`'.)

5.2 How to Run

Once you have compiled your code you should be able to run it. Since your program—as explained in Section 2.1—reads the inputs from `stdin` and writes all outputs on `stdout` the program will wait for inputs after starting it.

You can use your keyboard to type some test input, but usually it is less time consuming to prepare the input in a text file and pipe it directly into your program when starting it. In the command-line (terminal/console) it looks like this:

```
./funnyexercise < inputfile.in
```

The content of `inputfile.in` will be sent to `stdin` after starting the binary `funnyexercise`. The output will be printed on your screen though. In order to print the output to another file use the following command:

```
./funnyexercise < inputfile.in > funnyoutput.out
```

It is very useful to compare the output of your program to the expected output of the respective test set. This can be done by using the `diff` command.

```
./funnyexercise < inputfile.in | diff outputfile.out - --color
```

If the output of your program matches the output written in `outputfile.out` running the above snipped gives no feedback. Otherwise, it gives you colourful information about which lines differ and how exactly.

5.3 How to Debug

With the `cassert` library you can state implicit assumptions in your code as *assertions*. (You are not supposed to use this for *input checking*, but since judge inputs are supposed to be error-free, you can usually disregard the latter.)

You simply state a condition that should always hold in the `assert` macro, which is used like a function. For example, to read the number of sub-cases in a test, you might run:

```
int k;
cin >> k;
assert(k > 0);
```

If your program ever gets confused about its position in the input, and reads a negative number, you will notice immediately instead of later in the algorithm:

```
$ ./01_assert < 01_assert_sample.in
01_assert: 01_assert.cpp:16: int main(): Assertion ‘k > 0’ failed.
Aborted
```

For as far as Algorithms Lab is concerned, no advanced debugging and memory management tools are necessary. Simply writing printouts at specific places of your code should be more than enough for all the debugging purposes. If, however, you do want to make use of some proper tools, we recommend taking look at `gdb` (`https://www.gnu.org/software/gdb/`) for debugging and `valgrind` (`http://valgrind.org/`) for memory management and runtime control.

Chapter 6

How to Estimate the Runtime of Your Algorithm

6.1 The Clue

Of course we are usually searching for the fastest solution. But how fast should that be?

Every exercise can be solved in many different ways and each of them usually has a different runtime. So how should you know what we are looking for?

If you click on the 'Submit solution' link for the specific exercise you will find the time limit for this exercise. This is the maximal CPU-time your program is allowed to run on the Judge in order to come up with the correct solution for a single test set.

6.2 How to Use the Clue

There is a very simple rule of thumb: roughly 10^7 operations can be performed within 1 second. This is of course not always true and grossly oversimplified. It is surprisingly often a relatively good estimate though.

If you know that the solution you would like to implement has a complexity of $O(n^2)$, the exercise sheet tells you that $n \leq 50000$ and the time limit for the solution is set to 3 seconds you might reconsider your choice and implement the more complicated $O(n \log n)$ algorithm. This also 'works' the other way around: knowing how large the input can be you may be able to 'guess' what type of an algorithm you are expected to come up with in order to solve it.

6.3 Tips and Tricks

You believe you have arrived at the 'optimal' runtime but your solution still exists with a **TIMELIMIT**. Here are several things you might want to try still:

- Maybe input/output is the bottleneck of your algorithm. Do not forget to write the highly useful `std::ios_base::sync_with_stdio(false)` at the beginning of your program.
- The input might be given in an adversarial way. Try randomly permuting the input data

by using the `std::shuffle` function.

- Make sure you are passing the non-primitive variables to functions by reference and not copying the values every time you call a function. This may cause a major slowdown.
- Maybe a binary/exponential search can help instead of a linear search?
- Possibly cache the solution accross test cases and reuse them without having to compute the solution to the same test case again.