Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A Short Introduction to C++ for the Algorithms Lab

Michael Hoffmann and Miloš Trujić

Department of Computer Science, ETH Zürich

September, 2020

Acknowledgements

Several people have contributed immensely towards improving the presentation of this booklet with their ideas, feedback, corrections, and proofreading: Hlynur Jonsson, Johannes Kapfhammer, Anders Martinsson, Nicolas Trüssel, 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. 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, which are marked in orange. 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 submission process to get 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/
- B. Gärtner, M. Hoffmann, *An Introduction to C++*, Lecture Notes ETHZ, 2014.
- 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 $-10^3 \leq a_i \leq 10^3$, 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
}
```

IMPORTANT! The first line of the `main` function probably deserves an explanation.

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

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 are doing and insist on using the 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 Code Expert: Submission & Feedback

After you have tested your program locally, submit it to Code Expert (https://expert.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`.

Code Expert provides a light IDE that lets you edit, compile, run, test, and submit solutions. As a first step, you have to enroll into the corresponding course on Code Expert. The enrolment link can be found on the slides from the first tutorial and also on moodle. Then you can find the problems in the tab "Enrolled Courses". Clicking on one of the problem links brings you to the IDE, which should look roughly as shown in Figure 2.1.

In the middle of the screen you have an editor window where you can type or copy-paste your code, and directly below it a console where the results of compiling or running your code will be shown. To compile, simply hit the blue "compile" button (cogwheels icon). To the left of the editor window there is a file browser. Your source code resides in the `src` directory in the file `main.cpp`. In the directory `public` you can find the public test sets. To download these to work locally on your computer, hit the "download" button in the top-right corner of the file browser window.

To run your code on Code Expert with a specific test set of your choice, put the input into the file `user_test.in` and the expected output into the file `user_test.out`. By default these files contain the sample input and output, respectively. Clicking the blue "play" button (arrowhead icon), your code is compiled and run with the input from `user_test.in` and the output compared to the contents of `user_test.out`. The result is shown in the console.

To let Code Expert judge your solution by running it on the secret test sets designed by us, click the blue "test" button (flask icon). The evaluation is shown in the console. Here you can also see the point distribution (which is also described on the problem sheet) and the timelimits for each collection of test sets (which are not listed on the problem sheet).

The outcome of your trial can be seen (almost) immediately, as the system returns one of the

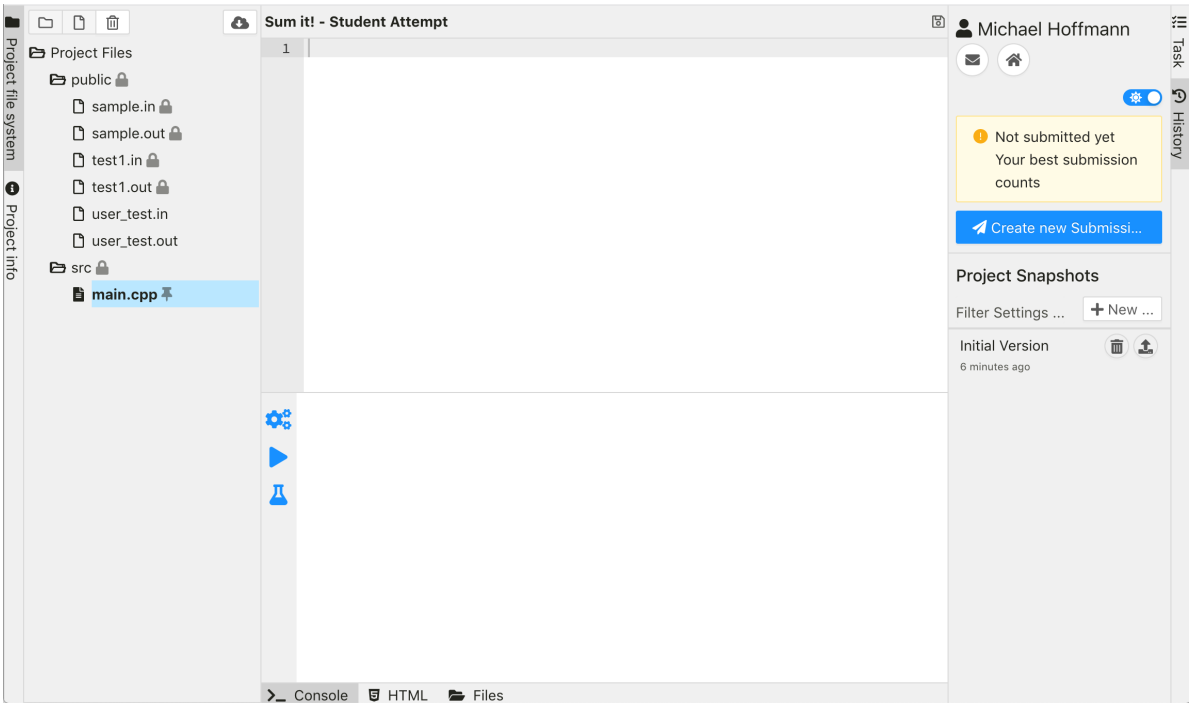

Figure 2.1: The Code Expert interface.

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.

TIMELIMIT Your program has exceeded the allowed running time.

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.

FORBIDDEN Your program tried to make a forbidden system call, or it tried to break out of the valgrind environment.

Submissions. To "make it count" you have to *submit* your code by clicking on the blue "Create new submission" button to the right of the editor window. Only submissions ...

- are stored and count for your score and—in the exam—for your grade,
- can be seen and reviewed by the assistants,

- will be run on *hidden* test sets during the *Problem of the Week* and during the exam. (For *hidden* test sets you do not see the results until after the deadline or at the exam review.)

IMPORTANT! Routinely make a new submission whenever you think your current solution may improve your score.

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 towards a full solution.

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 and accordingly score the listed amount of points.

Deadlines and points. Every problem set on Code Expert has a deadline. For regular problem sets this is two weeks after publication. For a Problem of the Week the deadline is two hours after publication. You can still submit problems and receive feedback for your solutions after the deadline. Late submissions are labeled as such and do not count for points. Points obtained during the semester mainly serve as a feedback to you; they have no impact on your final grade, which is based on the exam only—and a possible bonus obtained from the assessments.

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

Here we point out a few 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	Code Expert	Min Integer	Max Integer
`int`	≥ 16 bits	32 bits	-2^{31}	$2^{31}-1$
`long`	≥ 32 bits	64 bits	-2^{63}	$2^{63}-1$
`double`	64 bits	64 bits	-2^{53}	2^{53}

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* fundamental 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
```

You can run into very nasty bugs that seem to randomly appear during runtime if your program does not initialise a variable and then uses its (undefined) value. The reason is that sometimes the undefined value may be a value that works (say, zero), but some other time it may not.

IMPORTANT! Always make sure to properly initialise a variable before using its value.

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; // attention: finite precision!
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.

```
#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";
}
```

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
```

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`). There is also a generic function `std::make_pair` to create a corresponding pair from two given arguments.

```
#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
std::pair<char, long> p3 = std::make_pair('x', 4L);
```

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

Tuples. As a generalisation of `std::pair`, the class `std::tuple` represents a fixed-size collection of values of possibly different types. 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 create 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 initialisation. 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.

Alternatively, since C++11 one can define a `type alias` with the `using` keyword. The above example would then as follows.

```
using DSIC = std::tuple<double, std::string, int, char>;
// 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');
```

Typedef and type alias are equivalent for most purposes. Type aliases are slightly more general as they can be templated (if you do not know what this means, do not worry), and also the syntax is more intuitive. Also note that even though they use the same keyword, a type alias is different from a `using declaration` and from a `using directive` (which we recommend you to avoid).

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 purposes of the Algorithms Lab. This overview is not intended to serve as a reference, but more like a teaser. For more information and details, please follow the listed links to the online C++ reference documentation. Before we dive into the different data structures, let us very briefly explain some basic terminology that appears all across the C++ standard library.

3.1 Containers, Iterators, and Ranges

A *container* is a data structure to store and provide access to a collection of objects. The C++ standard library provides a number of containers, such as `std::vector`, `std::set`, etc. In order to abstract from implementation details, these containers are specified in form of *concepts*. Each concept defines the operations that the container supports along with efficiency guarantees for those operations. But it is left open how the container is implemented so as to achieve those guarantees.

Access to the elements of a container is provided via a range of iterators. Iterators can be regarded as an abstraction of pointers. More specifically, an *iterator* is an object that points to an element of a collection, and that can be used to iterate over the elements of that collection. To this end, an iterator can be *dereferenced* to obtain the element pointed to, and it can be *advanced*—using the `++` increment operator—to point to the next element of the collection. Some iterators also support a richer set of operations, such as the decrement operator `--` for bidirectional traversal, or even random access using addition or the index operator `[]`. Similar to a null-pointer, an iterator may also *not* refer to any element, in which case dereferencing it would be an error.

An iterator provides access to a single element of a collection and the means to advance it to then point to another element of the collection. But at some point it will have iterated over the whole collection. What then, and how can we tell? This is why a whole collection is not referred to by a single iterator but by a pair of iterators `[f, l)`, called *range* instead. The notation similar to halfopen intervals is intentional: `f` refers to the first element, which belongs to the collection, whereas `l` is *beyond* the last element and, therefore, does *not* refer to any element of the collection. Therefore, the following loop implements a full iteration over the underlying collection.

```
for (; f != l; ++f) {
    // do something with *f
}
```

Every container in the C++ standard library provides access to its elements via the range `[begin(), end())`. Additionally, most operations and algorithms in the C++ standard library define their interface in terms of iterators and iterator ranges.

3.2 Arrays, Vectors, and Deques

Built-in arrays, `T[]`

We recommend to not use them and always use `std::array` or `std::vector` instead.

Arrays, `std::array`

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

```
#include <array>
```

Initialisation:

```
std::array<int, 5> a; // The elements are not initialised and thus values are undefined.
std::array<int, 5> b = {}; // The elements are initialised to the default value,
                           // which for int is zero.
std::array<int, 5> c = {1, 2, 3, 4, 5}; // Initialisation 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]; // Initialises d to 2 (recall, 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
```

Iterating over Containers

One can iterate over an `std::array` (and every other standard container) using the `[begin(), end())` range.

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

// Convenient shortcut and single point of change (stands for array of integers)
typedef std::array<int, 5> AI;

int main() {
```

```
    AI a = {1, 3, 5, 7, 9};
    for (AI::const_iterator it = a.begin(); it != a.end(); ++it)
        std::cout << *it << " ";
    std::cout << std::endl;
}
```

In this case we do not modify the elements within the loop and, therefore, a const iterator suffices. If we wanted to change the elements—say, add one to each—we would use `AI::iterator` instead.

Alternatively, we could avoid specifying the iterators explicitly and use a range-based for loop.

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

int main() {
    std::array<int, 5> a = {1, 3, 5, 7, 9};
    for (int i : a)
        std::cout << i << " ";
    std::cout << std::endl;
}
```

Note that `i` is referred to by value so that any local change to `i` would have no impact on `a`. If we wanted to change the elements—say, add one to each—we would use `for (int& i : a)` instead. On a similar note, if the value type of the container is large—not just an `int` like here, then use `const &` or (if you need to change the elements) references.

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 in the case of `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; // Stands for a vector of integers

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

typedef std::vector<VI> VVI; // Vector of vectors of integers, i.e. a 2D matrix

int n = 100, m = 50;
VVI dp(n, VI(m, -1)); // A 2D vector, i.e. an 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 (amortised constant time). However, 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 complexity.

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

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

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	$O(n)$ ($O(1)$ amortised)
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 *amortised complexity* $O(1)$. An *amortised 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 amortised 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.

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

Exercise 3.2.1 (Vectors)**.** In this exercise you do some operations on 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 $\texttt{a}_0 \ \ldots \ \texttt{a}_{n-1}$, separated by a space, such that $-10^3 \leq a_i \leq 10^3$, 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`

The class `std::deque` (pronounced as 'deck', stands for **D**ouble **E**nded **Que**ue) is a vector-like indexed sequence container allowing fast insertion and deletion at 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	$O(n)$

3.3 Sets and Maps

Set, `std::set`

The class `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 use a set on a custom data type one needs to define the `<` operator on that data type (see Section 4.1). Internally, sets are usually implemented as *red-black trees*. In order to use them, 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};
std::set<int>::iterator it = s.find(3); // First one needs an iterator pointing to the element
s.erase(it); // it points to the element '3', which is then removed from s.
it = s.find(3); // now it == s.end() because 3 is not contained in s anymore
// => doing s.erase(it) here would lead to problems...
```

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 3.2).

Auto. When working with containers and generic data structures in general, an explicit referral to associated types can become a lengthy affair. As an example, consider the code from above where we wrote `std::set<int>::iterator` only to save the return value of the `find` function call. Given that the return type of a function is (well-)defined, why do we have to explicitly specify it still? Actually, we do not... The compiler can deduce this type and we can explictly ask for this service by declaring a variable to be of type `auto`.

```
std::set<int> s = {1, 3, 5, 7, 9};
auto it = s.find(3); // Let the compiler deduce the type of it
```

Similar to the variable in a range-based for loop (see above), you can also declare a variable as `auto&` or `auto const&` if you want to get a reference instead of the default value copy.

When to use and when not to use `auto` is a matter of taste. We recommend to not put it everywhere, but consciously use it where it really helps. Having an explicit type in the declaration can make code more readable because it serves as a visual assertion of the form 'here we have this specific type', and in case of a mismatch, the compiler will inform you about it straight away. Also, a statement like `auto i = 1;` is correct but makes little sense if only because `auto` is one more character to type than `int`.

Finally, note that when using `auto` on iterators, such as `begin()` and `end()`, of a non-const container, you also get non-const iterators. In order to obtain const iterators, use `cbegin()` and `cend()` instead.

```
std::vector<int> s = {1, 3, 5, 7, 9};
auto it1 = s.begin();  // it1 is of type std::set<int>::iterator
auto it2 = s.cbegin(); // it2 is of type std::set<int>::const_iterator
```

More operations on `std::set`. 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 `lower_bound` and `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).

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

Exercise 3.3.1 (Sets)**.** In this exercise you do some operations on 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.

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

- 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 $-10^3 \leq b \leq 10^3$, 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`

The class `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.

```
#include <unordered_set>
```

The table with time complexities of the most important operations on 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 operations have constant complexity 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 $O(n)$.
$\Longrightarrow$ Do not use hashtables blindly and expect average constant time complexity.

Map, `std::map`

The class `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 use a map for a custom data type one needs to define the `<` operator on that data type (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
```

The 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).

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	$O(\log n)$

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

Exercise 3.3.2 (Maps)**.** In this exercise you do some operations on 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 $0 \leq a \leq 10^3$ 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`.

IMPORTANT! 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. A d-dimensional `std::vector` is much faster, with $O(1)$ access vs. $O(\log n)$ access (or depending on the hash function, see below). However, `std::map` can be useful if the DP state consists of several different data types, not only integers.

Unordered map, std::unordered_map

The class 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.

```
#include <unordered_map>
```

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 $O(n)$.
$\Longrightarrow$ Do not use hashtables blindly and expect average constant time complexity.

3.4 Stack, Queue, Priority Queue

Stack, std::stack

An 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 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	$O(1)$

Exercise 3.4.1 (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 neighbour 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`

An `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	$O(1)$

Exercise 3.4.2 (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`

An `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 `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;
}
```

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	$O(n)$ ($O(\log n)$ amortised)
Access 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 that is capable of handling custom structures. In order to do this, one needs to define a linear order on the custom structure and pass it to the priority queue; see Section 4.2 for details.

Chapter 4

Pearls of Wisdom

4.1 Algorithms

The C++ standard library provides many reusable algorithms. Some that we consider particularly useful for solving Algorithms Lab problems are listed in this section.

Swap. The generic function `std::swap` exchanges the values of its two arguments, which should have the same type. Although its implementation is kind of trivial, using it over explicit code improves code readability and saves you from defining a temporary variable. It is defined in the `utility` library.

```
#include <utility>
```

In contrast, all the following functions are defined in the `algorithm` library.

```
#include <algorithm>
```

Minimum and maximum. There are a few very basic but still very useful generic operations for ordered types.

- The function `std::min` returns the minimum of two arguments.
- The function `std::max` returns the maximum of two arguments.

There are also generic functions `std::min_element` and `std::max_element` to compute the minimum or maximum, respectively, in a given range.

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

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

Finding values in sorted sequences. Another pair of functions that are worth mentioning are `std::lower_bound` and `std::upper_bound`. The function `std::lower_bound` computes the first element in a given range `[f, l)` 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 non-decreasing 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 $O(n)$.

```
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` member 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 $O(n)$.

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.[1]

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
```

Set operations. Lastly, the `algorithm` library also contains several useful functions for working with sets that are represented as ordered sequences—including as an `std::set`, but not necessarily so. These function include:

- `std::set_intersection` to compute the intersection,
- `std::set_union` to compute the union,
- `std::set_difference` to compute the difference, and

[1] This function is replaced by the function `std::shuffle` in C++17. However, the syntax of `std::shuffle` is more involved because it requires a generator and a seed or random source to be specified. Thus, here we stick with `std::random_shuffle` for as long as we can. Also note that some implementations of random sources try to draw randomness from hardware parameters, which does **not** work on Code Expert because you do not get to directly access the underlying hardware for obvious reasons.

- `std::set_symmetric_difference` to compute the symmetric difference (union minus intersection) of two given sorted ranges.

Numerical operations. For the type `double`, there are a few useful basic operations beyond the arithmetic ones. To use them, you need to include the `cmath` library.

```
#include <cmath>
```

- The function `std::abs` computes the absolute value of its argument.
- The function `std::floor` computes the largest integer not greater than its argument.
- The function `std::ceil` computes the smallest integer not less than its argument.
- The function `std::sqrt` computes the square root of its argument. Given that the result is stored in a `double`, it is an approximation only in general. Yet this approximation is required to be exact within the limits of `double`. That is, the result is the same as if you take the real result (with infinite precision) and then round it to the nearest `double` representation.

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 (const Edge& e : edges)
        std::cout << e.weight << "\n";
```

```
}
```

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; // Pair of integer and integer

// 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 (auto const& i : pairs)
        std::cout << "(" << i.first << ", " << i.second << ") ";
    std::cout << std::endl;
}
```

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>

int main() {
  typedef std::pair<int, int> PII; // Pair of integer and integer
  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 (auto const& i : pairs)
        std::cout << "(" << i.first << ", " << i.second << ") ";
    std::cout << std::endl;
}
```

Exercise 4.2.1 (Sort)**.** In this exercise you sort sequences 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 $\mathtt{a}_0 \ \ldots \ \mathtt{a}_{n-1}$, separated by a space, such that $-10^3 \leq a_i \leq 10^3$, for all $i \in \{0, \ldots, n-1\}$.
- 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 Mixed Examples

- Iterating through elements of a vector

```
std::vector<int> v = {1, 3, 5, 7, 9};
for (auto it = v.cbegin(); it != v.cend(); ++it)
    std::cout << *it << " ";
std::cout << std::endl;
```

- Searching for an element in a container

```
std::vector<int> v = {1, 3, 5, 7, 9};
auto it = std::find(v.begin(), v.end(), 42);
if (it != v.end()) {
    // 42 is in v at position it. 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 problems for the Algorithms Lab are installed on the student lab computers under Linux. Use whatever editor you are most comfortable with. If in doubt, we suggest to use `gedit`, `vim`, or `emacs` to edit your source code.

When working on your own computer, you have two options: (1) a native installation of the needed tools (the GNU compiler collection, the Boost Graph Library BGL, and the Computational Geometry Algorithms Library CGAL) or (2) a ready-to-use VirtualBox image that we provide. On Linux and MacOS both options work fine, while on MS Windows we recommend to use the VirtualBox image. Regarding both options, please check our technical instructions page for instructions and answers to frequently asked questions.

Even if you use your own computer to solve the problems over the semester, you should check out the setup on the student lab machines in ETH HG at some point. This is the setup you will use during the exam.

5.1 How to Compile

You need to compile your source code into a system dependent executable in order to run it. To compile your source, type the following command in the directory that contains your source file:

```
g++ -std=c++14 -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.

`-std=c++14` There are several different ISO standard versions for C++. For the Algorithms Lab we use C++14.

`-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 'Optimisation'!)

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

`-o <BINARY>` Place output in file `<BINARY>`. (Something like '`funnyexercise`'.)

5.2 How to Compile a CGAL Program

When using additional libraries, such as BGL or CGAL, a number of additional options have to be supplied when compiling and linking your source code. In order to save the work of typing these options, we use the command `make` and the cross-platform build tool `cmake`. To compile a program that uses CGAL[1], follow the steps described below.

Let us assume that you are in a directory where your source code appears in a file named `mycode.cpp`. In this directory, issue the following three commands:

```
cgal_create_cmake_script
cmake .
```

This creates a local makefile with an entry for every `.cpp` file in the current directory. If you do not have any such file, nothing happens. If you add a(nother) `.cpp` file later, you have to rerun the above commands to handle it. Otherwise, running these commands one single time suffices.

From now on, you can compile your program using the command

```
make
```

As a default, makefiles are created in a so-called release mode, with optimisations. To switch debug mode, run

```
cmake -DCMAKE_BUILD_TYPE=Debug .
```

To go back to release mode, run

```
cmake -DCMAKE_BUILD_TYPE=Release .
```

If you want to see the actual compiler and linker commands issued during the build, add the option `'-DCMAKE_VERBOSE_MAKEFILE=ON'`.

5.3 How to Run

Once you have compiled your code, you can run it. Since your program—as explained in Section 2.1—reads the input from `stdin` and writes all output to `stdout`, the program will wait for input after the start.

You can now use your keyboard to type some test input. But usually it is more convenient to prepare the input in a text file and then redirect the standard input to read from this file. In the command-line (terminal/console) it looks like this:

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

The content of `inputfile.in` is sent to `stdin` after starting the executable `funnyexercise`. The output is still printed on your screen. In order to also redirect the output to a file, use the following command:

[1] The same procedure also works for a program that uses BGL only or that uses none of the additional libraries.

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

The content of `inputfile.in` is sent to `stdin` after starting the executable `funnyexercise`, and the output is written to the file `funnyoutput.out`.

It is very useful to compare the output of your program to the expected output of the respective test set. If this expected output is available in a file `outputfile.out`, you can use the `diff` command to compare both:

```
diff -ub funnyoutput.out outputfile.out
```

If the output of your program matches the output given in `outputfile.out`, running the above command gives no feedback. Otherwise, you will see which lines differ and how exactly they differ.

You can also directly compare your program output to the expected output, without saving it to a file first:

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

You may want to check out the tools directory on `algolab.in`. There are, for instance, helper scripts to conveniently run your code on several different test sets.

5.4 How to Debug

With the `cassert` library you can state implicit assumptions in your code as *assertions*. 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;
std::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
```

Note: Assertions are by default disabled on Code Expert. In order to use them you have to add `#undef NDEBUG` as the first line of your `.cpp` file. But be careful: Checking assertions takes time!

As far as the Algorithms Lab is concerned, no advanced debugging and memory management tools are necessary. Simply reading the code and writing printouts at specific places of your code should be more than enough for all debugging purposes. If, however, you do want to use some proper tools, we recommend taking a look at the GNU debugger `gdb` for debugging and `valgrind` 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 blue 'test' button (flask icon) in Code Expert you can inspect the time limits for the exercise. This is the maximal CPU-time your program is allowed to run in order to come up with the correct solution for the corresponding 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 $n \leq 5 \cdot 10^4$ and the time limit for the solution is set to 3 seconds, you might reconsider your choice and go for a more complicated $O(n \log n)$ algorithm. This type of reasoning also works the other way around: knowing how large the input can be, you may be able to 'guess' what type of algorithm you are expected to come up with in order to solve it.

When estimating runtime bounds for large testsets, do not consider (that is, blindly multiply with) the maximum number of test sets to obtain a worst-case bound. Often there are only very few instances that stretch to the limit, combined with a number of smaller instances of various sizes. In the groups of test sets for some problems, group i is a strict subset of group $i + 1$.

Beyond asymptotic runtime bounds also be mindful of multiplicative constants. In a real timed environment you will notice if an algorithm runs four times slower or faster. And in a world where n is about a million, we have $\log n \approx 20$. So you should not think of 20 as a constant, really.

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, and follow our recommendations for reading input.
- The input might be given in an adversarial way. Try randomly permuting the input data by using the `std::random_shuffle` function.
- Make sure you pass non-primitive variables to functions by (const) reference and not by value. Copying large structures on every function call can cause a major slowdown.
- Maybe try a binary or exponential search instead of a linear search?
- Can you save time by caching and reusing results of computations rather than recomputing everything from scratch over and over again?