Univerzita Komenského v Bratislave
Fakulta Matematiky, Fyziky a Informatiky

Effective implementation and testing of algorithms

(Diplomová práca)

Bc. Peter Perešíni

Evidenčné číslo: 13029f46-b464-4cff-abf5-c279e85fdcb0

Školiteľ: RNDr. Michal Forišek, PhD.

Bratislava, 2011

Univerzita Komenského v Bratislave
Fakulta Matematiky, Fyziky a Informatiky

Effective implementation and testing of algorithms

(Diplomová práca)

Bc. Peter Perešíni

Študijný program:	Informatika
Študijný odbor:	Informatika (4017)
Školiace pracovisko:	Katedra Informatiky
Školiteľ:	RNDr. Michal Forišek, PhD.
Miesto a rok predloženia:	Bratislava 2011

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Peter Perešíni
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický

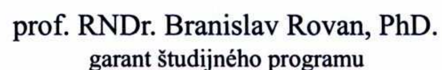
Názov : Effective implementation and testing of algorithms

Cieľ : The goal of the thesis is to create a set of implementations of various algorithms. The implementations should try to minimize the descriptive complexity, while maintaining efficiency and correctness. One intended application of the thesis is the preparation of future ACM ICPC teams.One part of the thesis should be focused on developing a framework for automated testing of such implementations.

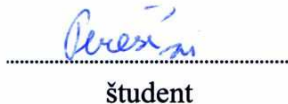
Vedúci : RNDr. Michal Forišek, PhD.

Dátum zadania: 20.11.2009

Dátum schválenia: 18.02.2011



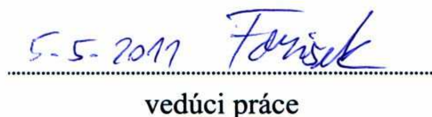
prof. RNDr. Branislav Rovan, PhD.
garant študijného programu



študent

vedúci práce

Dátum potvrdenia finálnej verzie práce, súhlas s jej odovzdaním (vrátane spôsobu sprístupnenia)



vedúci práce

I hereby declare that I wrote this thesis by myself, only with the help of the referenced literature, under the careful supervision of my thesis advisor.

. . . .

I would like to thank to my advisor RNDr. Michal Forišek, PhD. for his great advice and useful comments.

Abstrakt

Názov práce: Effective implementation and testing of algorithms
Autor: Bc. Peter Perešíni
Vedúci práce: RNDr. Michal Forišek, PhD.
Abstrakt: Práca sa zaoberá implementáciou efektívnych algoritmov. Dôraz implementácie však nie je kladený na rýchlosť, ako obvykle, ale algoritmická a popisná zložitosť. Cieľom práce bolo navrhnúť implementácie algoritmov, ktoré sú rýchlostne porovnateľné s bežnými implementáciami a zároveň sa jednoducho píšu / pamätajú. Jedno možné použitie práce je pri príprave študentských tímov na súťaž ACM ICPC.
Kľúčové slová: Efektívne algoritmy, C++, STL, jednoduchá implementácia

Abstract

Title: Effective implementation and testing of algorithms
Author: Bc. Peter Perešíni
Supervisor: RNDr. Michal Forišek, PhD.
Abstract: In the thesis we are implementing a set of various algorithms. The implementations try to minimize the descriptive complexity, while maintaining efficiency and correctness. One intended application of the thesis is the preparation of future ACM ICPC teams. The first part of the thesis is focused on enumerating possible problems and developing a framework for automated testing of such implementations.
Keywords: Effective algorithms, C++, STL, simple implementation

Contents

1 Introduction **1**
- 1.1 Thesis objectives 4
- 1.2 Outline 6

2 Typical errors and problems with implementations **7**
- 2.1 Bad pseudocode 7
- 2.2 Bad preconditions 10
- 2.3 Coding errors 11
- 2.4 Problems with integer numbers 12
  - 2.4.1 Representation of integral numbers 12
  - 2.4.2 Integer overflows 14
  - 2.4.3 Integer overflows in memory access context 17
- 2.5 Floating point problems 24
  - 2.5.1 Floating-point representation 24
  - 2.5.2 Problems with IEEE754 floats 25
  - 2.5.3 Summary 27
- 2.6 Function call stack limitations 28
- 2.7 Hardware errors 31

3 Infrastructure **33**
- 3.1 Programming language selection 33
- 3.2 Testing infrastructure 34
  - 3.2.1 Testing framework 34
  - 3.2.2 Managing compilation of large codebase 35
  - 3.2.3 Benchmarking 36
- 3.3 Preconditions – asserts for inputs 37
- 3.4 Design choices 38

4 Our implementation **41**
- 4.1 Math algorithms 41
  - 4.1.1 Modular arithmetics 41
  - 4.1.2 Modular inverse 44
  - 4.1.3 Primality testing 45
  - 4.1.4 Factorizing integers 48
  - 4.1.5 Binary search, root of a function, convex function minimum 49
  - 4.1.6 Rational numbers 50
- 4.2 Computational geometry 50
  - 4.2.1 General problems with geometry 50
  - 4.2.2 Line segment intersections 52
  - 4.2.3 Angle comparison 52
  - 4.2.4 Convex hull 53

4.3 Algorithms on strings/sequences 54
4.3.1 String search 54
4.3.2 Theoretical analysis of the rolling hash 56
4.3.3 Suffix arrays 58
4.3.4 Longest common subsequence 62
4.3.5 Minimal cyclic shift 63
4.4 Balanced structures 63
4.4.1 Interval trees and balanced data structures 63
4.4.2 Skiplist 64

5 Conclusion **67**

Source code of selected algorithms **69**
.1 Benchmarking 69
.2 Preconditions 71
.3 Sample code – binsearch 72
.4 Sample test – heap operations 75

Chapter 1

Introduction

Algorithms are the basis of informatics, no one can argue about that. Therefore, it is only natural that we need to implement them in our everyday life. But implementing algorithms is not an easy task. There are numerous problems concerning implementation of well-known algorithms. The reader interested in the most outstanding issues can consult chapter 2 where we will go through several types of problems one can probably create, implementing even very easy alogrithms. The conclusion of that chapter is that implementing good, efficient and most importantly **correct** algorithms in current programming languages like C++ is really hard. Therefore, the reader may wonder how one can obtain such good implementations from the community.

We considered this question and the conclusion is that there are several different ways how implementations of algorithms can be obtained or programmed easily:

- *Use an existing library.* This is probably the most promising way how to obtain implementation. One of the advantages of this method is that these implementations are (usually) tested by many users. Also, implementations of most used algorithms are highly optimized for concrete platforms. Thus well-known libraries that are tested by many users seem to be the best solution for our problem of finding good implementation.

  However, there are several drawbacks to this method. First of all, external libraries are big. It may be painful to use them for small projects. As an example, imagine you want to do a simple depth-first search. You may use the *Boost* graph library (`http://www.boost.org/doc/libs/1_45_0/libs/graph/doc/index.html`) for this task. If you use only this one simple search in your whole program, this may be hurting productivity for several reasons.

  First of all, you need to download the graph library itself, which depends on some other Boost libraries and you will probably end up including several Boost libraries in the project. This might add several thousand lines of code that needs to be compiled. Moreover, the Boost graph library is highly templated and the compilation time will be far greater than when compiling a simple algorithm. What is more, because of templates, you cannot compile the library as a shared object and just link it to the main project later - you will need to recompile everything each time you want to build your project.[1] Now, introducing a delay of several seconds to the compilation can slow down your development even in order of magnitude, mainly if you are programming with agile techniques and using a lot of instant unittesting. In a large project, which will do a lot of operations on graphs, this is outweighted by the benefits of the library, but for small projects it is an unnecessary complication.

  The second drawback is that you need to update the library itself once in a while

[1] This is not necessarily true for newer languages like Java or C++0x.

(bugfixes, updates, ...). The last but not least drawback is that the library licence may not suit you project, for example when you develop commercial applications.

- *Take a book or an article with pseudocode an rewrite it into actual code.* At first sight, this indeed seems to be a good solution. Pseudocode from articles and books is written in a simple way, there are comments and explanations on how it works, even proofs of correctness. So, you can rewrite the pseudocode and be happy that it is working. But this is not necessarilly the case. Pseudocode may be formally proved, but it lacks "real tests" because it cannot be tested. Thus you rely only on the author that made the proof of correctness. Also, the pseudocode may be correct from the mathemathical point of view, but it may fail on the hardware itself. There are numerous problems created when rewriting something from pseudocode to a native implementation. They are listed in chapter 2, but to give the reader a fast glimpse of them: representation of real numbers and severe problems with them, memory consumption (recursive calls), paralellisation (threading) issues, integer overflows/underflows, buffer overflows. Therefore, it is really hard to rewrite pseudocode to a correct implementation. This thesis aims to do exactly that.

- *Use the Internet as the source of solutions.* In short – never believe sources that are not thoroughly reviewed. The Internet is a medium with millions of solutions to millions of situations. But the quality of contributions is often very questionable. If you need to use the Internet, here is a recommendation: take the idea from the forum/post/whatever, but *implement it yourself and think twice before each step.* Many people post only partial solutions or solutions which do not work. Moreover, there are numerous solutions with security concerns. So, stay away. A slightly better case is taking source code of some opensource project and reusing it. But still – check if the code is well tested before you consider including it in your project.

  We will use a simple example to demonstrate the problems with believing internet forums. In our example, we needed an implementation of function `isPrime` which can take an integer argument and will return a boolean – true if the supplied integer was prime and false otherwise. We will look for an implementation in the Python language. To obtain such implementation, you can search some big search engine for "python isPrime". The results, of course, may depend on your search engine and time of the search and many other factors. For this example, we queried the Google search engine for `http://www.google.sk/search?q=python+isprime` and took the *first* result, which was in time of this writing `http://www.daniweb.com/forums/thread70650.html`. The author of the thread asks a simple question

  > "I need a function that returns True or False if an integer n is a prime. Any 'high speed' thoughts?"

  We picked one of the replies:

  > "Our snippet examples give you a list of primes. You could get a list of primes in the range of your numbers, and then see if your numbers are in it. For simplicity sake you can use this small function ..."

  with the source code shown in listing 1.1.

  Our claim is that the code is written really poorly. To support this, we will look more closely at it and try to explain its problems:

  First of all, Python is a language without strict types. This means that you may pass **any type** of variable as `n`. Now, the reader may wonder what happens if he/she calls the

Code Listing 1.1: Bad isprime() implementation in Python

```
# prime numbers are only divisible by unity and themselves
# (1 is not considered a prime number)

def isprime(n):
    '''check if integer n is a prime'''
    # range starts with 2 and only needs to go up the squareroot of n
    for x in range(2, int(n**0.5)+1):
        if n % x == 0:
            return False
    return True

# test ...
print isprime(29)    # True
print isprime(345)   # False
print isprime(8951)  # True
```

function with a string or a list argument. The program will fail in some unpredictable way. In the cases discussed above it will fail to compute the square root.

But there are more subtle errors: try to call `isPrime` with a floating-point value and the function will work. This starts to be interesting – if you read the Python manual about modular division, you can spot the sentence

> "The arguments may be floating point numbers, e.g., 3.14%0.7 equals 0.34 (since 3.14 equals 4*0.7 + 0.34.)"

So, the function will not fail and will return some unpredictable value.

This may seem to be ridiculous critique, but you may bet that sooner or later, someone will call the function with a floating-point value [2] and he/she will be wondering why it is not working correctly.

After we considered that the function is totally ignorant about the type of the argument and that this ignorance may lead to undefined behaviour, we may continue listing the problems, because this certainly is not the end of it. Even if we pass an integer to this function, there is still plenty of room for mistakes. Try passing a negative integer and the function will fail to compute the square root (and this is still the better case).

And finally, let us examine the function's behaviour on valid inputs. Even if you are really cautious and there is no call to the function with invalid arguments (which we cannot believe unless you have a very simple 10-lines-of-code program), the result of the function may be wrong. Try zero, and `isPrime` will return `True`. But zero surely is not a prime. And the most obvious problem – the function documentation itself states that **1 is not a prime**. Try this function and see for yourself.

To summarize our objections to the function, here is an example of the bad function calls:

```
>>> print isprime("aa")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in isprime
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'float'
```

[2] In an untyped language, the type of the variable may easily change and after some math operations one may end up with floating point instead of integer.

```
>>> print isprime(4.7)
True

>>> print isprime(-2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in isprime
ValueError: negative number cannot be raised to a fractional power

>>> print isprime(0)
True

>>> print isprime(1)
True
```

So, as you can see, these few extremely simple lines of code have several serious flaws. And this was only a few lines - try to copy one hundred lines of code and ask yourself the question “Do I believe this code?”. The answer should be “never unless it is well documented, well tested, I can see that the tests are covering corner cases, impossible imputs, etc. ”

To summarize the ways of obtaining implementations: there are good sources like big opensource libraries which are well tested, but there is a problem with their size, and subsequent complications when the project uses only minimal functionality of the library. This thesis therefore aims to fill this gap by providing correct implementations of simple and often-used algorithms, which are thoroughly tested, well documented, and the reader may not fear to copy/adjust the implementation.

1.1 Thesis objectives

The main goal of the thesis is to create a set of implementations of well-known algorithms. The task itself seems to be quite easy, but we can assure the reader that it is a very hard job. In the thesis, we needed to join algorithmic thinking with very deep knowledge of programming languages and computer architecture. Moreover, it was needed to provide a great deal of ingenuity to adapt the implementations to be as simple as possible, as error-free as possible and very well documented. We can also note that people writing such algorithms must know a variety of tips and tricks which cannot be found in standard textbooks for programming courses.

In this section, we will gradually introduce the goals of the thesis. At first we will discuss the testing infrastructure, then we will discuss the selection of the programming language for the thesis.

Documentation

Any implementation which is not well documented is usually hard to maintain and there is no one except the author of that code (or maybe even not the author) who understands it. The goal of this thesis is the opposite. We wanted to create implementations which are thoroughly documented, so that

1. everyone can understand the code well and can see what it is doing. This is important because our goal is not to provide a complex library, rather we are writing a set of

small programs, which everyone can take and rewrite from the paper to suit his/her needs. Usually these algorithms will be adapted for specific tasks and there will be an alternation of input/output representation like using arrays versus std::vector, a slightly different computation, or reporting results in a slightly different format.

2. everyone can see algorithm invariants, preconditions and postconditions. This part may be especially interesting for those who will read the next few chapters about possible problems with architecture. In short, we will document (and check in the code) for extra conditions so that the algorithm is guaranteed to be correct. Often, this is underestimated in trivial implementations, as someone wants to have a quick and dirty function in the smallest possible time. We think that properly checking for overflows, bad inputs, invariants, etc. is a part of good programming practice and helps other people when something goes wrong.

Error-free implementation

It may seem to be an easy job. As we said earlier – it actually is not. We noted in the previous paragraph that the code should be robust enough to handle all corner cases. The code should also handle invalid arguments and properly respond to them. It is much easier to debug the code if you call the function with bad parameters and the function will instantly fail/throw an exception. On the other side if the function will somehow work and it will fail in some unpredictable manner hundreds of lines later, it will be a debugging nightmare. To fulfill this objective, we analyzed the potential problems of the implementations and later we were guided by that analysis. Note that this list might not be complete and we may have missed some kinds of errors. These issues should be addressed with thorough testing.

Testing

The basis of the thesis lies in testing the written algorithms. We try to put the algorithms to as extreme conditions as possible and see if they behave correctly. The functions, for example, should check for invariants, corner conditions, report when they are unable to perform some operations[3] and so on. Moreover, we usually implement more than one algorithm computing the same problem and we cross-check the implementations on various kinds of inputs.

The most important point of the testing is that it should be automated. The rationale behind this is to enable the reader to

- quickly reproduce the test. If someone tells you he/she has tested the code, will you believe him/her? Of course not. The best way to be sure is to run automatic tests and see for yourself that everything is fine. Also, if you are on some strange architecture, a different compiler or some awkward hardware configuration, you need to test the algorithms to see that nothing went wrong – maybe the author missed some of the possible differences. An example of a such difference are various sizes of the `int` type on different platforms, 32-bit versus 64-bit pointers, big-endian vs. little-endian, etc..
- see what types of tests have been done. Looking at the testcases the reader can get an impression about what has been actually tested. Moreover, if there are many tests testing different kinds of corner cases, overflows, invalid inputs, ... the reader may assume the correctness of the algorithm and gain trust in the implementation itself.
- see the usage. The tests are probably the best documentation available – they show the correct and incorrect usage of the algorithm, they show boundary conditions, problematic cases. The reader may therefore think about the limitations of the algorithm and be cautious when using it.

[3]For example we passed too big arguments to them, or something went wrong during the computation

Benchmarking

Testing the algorithm validity is only one part of the testing. We also need to test the performance of the algorithms, find bottlenecks, compare various (simple and easy to understand) optimizations, compare different algorithms. Benchmarking is quite useful, because there are usually more algorithms solving one problem and these algorithms differ in code complexity and asymptotic time complexity. Therefore, it is very useful to know the expected performance of various implementations for a wide range of input sizes – people wanting to compute a specific task may then decide which algorithm to use. Usually, the asymptotically faster algorithms are much harder to implement, have a bigger O constant and for reasonably small inputs are often slower than the simple ones.

1.2 Outline

In the current chapter, we showed the motivation and the objectives of the thesis. Chapter 2 summarizes the potential problems of the implementations, considering hardware and software limitations. Chapter 3 shortly discusses the target architecture and infrastructure we set up before working on the implementations. Chapter 4 contains a comprehensive summary of all algorithms we implemented, notes about potential problems in the implementations, results of benchmarks, etc. The thesis results are summarized in chapter 5.

Chapter 2

Typical errors and problems with implementations

There are many issues and problems which may arise during the implementation of the algorithms. In this chapter we want to give the reader a comprehensive overview of the most important problems. We will not consider and explain all of the possible errors/problems, as this may end up with a list several times exceeding the supposed length of the thesis, and we may still still omit some of them.

At the beginning of the chapter we will dig into problems caused by bad pseudocode or by a lack of concentration. Later we will show the reader some of the more serious errors connected with computer architecture. These are usually much harder to find and fix because they are less-known, especially among inexperienced programmers. Finally, at the end of the chapter we are discussing errors which may come from the hardware itself even when the software is correct.

2.1 Bad pseudocode

Even when taking pseudocode from a peer-reviewed conference, one may not be one hundred percent sure about its correctness. For example there may be printing issues like a missing equals sign from the "$\leq$" character. But more importantly, the pseudocode is not a real implementation. The pseudocode was never compiled and ran as there is no compiler for pseudocode. Thus we can state the fact that *the pseudocode cannnot be tested on real inputs.* Of course, it may be formally tested by specifying invariants and using some rigorous mathematical method for proving its correctness. But usually the authors prove only the invariants that are interesting for the sake of the problem they are solving. Moreover, the core of the algorithm is more interesting and gets much more attention in the article than corner cases. But in the real implementation we need the exact opposite. We need an implementation which can handle any corner case. In fact, in real-world scenarios the special cases and invalid input checks may form a large portion of the implementation itself.

Also, there are sometimes hidden dependencies in the pseudocode like advanced data structures. The programmer needs to implement not only the algorithm from the pseudocode but also many data structures that are "well-known". But "well-known" does not mean that they are easy to implement and that the programmer will not make errors there.[1].

But back to the bugs in pseudocode. In the rest of this section we will present three pseudocode examples. The first two were encountered during the actual implementation of the thesis (and we spent several hours desperately trying to find the bug in our code instead

[1]In fact, if the underlying structures are well-known and used, they are probably available from standard libraries

of the original pseudocode). The third example is taken from the article [O'N08] where the authors explain that the "widely-known implementation of the Eratosthenes sieve in Haskell" is in fact a completely different algorithm.

Corner cases

In the first example, we will discuss algorithm 3.1 from [Duv83]. The algorithm should find the minimal cyclic shift of the supplied input word in linear time.

Figure 2.1: Duval's minimal cyclic shift algorithm pseudocode

```
Input: a word string  a_1,...,a_n  of  letters  over  A.
Output: the table  M[1...n]  according  to  (3.1).
{MINSUF: array  [1...n]  of  integers;  f[2...n]:  array  of  integers}

begin:  k:=0; j:=2; M[1]:=0; f[2]:=1;
    while j <= n do begin
        i:= k + f[j - k];
    99:  case  "compare a_i::a_j" of
      1 {a_i < a_j} : (M[j]:=k; i:=k+1; j:=j+1; f[j-k]:=i-k; goto 99)
      2 {a_i = a_j} : (M[j]:=M[i]+j-i; i:=i+1; j:=j+1; f[j-k]:=i-k; goto 99)
      3 {a_i > a_j} : (k:=M[i]+j-i;
                       if k=j-1 then begin M[j]:=k; j:=j+1 end)
      endcase
    endwhile
end
```

The problem with this pseudocode is in the `goto 99` statement – the program may end up with the variable `j` being $n+1$, which is out of bounds. Probably the smallest input showing this error is $a =$ 'xy'. In the first iteration of the `while` cycle the variables i and j are set to 1 and 2 respectively. The algorithm will take the first case of the switch and after the `goto 99` statement the state will be $i = 1$ and $j = 3$. Now we are out of bounds for the comparison.

Moreover, the programmer may be clever and "optimize" the statement `goto 99` into a more readable form shown in the following figure:

Figure 2.2: "Optimized" `goto 99` statement

```
while (j <= n) and (a_i <= a_j) do
  if (a_i < a_j) then
     ...
  else
    ...
  fi
endwhile
```

This is probably easier to understand than the `goto 99` syntax. This transformation of the code also solves the mentioned out-of-bounds issue.[2] The problem with this "optimization"

[2]Note that this optimization causes the side-effect of changing the behaviour of the algorithm, but one cannot figure this out unless he/she explicitly takes the out-of-bounds possibility into account!

is that it will create a new error. To be more specific, the program may eventually step into line 12 with j holding the value $n + 1$ and the assignment to M[n+1] is out of bounds again.

Print pixies

The second example of pseudocode containing an error is based on the article [BK03]. The algorithm presented on page 66 shows how to verify that a given array is indeed a suffix array of some sequence. In Theorem 2, the authors mention the SA[0,n) array, i.e. the array SA is

Figure 2.3: Karkäinnen's suffix array verification pseudocode

```
for i=n-1 downto 0:
  A[s[SA[i]]] = i
A[s[n-1]]++ // skip SA[i]=n-1
for i=0 to n-1:
  if SA[i]>1:
    c = s[SA[i] - 1]
    check SA[A[c]] + 1 == SA[i]
    A[c]++
```

indexed from 0 to $n - 1$. The algorithm itself (line 3) supports this claim. Hovewer, on line 5 the algorithm checks that SA[i] > 1 and then uses the character s[SA[i] -1]. So, the original algorithm **never** reads the first character s[0] of the string s.

Indeed, the correct version of line 5 is if SA[i] > 0 or if SA[i] >= 1. We suppose the authors of the article meant the second alternative and there was some "print pixie" which dropped the equality sign.

Bad implementation of pseudocode

We borrow the third example from the article [O'N08]. The article shows that the widely-spread functional programming implementation of "The Sieve of Eratosthenes", which is written in the programming language Haskell as

Figure 2.4: The Sieve of Eratosthenes in Haskell

```
primes = sieve [2..]
sieve (p : xs) = p : sieve [x | x <- xs, x `mod` p > 0]
```

is **not** the Sieve of Eratosthenes. In fact, the algorithm is correct (it computes primes), but its running time is even worse than trivial division! This is a nice example of how implemention of pseudocode in a (functional) language may turn out to be wrong.

After these three examples, we may conclude that even algorithms written in pseudocode or carefully implemented from pseudocode may contain serious flaws. These flaws are hard to spot unless the careful reader reads the whole article and verifies each claim. Therefore we conclude that it is really necessary to test the implementations, even when they are implemented from simple pseudocode and the programmer believes he/she has not made any error in the implementation.

Code Listing 2.1: Sorting real numbers with NaNs

```
#include <stdlib.h>
#include <stdio.h>
#include <algorithm>
#include <cmath>

double data[10];

int main() {
    for(int q=0; q < 10; q++ ) data[q] = rand()%1000;
    data[5] = sqrt(-1); // NaN
    std::sort(data, data + 10);
    for(int q=0; q < 10; q++) {
        printf("index %d: value %lf\n", q, data[q]);
    }
}
```

2.2 Bad preconditions

Many algorithms require some properties to hold for their correctness. We may call all such conditions *preconditions* of the algorithm. These preconditions may be trivial (argument must be non-negative), or complex. For example, the binary search algorithm implemented as `std::lower_bound` and `std::upper_bound` has a precondition that the input range should be sorted. Many times, the algorithm itself cannot check the precondition itself, because the checking would require more time than the operation itself. More algorithms of this kind are operations on the heap where the heap property must be guaranteed before the operation can start.

One of the complex preconditions required by many algorithms, especially algorithms related to sorting, such as sorting itself (`std::sort`) or binary search trees (`std::map`, `std::set`), is the presence of *total ordering*. Many people automatically assume total ordering for integers and real numbers. Moreover, ordering of more complex objects is usually based on comparing several individual fields of the structure, thus implicitly relying on the total ordering of the basic types.

The problem is that the assumption about the existence of total ordering of the basic types is wrong. The culprits are floating-point numbers. The reader may wonder how it is possible, that there exist two floating-point numbers which are not comparable. Indeed, no such pair of numbers exists. But floating points consist not only from numbers but also from special values. One such value is `NaN` – Not a Number. This value represents an error during a computation, which can be the result of an invalid operation, for example the square root of a negative number. The core of the problem with `NaN`s is that they are incomparable I.e. all the following conditions are false: `NaN < 0`, `NaN == 0` and `0 < NaN`. The reader may see that the property of total ordering is violated.

Now imagine we wish to sort an array of real numbers, but the value `NaN` was somehow introduced into the array during previous calculations. We will show that the sorting algorithm will fail to sort even the ordinary numbers (not only the `NaN` values).

Listing 2.1 contains a sample program which sorts real numbers. The input array contains the result of a bad computation. Listing 2.2 shows the "sorted" output of the program. This particular issue with floating-point numbers causes a lot of bugreports, one of them may be found at `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=41448`. The interesting part of the bugreport is from a developer who states

> "Basically when `std::sort` is given a type which is not totally ordered as required, it tends to corrupt the memory immediately before and after the given array."

Code Listing 2.2: “Sorted” output

```
index 0: value 383.000000
index 1: value 777.000000
index 2: value 793.000000
index 3: value 886.000000
index 4: value 915.000000
index 5: value -nan
index 6: value 386.000000
index 7: value 421.000000
index 8: value 492.000000
index 9: value 649.000000
```

The report itself is closed as *“this is not a bug”* – and they are correct, the well-documented precondition for `std::sort` is that the values are comparable by total ordering.

Thus, when using algorithms with complex preconditions one needs to double-check potential problems.

2.3 Coding errors

One big category of errors in the implementation are coding errors. These coding errors come from a lack of concentration of the programmer and may be simple typos. They may be hard to track by hand, but there are good static analysis tools which can handle most of such errors.

Variable name typos

In languages like Python, PHP or Javascript, you may mistype the name of a variable, function or class. The classical example is

Code Listing 2.3: Typo in the variable name

```
my_variable = 1;
print my_warialbe;
```

These errors may be easy to find if they are made when “reading” the variable, because the effect is usually an immediate error or warning and you know the line of code which contains the problem. If the typo is made while writing into the variable, it is much harder to find. A new variable will be created instead of storing content into the correct one. This may introduce problems such as data corruption and other complications, especially in complex code where there are many assignments and only one of them is a typo.

Conditional typos

This typo is probably most famous in the C and C++ context, although we have seen similar problems in the PHP language. In all these languages, you may mistype `‘==‘` inside an `if` block as an assignment `‘=‘`, leaving no trace of the problem.

Code Listing 2.4: Typo – ‘=‘ in assignment instead of ‘==‘

```
  if (a = 1) { do_something() }
```

Logical errors

Logical errors are usually bound to happen if the programmer is not concentrating on the task and does not verify boundary conditions. This includes '`>=`' versus '`>`' in conditionals, '`i++`' versus '`++i`' in complex statements like `j = array[++i]`, or wrong operator precedence '`v | mask << 3`' vs '`(v | mask) << 3`', and many more.

Similar font characters problem

The most famous problem is between the letter 'l' and the number '1'. To make things even worse, the letter 'i', which is usually reserved for iterating loops also resembles the problematic pair. With a bad font, it can be really hard to distinguish between them. Even when they appear quite differently in the font rendering, the programmer may read the wrong character if he/she is just scrolling through the code. The effect is that the programmer will get very confused, because the program does completely different things than what he/she thinks it does. To test your senses, try to find errors in the following code

Code Listing 2.5: Typo – 'l' versus '1'

```
1 long long int MAX = 111l1;
2 int i;
3 int l;
4 for (i=l; i < MAX; i++) {
5   printf("i: %d, l: %d\n", i, 1);
6 }
```

The solution is to never use the letter 'l' on its own. If the 'l' is used as a variable name, you may change it to something more appropriate. If the 'l' is part of some other construct, for example "long long" specifier to the integer constant in C/C++, usually you may write it in capital letters. The reformatted code may look like this:

```
1 long long int MAX = 111LL;
2 int i;
3 int q = 0;
4 for (i=q; i < MAX; i++) {
5   printf("i: %d, q: %d\n", i, 1);
6 }
```

2.4 Problems with integer numbers

Integer overflows are one of the most dangerous errors. They are logical errors created while implementing a specific algorithm from pseudocode resulting from a lack of precision. In this section, we will briefly introduce the problems with standard integer types and propose solutions trying to mitigate most of these errors. But before we can proceed, we need to introduce the representation of integers in the current computer architecture.

2.4.1 Representation of integral numbers

Unsigned numbers

Any unsigned integer number can be written in the form

$$\sum_{i=0}^{\infty} b_i 2^i, \quad \forall i : b_i \in \{0, 1\}.$$

Numbers b_i are called bits and they are represented in computers as the presence/absence of electric charge. But computers do not have infinite memory and the sum is cut at some point. If we use the first n bits, i.e. we use bits corresponding to numbers 2^0 (the lowest bit) up to 2^{n-1} (the highest bit), we can represent any integer number from zero up to $2^n - 1$. The values of n used in current computer architectures are 8, 16, 32 and 64. Because of the limited range of unsigned numbers, computations may "overflow" or "underflow". In the case of overflow, the result is truncated to contain only last n bits. In case of underflow (i.e. a negative result), the resulting negative number $-x$ is represented by the same number as if we calculated the representation of $2^n - x$. The representation of unsigned numbers for $n = 8$ is showed in figure 2.6

Signed numbers

Negative numbers are in mathematics represented by prefixing them with the special character (the minus sign). However, on modern computers, there is no such special symbol. Therefore it is needed to represent signed numbers only in binary code. There are several widespread representations which can be used, and we will discuss them here.

Sign and magnitude

The basic solution for signed numbers consists of reserving one bit (often the most significant bit of the number) to hold the sign instead of the value. Usually the value 0 of the sign bit means positive numbers and 1 means negative numbers. The remaining part of the number represents the magnitude (absolute value) of the number.

Consider for example 8-bit numbers. Then the value '-47' is represented as 1 (minus sign) and another 7 bits holding the value 47, i.e. together 10101111. This approach is somewhat similar to that used in mathematics, i.e. writing sign symbol and then the magnitude.

One of the drawbacks of this method is that the number zero can be represented in two different ways – either as $+0$ or -0. This may add additional complexity when using comparisons. Also, math operations with such numbers are more complicated, because one needs to determine the resulting sign and the operation. For example, addition of two such numbers is addition if the signs are equal ($++$ and $--$), but it is subtraction in case of different signs ($+-$ and $-+$).

Nowadays the idea of storing the sign bit separately is used mainly for floating point numbers and we will discuss that later in this chapter.

Excess-N method

This is the second easiest method. Sometimes it is also called *biased* representation as it uses a fixed number N as a biasing value. The value of the excess-N representation is shifted by N with respect to the unsigned representation. I.e. signed zero is represented as unsigned N, and $-N$ is represented as unsigned zero.

The excess-N representation is used primarily for the exponent of floating-point numbers.

The problem with this method is again efficiency of computations. After each addition we need to adjust the number by N. In case N is a power of two this can be implemented in hardware easily. But multiplication and division are more tedious. And probably the biggest drawback is that these numbers are incompatible with the unsigned representation – the representation of positive numbers in excess-N representation is different from the unsigned representation.

One's complement

To overcome the problem with representing the sign and the magnitude and with using both addition and subtraction depending on the signs of the numbers, another method was developed. Basically, we will "flip" the range of negative numbers. This is done by doing logical negation (not) on all bits of the magnitude.

For example let us take the number -47. The representation of 47 is 0101111, and therefore the one's complement is 11010000.

To add two numbers in this system one needs to do conventional addition with a single modification – at the end one must add any resulting carry bit back to the result. The reason for this is the presence of -0, which is different from $+0$.

The example of such computation is on figure 2.5

```
     binary    decimal
    11111110     -1
  + 00000010     +2
------------    ---
  1 00000000      0    <-- not the correct answer
           1     +1    <-- add carry
------------    ---
    00000001      1    <-- correct answer
```

Figure 2.5: One's complement example

Two's complement

The two's complement representation is the standard representation of signed integral numbers. The problem with two different representations of zero value (and thus the problem with addition of carry in one's complement) is solved by shifting the negative range so that -0 will be shifted out and the smallest(in magnitude) negative value will be -1. This number will be represented as a vector of all 1's. The two's complement representation is very convenient. For example, to add two signed numbers, one may use the same algorithm as for the unsigned representation. Multiplication is a bit trickier, but it can be done by negating the numbers and doing positive multiplication or by specialized algorithms.

One potential problem with the two's complement is that it can store a non-symmetrical range. The number represented by only its most significant bit (In 8-bit case the representation 10000000, i.e. -128) does not have its positive counterpart. We will discuss this problem later.

2.4.2 Integer overflows

The most known errors related to the limited integer representation of numbers are integer overflows. They emerge when the program is trying to do math operations resulting in a number that does not fit into the required type. Let's start with a basic example:

Code Listing 2.6: Integer Overflow

```
1 int c = 50000;
2 c = c * c;
3 printf("%d\n", c);
```

When the code is compiled and ran, it outputs a suspiciously looking negative number -1794967296. The correct result 2500000000 won't fit into signed integer type and is truncated.

Binary	unsigned representation	two's complement
00000000	0	0
00000001	1	1
00000010	2	2
...		
01111110	126	126
01111111	127	127
10000000	128	-128
10000001	129	-127
10000010	130	-126
...		
11111110	254	-2
11111111	255	-1

Figure 2.6: Two's complement representation

Sometimes, integer overflows are hard to notice. The following code tries to iterate over all values of `unsigned char`.

Code Listing 2.7: Overflow

```
unsigned char c = 0;
while (c <= 255) {
  do_something();
  c++;
}
```

The problem is, however, that the comparison is always true due to the limited range of the data type and that the loop will run forever. To be more precise, `(unsigned char) 255 + (unsigned char) 1 == 0`. A good compiler will issue a warning though.

When we are talking about overflows, we need to mention also underflows, which are basically the same type of problem but in the other direction. A popular code may look like this:

Code Listing 2.8: Unsigned underflow

```
for (unsigned int i = 0; i < n; i++) {
  if (i - 1 < 10) {
    do_something();
  }
}
```

and it does not work for `i == 0`. This suggests that "`if (i - 1 < j)`" is **not** equivalent to "`if (i < j + 1)`". You may wonder why someone would use unsigned int in the loop. The answer is, that you may not notice that the type is unsigned, consider for example `size_t`.

Another complication is that the negative value range of signed integers is bigger by one than the positive range (because zero is in the positive range). This may create strange and unexplicable behaviour:

Code Listing 2.9: Unary minus overflow

```
int c1 = -2147483647;
int c2 = -2147483648;
printf("%d %d\n", c1, -c1);
printf("%d %d\n", c2, -c2);
```

with the output

```
-2147483647 2147483647
-2147483648 -2147483648
```

Moreover, we can conclude that "`if (c == -c)`" is **not** equivalent to "`if (c == 0)`" (but it **is** equivalent to "`if (2 * c == 0)`").

Not-so-evident owerflows

The typical problem with limited ranges is that they can overflow. People are usually considering that integers may overflow in math operations. If you have math-heavy code, you will probably check for overflows and nasty corner-cases. But the question is – do you always think about this issue? Well – let's take a quiz. Consider the following code of the binary search algorithm:

```
uint bin_search(int value, uint start, uint end) {
  while (end > start + 1) {
    uint middle = (start + end) / 2;
    if (array[middle] > value) {
      left = middle;
    } else {
      right = middle;
    }
  }
}
```

Do you see the possible problems? There are two obvious errors that are hidden in this code. The first one is "What if someone calls `binary_search` with `start == MAX_UINT`?". You may say that this is a nasty input and that the function is not supposed to handle invalid intervals. Well, the function **should** warn about wrong input.

But there is a more serious problem – the algorithm **does not work** even for valid ranges. Consider calling `bin_search(47, 3000000000u, 4000000000u)`, which is a valid range of a billion integers. The value of `middle` in the first computation certainly won't be `3500000000u`. Clearly, the algorithm will not work correctly. But be frank, have you ever considered this as a potential problem with binary search? No? It cannot be ignored it as the four billion integer array used for binary search may be quite common in a few years. You may see an example of our well-documented overflow-checking implementation in appendix 3.

Integer conversions

Converting between signed and unsigned integers and between types of various lengths may also cause problems, especially in C and C++, as these conversions are done automatically based on types of operands.

These rules may even be different for various languages. Take for example the Pascal language:

Code Listing 2.10: Pascal type conversion

```
var i: integer;
    j: int64;
begin
  readln(i);
  j := i + i;
  writeln(j);
end.
```

Which gives the output

```
Input : 2000000000
Output: 4000000000
```

But now, consider the same program in C:

Code Listing 2.11: C type conversion

```
#include <stdio.h>
int main() {
  int i;
  long long int j;
  scanf("%d", &i);
  j = i + i;
  printf("%lld\n", j);
}
```

and the output is

```
Input : 2000000000
Output: -294967296
```

Now, you see that the result of $i + i$ overflows.

This can be pushed up further. Consider the following program:

```
#include <stdio.h>
int main() {
  int i;
  long long int j = 4000000000ll;
  scanf("%d", &i);
  if (j - 2 * i == 0) {
      printf("j - 2 * i = 0 equal\n");
  } else {
      printf("j - 2 * i = 0 nonequal\n");
  }
  if (j - i - i == 0) {
      printf("j - i - i = 0 equal\n");
  } else {
      printf("j - i - i = 0 nonequal\n");
  }
}
```

The output is of course a bit unexpected:

```
Input : 2000000000
Output: j - 2 * i = 0 nonequal
        j - i - i = 0 equal
```

The problem here is operator precedence: in the first case, we compute "`int 2 * int i`" with `int` result and then "`long long int j - int tmp`" with `long long int` result. In the second case, however, the computation "`long long int j - int i`" produces a `long long int` temporary and continues with "`long long int tmp - int i`" resulting in `long long int`. Based on this fact, it is obvious, that `j - 2 * i` is not equivalent to `j - i - i`.

2.4.3 Integer overflows in memory access context

Memory access is traditionally done with either pointers into the memory or a combination of a pointer to the array and index of the element in the array. Both of these methods rely on pointers fitting into some integral type.

As we will show in this chapter, doing so may introduce hidden problems. We will start with the following program:

Let's see what the problem is. Suppose we are working on a 32-bit machine. The function `resize` has signature `resize(size_t size)`, where `size_t` is defined as unsigned 32-bit integer. You may see that the conversion from the negative integer -1 to a positive `unsigned`

```
#include <vector>

int compute_vector_size() {
  // hard computation
  return -1;
}

int main() {
  std::vector<char> v;
  v.resize(compute_vector_size());
}
```

`int` of size around $4 * 10^9$ is made. The program will crash instantly on today's 32-bit computers, because you cannot allocate such a big amount of memory. But suppose we have a 64-bit architecture with 8GB of RAM and you will use this legacy 32-bit compiled code. Now, the program is able to allocate the required memory[3]. We may see that the presented inherently buggy code will run without any complaint on today's hardware. And this is the real problem of the situation. You may end up with a program eating several gigabytes of memory and you have no idea what happened.

This is not only the case for memory allocation. Now that we have considered problems of `int` conversion to 32-bit `size_t`, we can continue to put up more examples of this sort.

The other very interesting example is adapted from [LeB08]. It shows that type conversion between pointer-distance type `ptrdiff_t` and `unsigned int` can behave quite differently on different platforms:

```
void assume_shorter_than(
    int* ptr_start,
    int* ptr_end,
    unsigned int count)
{
  if (ptr_end - ptr_start < count)
    return true;
  else
    throw ArraySizeError();
}
```

The code checks that the array represented by its starting and ending pointer is smaller than specified size. Suppose some programmer will forget the correct order of the arguments and pass the pointers in the opposite order.

In that case `ptr_current > ptr_max` and the result of `ptr_max - ptr_current` is a negative number of type `ptrdiff_t`. Before the comparison it is automatically cast to (a quite big) `uint32` and if `count` is reasonably small, the code will correctly throw an error.

Of course, the analysis we just completed is missing some important bits. First of all, it will not work for very big values of `count`. For now we will ignore this particular problem. The problem which will catch our eye is 32- vs. 64-bit compilation. The problem starts with the following observation: C++ does not fix size of types. On 32-bit systems, `int` and `ptrdiff_t` are both usually 32 bits long and the program itself works. But with 64-bit architecture, there are notable differences between these types, and they are not standardized. There exist even differences between compilers/platforms.

[3]This is not entirely true, on Linux one process may allocate at most 3GB of RAM. The same limit can be reached also on Windows if you tweak boot settings with parameter "/3GB". So, the `resize(-1)` is somewhat impossible to reach. But this can be "fixed" by calling `resize(-2000000000)`

data model	short	int	long	long long	pointers/size_t	sample OS
LLP64/IL32P64	16	32	32	64	64	Microsoft Windows (X64/IA-64)
LP64/I32LP64	16	32	64	64	64	Most Unix and Unix-like systems, e.g. Solaris, Linux, and Mac OS X
ILP64	16	64	64	64	64	HAL Computer Systems port of Solaris to SPARC64
SILP64	64	64	64	64	64	Unicos

Figure 2.7: Size of integer types on different platforms

We have adapted the table shown in figure 2.7 from [Wik11] to show the reader the various platforms and the sizes of different integer types on them.

Now, let us discuss what the program does in the first model. We have "`if (ptr_max - ptr_current < count)`", which is typed as "`if (ptrdiff_t < unsigned int)`". The left side is 64-bit, right side is only 32-bit, so it is up-cast to 64 bits too. So we have "`if (small negative int64 < small positive int64)`". And our array of negative size will pass the function. This may be considered as a serious flaw.

The problems with memory will not end here. We may move a bit further. Consider a pointer somewhere into the memory. The type of pointers in C/C++ is `uintptr_t`. Suppose that we have two pointers and calculate their difference. The resulting type is `ptrdiff_t`, which is signed. Oops, somehow we may overflow. Now, in normal situations, this is not a problem as it is pointed out in `http://www.cplusplus.com/reference/clibrary/cstddef/ptrdiff_t/`.

> "`ptrdiff_t`: Result of pointer subtraction
>
> This is the type returned by the subtraction operation between two pointers. This is a signed integral type, and as such can be casted to compatible fundamental data types.
>
> A subtraction of two pointers is only granted to have a valid defined value for pointers to elements of the same array (or for the element just past the last in the array).
>
> For other values, the behavior depends on the system characteristics and compiler implementation."

But as we said earlier, the Internet is a very bad source of knowledge. In this case, the site `http://www.cplusplus.com` is wrong. For the purpose of this thesis, we looked up the exact specification of the C++ language (current C99 draft, [c9905]) with the following result:

> "When two pointers are subtracted, both shall point to elements of the same array object, or one past the last element of the array object; the result is the difference of the subscripts of the two array elements. The size of the result is implementation-defined, and its type (a signed integer type) is `ptrdiff_t` defined in the `<stddef.h>` header. **If the result is not representable in an object of that type, the behavior is undefined.** In other words, if the expressions `P` and `Q` point to, respectively, the `i`-th and `j`-th elements of an array object, the expression `(P)-(Q)` has the value `i-j` **provided the value fits in an object of type `ptrdiff_t`.**"

There are several ways how to allocate and use arrays in C++. We had examined the most prominent ones for the presence/absence of the problem with pointer difference overflow. Our experimental results are summarized based on allocation type:

- *static allocation*: Static allocation is a basic type of allocation. The allocated memory may reside in the global scope, or may be a part of a function call, then it resides inside the stack.

  The validity of allocation of too big arrays can be checked with this simple code:

```
1 #include <stdio.h>
2
3 const unsigned int Mi = 1000000;
4 char data[2500 * Mi]; // 2.5GB of memory
5
6 int main() {
```

```
7       printf("I'm running ok");
8   }
```

for which the compilation just fails:

```
>g++ alloc.cpp -m32
alloc.cpp:4: error: overflow in array dimension
```

So in this case, we are safe.

- *C* `malloc` *(or similar)*: The `malloc` function can allocate arbitrarily large chunks of memory (up to size `size_t` or the OS limit). `malloc` however does not return “an array of objects”, but rather a continuous block of the memory. Thus, we may consider it to be normal behaviour that pointer difference will not work correctly.

- *C++* `operator new[]`: The operator `new` is used for C++ style of allocation. Its advantage over C style `malloc` is that `new` takes the size of array as an argument (as opposite to the total allocated size in `malloc`, which was often used as `malloc(sizeof(Type) * count)`). Also, `new` returns the correct pointer type, not `void*`. Another advantage is that failure to allocate memory results in `std::bad_alloc` exception instead of returning a `NULL` value. We consider this to be good practice, as programmers often forget to check return values of functions for failures. There is also a big difference in semantics. While `malloc` only allocates memory, operator `new[]` allocates memory and calls constructors for the objects.

  The operator `new`, however, internally uses `malloc`, and therefore is susceptible to the same pointer-overflow problem.

```
1   #include <stdio.h>
2
3   int main() {
4       const unsigned int Mi = 1000000;
5       unsigned int s = 2900 * Mi;
6       char* x_start = new char[s]; // allocate 2.9G of memory
7       char* x_end = x_start + s;
8       printf("ok, ptr is %x, end ptr is %x \n", x_start, x_end);
9       if (x_end - x_start < 0)
10        printf("BAD! we have array which doesn't fit into ptrdiff_t\n");
11  }
```

which will result in

```
>g++ new_alloc.cpp -m32 && ./a.out
ok, ptr is 4a752008, end ptr is f74f9d08
BAD! we have array which doesn't fit into ptrdiff_t
```

- *C++* `std::vector`: We have already shown that `malloc()` and `operator new[]` may return arrays longer than the maximal pointer difference. One may wonder what about basic STL structures like `vector`.

  STL's `vector` is a robust replacement for manual allocation of (variable-sized) arrays. We may therefore assume that it will be safe to use in different scenarios. But this assumption is again wrong. Of course, now we cannot blame vector only because it can allocate more than 2GB of memory. The point is, you are not using pointers

anymore. Instead of pointers, the `std::vector::iterator` class is used. However, `std::vector::iterator` is in this case implemented just by using plain pointers.

To prove that the `std::vector`'s iterators are susceptible to integer overflows (or equivalently that `std::iterator_traits<std::vector<> >::difference_type` is not enough to store difference of iterators), we found a simple example of a failing program. Our main result in this area is an example of a well-tested implementation which may fail. The example we uses STL's binary search algorithm `std::lower_bound`, which is implemented in the following style:

```
template<typename _ForwardIterator, typename _Tp>
_ForwardIterator lower_bound(
    _ForwardIterator __first, _ForwardIterator __last,
    const _Tp& __val)
{
  typedef typename iterator_traits<_ForwardIterator>::difference_type
      _DistanceType;

  _DistanceType __len = std::distance(__first, __last);

  while (__len > 0) {
    ...
  }
}
```

We exploited the `__len > 0` test and produced the following (counter)example:

```
#include <stdio.h>
#include <algorithm>
#include <vector>

const unsigned int Mi = 1000000;

int main() {
  std::vector<char> vect(2500 * Mi);
  printf("vector allocated, filling with data\n");
  for (unsigned int i = 0; i < vect.size(); i++)
    vect[i] = i / 10 / Mi;

  std::vector<char>::iterator it =
    std::lower_bound(vect.begin(), vect.end(), 10);

  printf("found value %d at position %u\n",
      (int) (*it), (unsigned int) (it - vect.begin()));
}
```

The output of the program compiled both for 32-bit and 64-bit architecture is shown below.

```
>g++ -m32 vector.cpp && ./a.out
vector allocated, filling with data
found value 0 at position 0

>g++ -m64 vector.cpp && ./a.out
vector allocated, filling with data
found value 10 at position 100000000
```

The bigger problem is that the same *error may show up even when the result fits into* `ptrdiff_t`. The problem arises from the computation itself – if we allocate 750 000 000

integers, the difference of two memory addresses will still be negative. But this difference is used in the intermediate computation – to obtain the difference of `int*` pointers, the raw difference of those two pointers must be obtained first and the result must be divided by the length of the `int` type. Therefore, it is not surprising that the result will overflow in the first stage and then it will be divided by 4. To summarize our results – the allocation of >2GB of memory in 32-bit applications may result in overflows at several places.

Moreover, the same problems affect filesystems – if you use 32-bit variables for seeking in the file you will fail to seek in files greater than 4GB of data. Nowadays such files are quite common, consider for example images of DVD discs.

What can be done

There are numerous examples very similar to these described here. One must be very careful when rewriting code from pseudocode to real implementation. If you do not consider integer overflows, it is almost certain that some disaster will happen. Probably the best example is the launching of the Arianne 5 rocket on June 4-th, 1996, which exploded 40 seconds after start. The bug was caused by integer overflow when converting from a 64-bit floating-point number to a 16-bit integer [LIO96].

To fix most of the errors, one should consider using "safe int" implementations. These integers are relatively slow, but they will signal any overflow/underflow in case it occurs. We recommend using Microsoft's `SafeInt` implementation available at [Mic].

But using `SafeInt` in each situation can be very tedious and slow. Therefore, there are other tricks we came up with.

First of all, there is the power of the C++ templating engine. C++ templates are basically processed by a preprocessor. This means that they are like macros and are extremely flexible. You may define templates of one class with totally different implementations, or, for that matter, with totally different sets of functions/members.

One nice feature of templates is that the compiler cannot automatically cast between types when there is an ambiguity.

For example template `template <typename T> T std::max(T a, T b)` used in code as `std::max(int, unsigned int)` will not compile, because the compiler cannot decide if to use `max(int, int)` or `max(unsigned int, unsigned int)` version. This is somewhat similar to function overloading, but more convenient.

So, the first trick we developed for safe coding is using a combination of templates and `std::numeric_limits`.

In this case the implementation is specially tailored for the type the caller used. Thus, the caller cannot make a mistake by wrong conversion. The biggest set of the problems solved with this way is "downcasting" to a smaller type. For example function `int max(int a, int b)` in plain C may be called with `long long int` arguments, but the function `template <typename T> T max(T a, T b)` will be called with correct arguments each time (and we also get an additional check for type mismatch between arguments).

In this way, we can solve all "downcasting" issues, except the ones that will be returned from the function. There isn't any good method to disallow return-value downcasting though.

The second trick, which we came up with while studying signed-vs-unsigned examples and problems with vectors, arrays and pointers, is that basically if you use unsigned values, you probably will not need the higher half of them. Like in the vector example, you probably never want to allocate more than 2GB of data. In that case, let us just assume that anything above this limit is considered "out of range" and we should issue the error. Of course, there are some valid reasons why to use full unsigned values (like using bitmasks / bit operations, ...), but in the general case, it is much easier to disallow potentially dangerous stuff and if people really need the missing one bit, they probably know exactly what the potential pitfalls

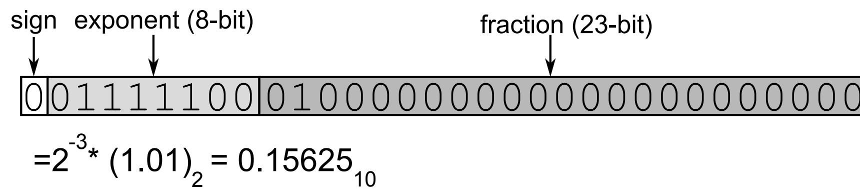


Figure 2.8: 32-bit floating point number representation

are and how to avoid them. Therefore, we propose to limit all unsigned integers only to the signed part.

2.5 Floating point problems

The current representation of floating-point numbers is based on the IEEE-754 standard [iee85]. At the beginning, we will introduce the standard itself, and later we will dig into the problems which may arise from this representation.

2.5.1 Floating-point representation

The IEEE-754 standard defines several types of floating point number representations. The various types differ by size. The standard requires availability of 32-bit (single-precision) numbers. The 64-bit (double precision), $\geq$ 43-bit (single-extended precision) and $\geq$ 79-bit (double-extended precision) are optional. All these types share a common structure, the difference is only in the number of bits used to store the exponent/mantissa and the bias of the exponent. A curious reader may consult the summary table specifying the parameters of these IEEE-754 types in figure 2.9.

Generally, an IEEE-754 floating-point number consist of three parts – the sign bit, the exponent and the mantissa. A graphical representation of a 32-bit (single precision) number is available in figure 2.8.

- the *sign bit* determines the resulting sign of the number and it is a single bit with 0 meaning positive numbers and 1 negative.
- the *exponent* contains the location of the radix point. The exponent is stored in excess-N form (biased representation). In 32-bit and 64-bit precisions the bias is set to $2^{n-1}-1$ where n is the number of bits of the exponent.
- the *fraction* is the part of the mantissa without its first bit. The first significant bit of the mantissa is not stored but rather it is determined from the exponent.

There are several possible combinations of exponent and fraction values:

- $exponent = 0$ and $fraction = 0$. The resulting number is ± 0 depending on the sign bit.
- $exponent = 0$, $fraction > 0$. Such numbers are called *denormalized numbers/subnormals*, because the missing bit of the mantissa is set to zero. These numbers allows to store small numbers around zero. The value of such numbers is $value = (-1)^{sign} * 2^{exponent-bias} * (0.fraction)$
- $0 < exponent < MaxExponent$. These are ordinary numbers. The hidden bit of the mantissa is set to 1 and their value is $value = (-1)^{sign} * 2^{exponent-bias} * (1.fraction)$

	single	single extended	double	double extended
fraction bits	24	≥ 32	63	≥ 64
biased exponent max	127	≥ 1023	1023	≥ 16383
biased exponent min	-126	≤ -1022	-1022	≤ 16382
exponent bias	127	unspecified	1023	unspecified
exponent bits	8	≥ 11	11	≥ 15
format width (bits)	32	≥ 43	64	≥ 79

Figure 2.9: Summary of IEEE-754 floating-point formats

- $exponent = MaxExponent$ and $fraction = 0$. These are *infinities*, i.e. their value is $value = (-1)^{sign} * \infty$
- $exponent = MaxExponent$ and $fraction \neq 0$. These are *NaNs* – Not a Numbers. They represent "errors" during computations.

2.5.2 Problems with IEEE754 floats

Equality test

The equality test is one of the most basic tests in programs. It turns out, however, that you cannot test floating-point numbers for equality. The problems arise from rounding errors.

The example code is [2.12] with output [2.13].

Code Listing 2.12: Rounding test

```
1  #include <stdio.h>
2
3  int main() {
4      double a = 1.0 / 3.0;
5      if (1.0 == 3.0 * a) {
6          printf("round OK\n");
7      } else {
8          printf("rounding problems!\n");
9      }
10 }
```

Code Listing 2.13: Rounding does not work

```
1  rounding problems!
```

In the code, the number obtained by division by three and multiplication back is not the same number as the original one. This may seem trivial to the reader, but there is something more about this problem. The original output was compiled with `g++`. Now, try to compile it with `g++ -O2` and the result [2.14] is quite different.

Code Listing 2.14: Rounding works with compiler optimizations

```
1  round OK
```

The problem of computer optimizations changing results will be discussed later.

According to [GS05] the results of equality tests can behave differently under different compiler optimizations. We will discuss this later when discussing the x87 floating-point unit, but to get a glimpse:

Code Listing 2.15: Equality test

```
#include <stdio.h>

int main() {
  double a = 3.0, b = 7.0, c;
  c = a / b;

  if (c != a / b) {
    printf ("something went wrong with comparison\n");
  }
}
```

Code Listing 2.16: Equality does not hold

```
something went wrong with comparison
```

We can see that testing for equality of two values can be problematic *even when we use exactly the same computations*.

Rules of associativity

As a starter, consider `double x = (a + b) - b`. In normal mathematic, this is equivalent to `x=a`. In floating point arithmetic this is not necessarily the case, for example if $a = 1$ and $b = 10^{100}$, the result of the full expression is 0 and the result of the optimized is 1. Thus the order of summation is very important. Further discussion about the effects of changing the order of summations can be found in [Hig93].

The compiler may introduce many problems while optimizing code. There is one important place when a change in summation order may occur. It is called "vectorization" and it is an optimization for the SSE unit. The SSE unit may perform the same operation on multiple arguments in one instruction. Thus the compiler may wish to exploit this behaviour and increase the speed of the program. Consider the following program which was borrowed from [Mon08]:

```
double s = 0.0;
for(int i=0; i<n; i++) {
  s = s + t[i];
}
```

If n is even, the program may be rewritten as

```
double sa[2], s; sa[0]=sa[1]= 0.0;
for(int i=0; i<n/2; i++) {
  sa[0] = sa[0] + t[i*2+0];
  sa[1] = sa[1] + t[i*2+1];
}
s = sa[0] + sa[1];
```

Now, the numbers are summed up in completely different order and thus the optimization may give completely different results, especially when we are summing numbers of different magnitudes or many positive and negative numbers.

x87 fpu extended precision problems

Current processors use x87 floating point unit with 80-bit registers (`long double` in C++). We will use several examples from [Mon08] to demonstrate various problems with optimisations.

```
double v = 1E308;
double x = (v * v) / v;
printf("%g %d\n", x, x==v);
```

Compiled with gcc under Linux, this code will print the value 10^{14}. Note however, that according to the IEEE-754 standard, the operation of $v * v$ should result in $+\infty$ and the division won't affect this. However, the calculations are done in extended 80-bit precision and they won't overflow there. To test this hypothesis, the authors of [Mon08] forced storing the variables by -ffloat-store compilation parameter and indeed the result was ∞. This same example can be pushed up further:

```
1 double foo(double v) {
2         double y = v * v;
3         return (y / v);
4 }
5 main() { printf("%g\n", foo(1E308));}
```

We explicitly requested the storage of the result of $v * v$. But still, the compiler may re-use the value of y already stored in the register to perform additional division. Again, the result of the computation is different for different compilation options.

The authors of the article moreover tried to experiment with another setting. When calling the functions, you **need** to store the variable and you can't reuse anything in the registers.

```
1  static inline double f(double x) {
2    return x/1E308;
3  }
4  double square(double x) {
5    double y = x*x;
6    return y;
7  }
8  int main(void) {
9    printf("%g\n", f(square(1E308)));
10 }
```

The calling convention is to return the floating-point values in x87 registers. But before calling the function `f`, the result needs to be pushed onto the stack and converted into a `double`. However, in case we inline the function `f`, there is no pushing on the stack and the value is passed directly in the x87 register. Thus, the act of inlining a function may change the results. Actually, the gcc compiler does not inline functions when invoked without optimizations, but it will inline the marked function with the -O2 switch.

The most annoying "feature" of x87 fpu optimizations is the fact that they are not debuggable. Consider for example that you will add a single `printf` call to show the variable before the problematic code. The compiler needs to save the value to memory due to the function call. Even more, the compiler is forced to re-load the value from the memory, because the register value may change during the `printf` call. After the call the compiler will use the double for next calculations and the results are different. Just adding the debug code will change the behaviour of the program and it will not fail now.

Double rounding

The authors of [Mon08] suggest that sometimes, rounding from type A to type B and then rounding type B to type C can yield different results than direct rounding from A to C. The reader may wish to read the mentioned article for more information.

2.5.3 Summary

Using floating-point numbers in algorithms is very tricky. For example, there are no simple equality tests. The authors of [Daw06] tried to find a correct solution, but their solution is quite complex and it exploits the memory representation of the floating values (which is not

very good for portability). Our recommendation is to avoid floating point numbers as much as possible. For example in banking applications floating-point numbers are never used and the values are stored as integers – the account balance in cents.

2.6 Function call stack limitations

One of the problems with recursive functions is that they take a lot of space on the stack. This may not seem as a problem because we have plenty of memory. But there are situations when you cannot use so much memory. The notable ones are

- *Linux stack size limit*: In Linux there is a default for maximum stack size that an application can have. The default is usually set to 8MB. This is far less than the size of the available memory.

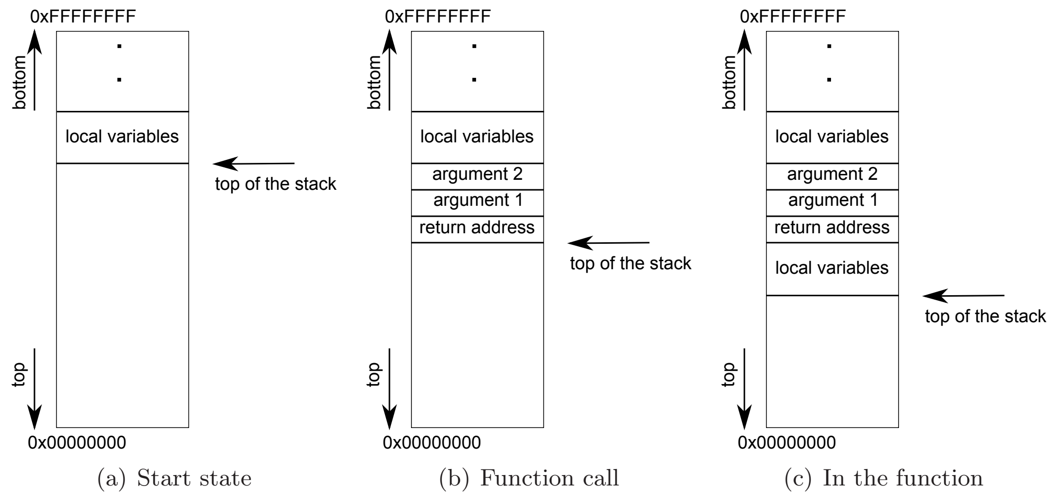
- *threading*: When programming parallel computations, programmers usually use threads. The advantage of threads is that they are much lighter than processes and thus are easier to switch. But using threading may impose its limits. The previously mentioned limit of the available stack size may be giant when compared with the stack limit for each thread. For example, the default stack limit for one thread on Linux is 2MB. Good practice in large projects is to limit the stack even more. In such a scenario you cannot use recursion much and the only way around is to simulate recursion on the heap instead of the stack. That might be quite a complex task depending on the type of the problem.

To introduce the problem, let us start with the explanation of what a function call stack looks like. There are different calling conventions and each of these conventions has a different scheme of the stack. In this thesis we will discuss the `x86 cdecl` calling convention, because it is the most common calling convention used by C compilers.

The `x86 cdecl` convention for calling a function is this: First the caller pushes the function arguments onto the stack. The arguments are pushed onto the stack from right to left. This enables support for functions with a variable number of arguments. After the parameters have been pushed onto the stack, the 4 or 8 byte pointer (depending on the type of the architecture) to the return address is pushed onto the stack. Now the function is ready to be called and the `call` instruction is used. The function itself stores any local variables on top of the stack. Returning from the function is done via the `ret` instruction, and after that the caller is responsible for cleaning the stack (removing the return address and arguments). The whole process is shown in figure 2.10.

A quick calculation tells us that if we have two (4 bytes long) integers to store as function parameters plus one local integer variable, we will need (on a 32-bit architecture) 16 bytes of stack. These are however really the minimal requirements. In real-world code there are usually more variables. Moreover there are complications like C++ exception handling, destruction of classes, etc. Also note that the compiler may increase the stack frame size for better speed (aligning data structures at the word boundary usually helps). For example, the `gcc` compiler needs to have stack frames aligned to 16 bytes. Therefore the function may occupy more space on the stack than is strictly needed.

To determine whether the real-world scenario of running out of the stack is possible, we have written some very simple programs:



(a) Start state (b) Function call (c) In the function

Figure 2.10: Function call stack visualization

```
#include <stdio.h>

void recursive(int value) {
  printf("%d\n", value);
  recursive(value + 1);
}

int main() {
  recursive(0);
}
```

The example above consists of a simple function call which takes one integer argument. We may wonder how many recursive calls can a program achieve:

```
>ulimit -s
8192

>g++ stack1.cpp && ./a.out

261923
261924
Segmentation fault


>g++ stack1.cpp -O2 && ./a.out
infinite execution
```

The second line of the output shows that we have 8MB of stack available for our program. When compiled without any optimizations, the function can be nested approximately 260 000 times. This corresponds to 32 bytes for each invocation. The same example compiled with advanced optimizations turns out to run indefinitely – the tail recursion was optimised to a simple loop.

Let us consider a more complicated scenario, we selected depth-first search as a good candidate.

```
1  #include <stdio.h>
2  #include <vector>
3
4  std::vector<int> edges[3];
5  bool visited[3];
6  int count;
7
8  int recursive(int vertex, int parent) {
9    // for correct DFS uncomment following line
10   // visited[vertex]=true
11   if (visited[vertex])
12     return 0;
13
14   printf("%d\n", count++);
15
16   int child_count = 0;
17   for (std::vector<int>::iterator it = edges[vertex].begin();
18          it != edges[vertex].end(); ++it) {
19     if (*it == parent) continue; // ignore reversed edge
20     recursive(*it, vertex);
21     child_count++;
22   }
23   return child_count;
24 }
25
26 int main() {
27   edges[0].push_back(1);
28   edges[1].push_back(2);
29   edges[2].push_back(0);
30   recursive(0, -1);
31 }
```

The corresponding outputs are shown below:

```
>g++ stack2.cpp -m64 && ./a.out
130917
130918
Segmentation fault

>g++ stack2.cpp -m64 -O2 && ./a.out
174557
174558
Segmentation fault


>g++ stack2.cpp -m32 && ./a.out
174574
174575
Segmentation fault

>g++ stack2.cpp -m32 -O2 && ./a.out
130902
130903
Segmentation fault
```

In the case of a simple DFS one invocation of the recursive function takes 48 or 64 bytes of the stack. The funny thing is that optimizations for 32-bit architecture and 64-bit architecture have opposite effects.

Based on these observations we may say that even when the function is using very limited local storage, it cannot do much more than 100 000 recursive calls (on Linux with the default settings). Moreover, this will get a lot worse if we start using classes/bigger structures as local variables to that function. And the compiler optimizations may also produce a waste of some space because of stack frames alignment or other optimizations for speed.

The 100 000 recursive calls limit may be a problem for some algorithms. We showed that DFS may get into trouble, especially on large graphs which are almost paths. But DFS is not the only one. Therefore we may conclude that some algorithms need to be prepared for such situations and use their own stack (allocated on the heap, not on the stack area).

2.7 Hardware errors

A special kind of program errors are hardware errors. This thesis does not deal with the problematics of hardware errors, but nevertheless we thought it would be for the best to include the basic examples.

Even when the software is completely correct, there are times when hardware fails in some way. Such errors may be hard to spot. But there are applications for which such small failures may be critical and there is a need for some type of a detection/recovery. Even when the application is not critical but the task size is extremely large, these kinds of errors can cause big problems because the probability of their occurrence on large-scale computations is quite high. There are several examples of hardware errors from the past which we want to mention.

In 1994 Intel processors contained a bug in the hardware table used for floating-point division [Int04]. The result was that around 1 out of $9 * 10^9$ random floating point divisions resulted in an answer that was wrong starting at the 12-th binary place (4-th decimal place). This particular bug was found by Thomas Nicely during number-theoretic computations.

Another type of hardware errors is corruption of data. Memory databanks hold small electric charge. This charge may be changed if a high-energy particle strikes the chip. Thus computers working in an environment with high radiation levels (or the cosmos) have a much higher chance of bit-flipping. But cosmic radiation is quite intensive even on ground level and it cannot be shielded (unless you want to hide your computer below several meters of lead).

Back around year 2000, the memory chips did not have ECC (error correcting codes) and during processing of big amount of data the errors were showing up. Jeff Dean from Google company in his presentation [Dea11] said that "Sort 1 TB of data without parity: ends up 'mostly sorted'. Sort it again: 'mostly sorted' another way". Of course much has changed since then and ECC memories are available now, but the lesson learned still holds. Huge amounts of data need to be protected by error correction codes. If you want to know more about memory errors, you may wish to read a nice whitepaper [mem].

But the memory is not the only possible medium in which errors may occur. Consider hard drives. Dust or other tiny particles may settle on the disk's plate and create problems. Hard drives, however, contain checksums and error-correcting codes. Thus the chance of a real non-recoverable error is really small. But while the average capacity was growing steadily over last few years, the non-recoverable error rate increased more sluggishly.

In the following lemma we estimated the probability of encountering a read error while reading the whole disk:

Lemma 2.7.1 *Let $p \ll 1$ be a probability of a read error resulting in a non-recoverable bit. Let n be the number of bits of the hard disk. If we assume independence of read errors, the probability of reading the whole disk without an error is approximately e^{-np}.*

Proof: If we assume that read errors are independent, we can model the situation using Poisson distribution

$$f(k, \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

where $f(k, \lambda)$ is expected probability of k events when the expected number of events is λ. In our case we are asking for the probability of no errors($k = 0$) and $\lambda = pn$. Thus, the final probability is

$$f(0, np) = \frac{(np)^0 e^{-np}}{0!} = e^{-np}$$

□

In the case of current hard drives, the error rate for 250GB disks is said to be less than $p = 10^{-14}$. Suppose that you own a RAID5 array with four 250GB disks. In case of a failure of one drive, you need to read the 750GB in order to reconstruct the failed drive. This leaves $np = 10^{-14} * 8 * 750 * 10^9 = 0.06$. Thus the probability of a non-recoverable read error during the reconstruction is $e^{-0.06} = 0.94$. In 6% of cases you won't be able to recover. These figures may be more striking for even bigger disk sizes.

Chapter 3

Infrastructure

3.1 Programming language selection

The programming language which we used for this thesis is C++. In this section we will briefly discuss this decision and possible alternatives.

The programming language C++ is an object-oriented language which evolved from the programming language C and thus it is a procedural programming language.

C++ is currently one of the most widespread programming languages in the world. There are numerous compilers for many different architectures available. Also, the compilers use very tight optimizations and the C++ performance is still outstanding compared to other languages.[1]

One major advantage of C++ over other new languages is the compatibility with C. C++ can be linked with C libraries without any problem, and there are millions of C libraries out there. This is the main reason why it is still widely used.

C++ is a complicated programming language. Though it is quite old, it posesses very useful concepts like macro preprocessing and especially templating, which is more powerful than in any other language we know. Also, there are numerous syntactic rules and many ways to write the same code. Some of these cases are more complicated, confusing or "magic". Therefore in this thesis we will try to use only the basic features of the language and we won't try really advanced constructs such as multiple inheritance and similar. In this way, the code written in C++ will also be somewhat portable to Java, C# or PHP (although some things will be done completely differently because of different standard libraries and different data structures available). So, writing code in C++ is an ideal choice for this thesis.

The basic building block of C++ programs are algorithms and data structures from the Standard Template Library (from now only STL), which is a highly templated and general set of data structures, helpers and various simple algorithms. As an example set of STL content, there are data structures like `vector` (i.e. variable length array), `stack`, `queue`, `priority_queue`, `map`/`hash_map`, `set`. From algorithms there is `sort`, binary search (`lower_bound`, `upper_bound`), `next_permutation`, `set_union`, `copy`, `for_each` and many more. From this point of view, STL is a very good opponent for any standard library of modern languages.

In this thesis, we *won't reimplement STL* – we will do exactly the contrary – we will try to use STL and its concepts to make our implementations more adjustable and portable.

[1] Although Java with JIT compilation has comparable running times.

3.2 Testing infrastructure

3.2.1 Testing framework

Testing is an important part of programming practice. There are many ways how to test a piece of software. For this thesis we needed an automated, easy to run and easy to use testing framework. We therefore invested some time to learn about different variants of testing frameworks available. The summary of our selection is written in this section.

At first, we would like to introduce the reader to the problematics of testing. Software testing can be categorized into two broad categories - manual and automatic testing. However, there is no strict boundary between these two - there are tools than can help automating manual testing, or sometimes automatic tests need manual intervention. For testing of algorithms, the fully automated way is preferred. As for the tests themselves, we may view them as small pieces of code which can check some specified functionality of a program/class/function. The smaller and more isolated tests are, the better the chance of finding and isolating the defects in the code. Therefore, the ideal testing framework allows us to write many small tests for classes and functions. The testing framework should also handle as much repeating constructs as possible. This helps the programmer/tester to concentrate more on the test itself. For example, instead of writing

```
1  int main() {
2    int result = my_function_call(42);
3    if (result != 47) {
4      printf("The test failed - expected value 47 but %d returned", result);
5    }
6
7    try {
8      my_function_call(-1);
9    } catch (std::exception e) {
10     printf("The test failed - there was an unexpected exception!\n");
11   }
12 }
```

it is much more natural and much more convenient to write

```
1  TEST(MyFunctionTest) {
2    EXPECT_EQUALS(47, my_function_call(42));
3
4    EXPECT_NO_THROW(my_function_call(-1));
5  }
```

Googletest

There are plenty of unit testing frameworks achieving those goals available for the C++ language. To name a few - "C++test", "cppUnit", "QtTest", "UnitTest++" and many more. Our selection was Google C++ testing framework called "googletest". We selected this framework mainly because of its ease of use. The framework allows to easily to set-up small unittests that are discovered automatically by the framework. This is a very useful feature as requiring manual registration of tests would probably introduce the chance of forgetting to register newly written tests.

Googletest can easily create test functions or even test classes. There is support for more advanced features as well. Moreover there are a lot of custom assertions available (`EXPECT_EQ`, `EXPECT_LESS`, `EXPECT_THROWS`, ...). For advanced testing, there is Google Mocking Framework, which enables users to test interactions between classes. Overall, the Googletest framework is easy to use and fast to set-up and we decided that it would be the best choice.

For a testing framework, it is desired that tests can be run easily. In case of googletest, this means that you need to compile the testfile and run the binary (which can be automated by a Makefile). Also, the framework should provide nice and easy-to-understand output. To demonstrate the capabilities of googletest, we included the test output of one selected test run:

```
Running main() from gtest_main.cc
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from ModInvTest, where TypeParam =
[ RUN      ] ModInvTest.invalidInit
[       OK ] ModInvTest.invalidInit (1 ms)
[ RUN      ] ModInvTest.small
modular_inverse_precomputed_unittest.cpp:24: Failure
Value of: m.getInverse(4u)
  Actual: 4
  Expected: 4747u
  Which is: 4747
[  FAILED  ] ModInvTest.small, where TypeParam =  and GetParam() =  (0 ms)
[ RUN      ] ModInvTest.consistency
[       OK ] ModInvTest.consistency (2155 ms)
[----------] 3 tests from ModInvTest (2156 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran. (2156 ms total)
[  PASSED  ] 2 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] ModInvTest.small, where TypeParam =  and GetParam() =

 1 FAILED TEST
```

Moreover, the output of the test run itself is colorized (green for passed tests and red for failed) and thus it is even more readable than our example. Therefore, it is really easy to recognize the failed conditions and figure out the failed test scenarios. The curious reader may look in appendix at listing 4 which shows one of the many unittests from this thesis.

3.2.2 Managing compilation of large codebase

In large projects there are many source code files distributed around several directories and the user needs to (re)compile several of them. The main problem is dependencies – to compile one specific target, other targets need to be built first. Also, the compiler is usually supplied with complex commandline of various options. In the linux environment, there is a program called "make" which tries to solve this problem. The programmer writes a "Makefile" which is basically a script describing how to compile each target and what are the dependencies between targets. However, as the project grows, the Makefiles are harder to maintain. Therefore, there is a need for building infrastruture that is more advanced than Makefile. There are several such "build frameworks" available and they can take over the whole task of maintaining the dependencies and compilation command lines. The basic choices for C++ language are for example GNU autotools or cmake. We believe that for a medium project as this thesis is, these tools are too heavy-weight. Therefore our choice was to write a very simple Makefile-creating platform in the Python language. This decision turned out to be very useful, as now we can automatically create Makefiles with "test" and

"benchmark" rules and so on, automatically compiling and running the whole set of tests or benchmarks without any additional complications. The source code of this Makefile generator and the generated Makefiles can be found in the electronic attachment.

3.2.3 Benchmarking

Besides a testing framework, we also needed some way of benchmarking code. In practice it was easier to provide our own solution with one simple macro over finding some complicated way how to benchmark the results. Now we will discuss this solution and the problems with it.

There are several problems related to benchmarking. For example, there is an issue about what to measure. There are several available metrics and neither of them is superior. The short list may begin with

- *CPU instruction count* Measuring instruction count can provide the best estimate of the performance, but it may miss some other important aspects that are determining overall performance. For example, the instruction count cannot take into account memory cache misses and I/O operations. Another problem is the benchmark overhead which in this case isn't negligible.

- *Total elapsed time* Provides the best possible estimate on user-experienced performance. The drawback is that the measurements may be noisy as the operating system may be allocating resources to other programs, which may disturb the benchmarking.

- *Processor time* Provides the best possible estimate of the running time of the benchmark not depending on other processes. On the other hand, the total running time of the benchmark may depend not only on the time the process spent using the CPU. If the process does a lot of I/O operations, the processor time will be only a fraction of the real time and there will be a lot of waiting for the hard drive.

In this thesis, we decided to measure the total running time. The main reason behind this is that it is easily available on all platforms. On the other hand it is hard to get information about CPU instructions or processor time, especially if we want the program to be portable to many platforms. To mitigate the problem with the noisy measurements, we conducted an experiment on the noisiness of the results and set the "minimal benchmark time" to be 1.5 seconds. After this time, the fluctuations between several runs of the same benchmark were relatively small for our purposes. Note that we are not aiming to optimize programs to the last instruction, the purpose of our benchmarks is to compare algorithms with usually different asymptotic time.

Another complication with the benchmarks is that their speed on different machines is quite different. Therefore, it is impossible to "time" benchmarks by specifying the number of iterations to be performed. To mitigate this problem, we created the `AUTO_BENCHMARK` macro, which runs the benchmark for at least the specified minimum running time. When measuring fast functions/constructs it is important to have as little benchmarking overhead as possible. The one notably slow operation is getting the system time. Therefore we decided to have a benchmarking macro with "exponential growth" of the iterations count between time-tests. This solution is very flexible and produces benchmark time that runs at least the specified minimum time and at most three times as long. Moreover, the benchmarking overhead in the inner loop is practically zero. The implementation of our benchmarking macro can be seen in appendix 1.

The last complication with benchmarking is the code optimization done by a compiler. Sometimes it is indeed hard to write a benchmark which will run the whole code. Especially if a major part of the tested functionality is provided in the header files (and this is true for

our thesis, mainly because we are using a lot of templates, which must be placed in header files), the compiler may use aggressive optimizations. One such optimization is that if a function does not have side effects and the return value is not used, it is virtually "useless" and the call can be skipped. Of course, this may happen in the benchmark itself – usually, you are not using the return values of the functions you are benchmarking. Or, if you call the function with constant arguments and the function does not have side effects, the compiler may cache the result and skip the next function call. The compiler may even decide to optimize out unnecessary loops and so on. Therefore, when benchmarking, it is important to check whether the benchmark is still valid, or if the compiler did a very good job optimizing the code and removed a significant part of it.

The other possibility is to benchmark the unoptimized code, but this is not necessarily the best solution. For example, the STL `std::vector` data structure is several times slower than ordinary C arrays without optimizations, but it is almost identically fast when used with `-O2` optimizations. Thus, the characteristics of the unoptimized and optimized code may vary dramatically between various algorithms and the benchmark will not give reliable metrics.

3.3 Preconditions – asserts for inputs

As we discussed earlier, a function should check its input values. This is especially needed when bad parameters may put the program into an undesirable state like rewriting a part of memory which was not supposed to be manipulated. Even functions without bad side-effects should check their arguments, because the faster the error is discovered, the easier it is to trace and fix the bug.

Thus each function/class in our implementation checks all its arguments for any values that are out of normal. Functions should also check for values that would produce overflows or other errors. The number of such checks is therefore quite big and therefore we needed a way to easily check the code and not introduce too much bloat to it. We considered several possibilities for the implementation of preconditions.

- manual if-then check and returning of the "error" value:

```
1 const double E_BAD_SQRT=-1;
2 double sqrt(double x) {
3   if (x < 0) {
4     return E_BAD_SQRT;
5   }
6 }
```

- manual if-then check and throwing an exception:

```
1 double sqrt(double x) {
2   if (x < 0) {
3     throw std::invalid_argument("Square root of negative number");
4   }
5 }
```

- use existing `assert` macro:

```
1 double sqrt(double x) {
2   assert(x >= 0);
3 }
```

Neither of these possibilities was good enough for us. The first option is bad because programmers never bother to check return values of functions. It would be better to throw an error or to halt the execution as the assert macro does.

Manual check is a quite good solution but there is the problem with its verbosity. You need three lines of code to check a single condition. You may try to fit everything onto one line, but it is still quite verbose and the line will be long.

The disadvantage of an `assert` macro is that it will instantly halt the program and there is no way how to recover from this. This makes it impossible to catch the error in the program itself. But the program may want to recover. For example in a simple application such as a calculator, entering square root of -1 should report an error to the user and allow for correction and not halt the whole program. The same reason applies for testing – you cannot test whether the code fails in cases when it should fail, because an assert will just kill the program.

We therefore propose a solution called a `Preconditions` class. `Preconditions` is a class with many useful functions for checking the input values. The basic usage is

```
1 double sqrt(double x) {
2   Preconditions::check(x >= 0, "Square root of negative number");
3   }
4 }
```

The `Preconditions` class itself checks the expression for validity and reports an error by throwing `std::invalid_argument` exception in case the expression does not hold. This way, the coding overhead is as little as possible. Moreover, the statement itself suggests that it is checking the pre-conditions of the algorithm and thus is distinguished from the ordinary checks. The programmer may even use the `Preconditions` as a documentation of the function inputs.

The best part is that the most used checks can be implemented in the `Preconditions` class to simplify the checking even more. For example instead of checking the range $[0, right)$ by `Preconditions::check((x >= 0) && (x < right))` or using two separate checks, we implemented a simple function `Preconditions::checkRange(x, 0, right)`.

We hope that this checking of preconditions helps the programmer to write safe and simple code and that other projects will adapt it too. To see the implementation, consult appendix 2

3.4 Design choices

There were several nontrivial design choices made during the implementation of the algorithms from this thesis. We will discuss the most important ones now.

- *Passing output parameters*: There are several ways how to pass output parameters of functions. The basic solution is to return the output in the result. This is of course the preferred way. But sometimes this cannot be done. For example, if the resulting structure is too big to be copied, returning the whole object may cause memory and performance problems. Therefore, there is a need for other way of returning objects. The two most simple solutions of passing a modifiable argument to the function are sending a reference of the parameter or sending the pointer to the parameter itself. The two methods may be used in following way:

```
1 int doMagic1(const vector<int>& input, vector<int>& output);
2 int doMagic2(const vector<int>& input, vector<int>* output);
3
4 vector<int> input;
5 vector<int> output;
6 doMagic1(input, output);
7 doMagic2(input, &output);
```

The benefit of the first method over the second is that the compiler verifies that the reference is a pointer to the actual variable. Thus the programmer cannot accidentally pass a wrong reference. The second method is not checking the correctness of the pointer and if a programmer calls the function with bad parameters like `doMagic2(input, NULL)`, the function will fail.

The benefit of the second alternative is that the programmer will see the difference between the normal variables and possible output variables. In other words, seeing that the function needs a pointer means that the function can manipulate with the object (unless the pointer is `const`). On the other hand, if the programmer sees `doMagic1(variable1, variable2)`, he/she cannot guess if the two passed vectors will be changed or not. Thus he/she must check the signature of the function. This may be very dangerous, especially if it is not very clear if the variable is used only as input, only as output or both as input and output.

From these two approaches, we therefore prefer the second one as it explicitly differentiates between input and output variables.

- *Alternative/replaceable subalgorithms*: A much harder design problem is how to implement algorithms with alternative sub-algorithms. For example the Miller-Rabin algorithm for primality testing uses an algorithm for fast modular exponentiation. We can go even deeper. Fast modular exponentiation uses an algorithm for fast modular multiplication as a subroutine.

  The design problem is how to make these sub-algorithms replaceable. The standard solution is the design pattern called Command or Strategy, which aims to provide exactly this kind of functionality. However, the problem is that these patterns usually work with a hierarchy of classes. This enables the possibility of highly-customizable sub-algorithms. The cost for these design patterns is the code overhead. If we use the mentioned patterns, we need to instantiate a lot of objects. In our example of primality testing algorithm, the class instance of fast modular exponentiation must be created before and then we need to pass this class instance to the algorithm itself. And again, for the exponentiation we need to instantiate a multiplication first and then pass it to the algorithm.

  Thus we searched for another type of solution and we were inspired by STL library. Basically we use the mentioned design patterns, but not in the classical class representation. Instead, we use C++ templates to do the "instantiation" and there is no need for real instantiation. The disadvantage of this method is that the algorithm cannot be replaced on the fly during program run – the sub-algorithms are inserted into the code in compiling phase. The second disadvantage is that templates cannot provide such level of type checking as class hierarchy. Indeed, you may pass a totally different algorithm as the parameter of the template and the program will usually fail to compile with very user-unfriendly error messages. But there are some standard ways how to mitigate this and it will be much easier to do in the upcoming C++0x standard. The basic usage is shown in the next figure.

```
// algorithm with a sub-algorithm
template <class PowerModImpl>
class PrimesFast_ {
 // implementation, can use PowerModImpl::functions
}

// "default" version of algorithm
typedef PrimesFast_<math::powermod::PowermodExtended> PrimesFast;

// usage
PrimesFast::isPrime(5);
PrimesFast_<MyPowermodImpl>::isPrime(7);
```

General guidelines

The general guidelines for our codebase are

- *Use only limited features of C++*: The C++ language is very expressive and there are many ways of writing code in it. We try to use the smallest and well known core of the language and try to avoid using bizarre extensions.
- *Use consistent code style*: The style of the code is important. It is much easier to read code with consistent naming conventions. The indenting conventions are important as well. Our coding style guide is based on the [Goo] with the exception of the naming convention which is Java-style.
- *Document anything nontrivial*: Documentation is important. If something is not trivial at the first glance, there should be explanation about what it does. Usually, there are many hidden presumptions and it is not easy to understand them. Even more, documenting nontrivial pieces of code helps the reader to understand the problems.
- *Test for invalid inputs*: As we have discussed earlier, if the input is invalid, the function should report a problem with the arguments.
- *Test corner cases*: Algorithms should be tested for corner cases – empty inputs, too small inputs, too big inputs, etc.
- *Test standard inputs*: The implementation should be tested on a wide range of different inputs. The best way is to cross-check with another implementation on some large random sample of inputs.

Chapter 4

Our implementation

4.1 Math algorithms

4.1.1 Modular arithmetics

Many discrete-math algorithms use calculations "modulo m". Formally, they compute operations on numbers a, b belonging to some ring X. The standard operations include:

- addition/subtraction
- multiplication
- exponentiation
- greatest common divisor (not the modular operation, but it is useful for many algorithms working with modular arithmetics)
- modular inverse

In this section, we will briefly discuss problems with modular arithmetic and our implementations.

Modular addition and subtraction

The standard modular addition code in C++ is

```
result = (a + b) % modulo
```

The problem we have already discussed several times is that the computation may overflow. However, if a and b were from the range $[0, m)$, overflow may occur only if m is bigger than half of the type's maximum value. Thus, overflow may be checked very easily. There is a bigger issue with subtraction. The Standard code

```
result = (a - b) % modulo
```

does not work in this case. The problem is, that if $a < b$, the result of the computation is negative. If the basic type of variables a and b is unsigned, overflow occurs. If the basic type is signed, there is still an error in the result – the remainder of the division will be negative. Thus, the code needs to be fixed in one of the following ways:

```
result = ((a - b) % modulo + modulo) % modulo
result = (a - b) % modulo; if (result < 0) result = result + modulo;
```

If we assume, that a, b are normalized to range $[0, modulo)$ before this operation (probably because they are results of previous modular operations), we can use following simplification

```
result = (modulo + a - b) % modulo
```

Modular multiplication

The integer overflow error is more evident in the context of modular multiplications. For the computations to be correct, the intermediate result before division should be enough to hold the whole multiplication. In practice, this means that the size of the intermediate result should be twice of the size of variables a and b. If the underlying variables are 32-bit integers, this may be done easily by using 64-bit computations. But beware of the following buggy code:

```
int32_t a,b;
// This is a bug, as the calculation is done in 32 bits
// and then converted to 64 bits!
int64_t result = a * b;
```

Because of lack of 128-bit integers, there is also a problem with 64-bit numbers modular arithmetic. In the thesis, we implemented standard modular multiplication and also implementation which can compute up to 63-bit modular arithmetic. The basis of the implementation lies in emulating modular multiplication by a series of shifts and additions (like in actual multiplication), but performing the modulo operation at each intermediate stage. If we limit the size of the numbers to $64-t$ bits, we may use remaining t free bits in intermediate calculations and thus we may multiply by t-bits sized blocks. The running time is $\Theta(\frac{64-t}{t})$ and the algorithm may be fine-tuned depending on the number of free bits. Also, we provide another implementation, which can compute modulo of 63-bit numbers (i.e. $t = 1$), but it is highly optimized. The major optimization is the conversion of the slow remainder calculation with a fast conditional `if` statement and subtraction.

Our work consists of the following implementations:

- Naive 32-bit modular multiplication using 64-bit variables
  - + requires two 64-bit operations
  - − cannot handle 64-bit arguments
  - ◦ see the class `MultmodSimple` implemented in *math/powermod/multmod_simple.h*
- $(64-t)$-bit modular multiplication
  - − requires $2\left\lceil\frac{64-t}{t}\right\rceil$ multiplies/divisions
  - + can handle $(64-t)$-bit arguments
  - ◦ see the class `MultmodExtended<t>` implemented in *math/powermod/multmod_extended.h*
- 63-bit optimized version
  - + requires no multiplication/division and at most 3*64 additions/subtractions.
  - + can handle 63-bit arguments
  - ◦ see the class `MultmodOpt` implemented in *math/powermod/multmod_extended.h*

Fast exponentiation

There are occasions when you need to compute the b-th power of element a belonging to ring X, i.e. we need to compute $a^b \in X = a * a * \cdots * a$. There are two rings in which this computation is very common: Z_m – the ring of all integers modulo m and $M_n(R)$ – the ring of all $n \times n$ matrices over an arbitrary ring R.

The basic solution is to repeatedly multiply b times, which takes linear time (in b). This is very inefficient especially for very large values of b. An example of an algorithm which

needs fast exponentiation can be Pollard's rho factorization method, RSA encryption/decryption, finding modular inverse over prime field, fixed point of matrix multiplication (if the multiplication is convergent).

But before we can continue the discussion of faster algorithms, we must warn the reader: *Do not implement algorithms from this section when you are working with cryptography and/or security.* Many of the algorithms have a serious flaw called "timing attack". The attacker can deduce some secret variables used during the computations by measuring the execution time of the algorithm on different inputs. For more information about timing attacks, you can consult [Koc96].

There are basically two major implementations – one which computes the result going from the most significant bits of b, the other from the least significant bits. These algorithms may be further improved by minimizing the number of required multiplications. We have implemented the second of the algorithms.

- computation from least-significant bits.
  - + simple, running time $O(\log b)$
  - ◦ uses modular multiplication algorithm
  - ◦ see the class `Powermod_<MultmodImplementation>` implemented in *math/powermod/powermod.h*

There are the following common problems with implementations of modular exponentiation:

- integer overflows (the same problems as for modular multiplication)
- $x^0 \pmod{m} = 1$, but there exists the special case $x^0 \pmod{1} = 0$. We can demonstrate this on a simple implementation:

```
1 powermod(x, n, mod) {
2   result = 1;
3   for (i = 0; i < n; i++) result = (result * x) % mod
4   return result
5 }
```

  The correct fix is either checking for `mod == 1` or using `result = 1 % mod` . This special case was found by my supervisor and I am thankful to him for that.

Greatest common divisor

Greatest common divisor is the standard textbook algorithm. In fact, the textbooks usually use the recursive version, but there is also an optimized version using only a loop.

Problems of common implementations are:

- usually implementations do not check for negative values of arguments. The output in that case might be negative, which is incorrect
- signed range – even if authors checks for negative values and switch the sign of arguments or function result, overflow might occur during this sign change (see section 2.4.2).

This thesis contains several variations of gcd algorithms

- using STL's `__gcd()`
  - – `__gcd()` does not check sign of the operands
  - + we check for all overflows and sign problems
  - + time $O(\log n)$
  - + easy to implement
  - – see the function `gcd` implemented in *math/gcd/gcd.h*
- extended Euclidean algorithm using recursion
  - ○ classical textbook implementation
  - + computes the values a, b, such that $ax + by == gcd(x, y)$.
  - + we check for overflows and sign problems
  - + time $O(\log n)$
  - – see the class `ExtendedGCD` implemented in *math/gcd/extended_gcd.h*
- extended Euclidean algorithm using loops
  - ○ optimized version which avoids recursion
  - + computes the values a, b, such that $ax + by == gcd(x, y)$.
  - + we check for overflows and sign problems
  - + time $O(\log n)$
  - – see the class `ExtendedGCDLoop` implemented in *math/gcd/extended_gcd_loop.h*

4.1.2 Modular inverse

There are several algorithms for computing modular inverses. For the implementation we picked up the following: precomputation of the whole table in linear time, using Fermat's little theorem $a^{p-2} \equiv a^{-1} \pmod{p}$ and using extended gcd algorithm. The summary of their properties is in the following list:

- Precomputation of whole table
  - ○ time $O(n)$, memory $O(n)$
  - + easy to implement
  - + fast if we need to store the whole table anyway
  - – works only for primes
  - ○ see the class `ModularInversePrecomputed` implemented in *math/modular_inverse/modular_inverse_precomputed.h*
- Using Fermat's little theorem
  - ○ time $O(\log n)$, memory $O(1)$
  - – more complex (especially for 64-bit numbers), needs fast exponentiation
  - – works only for primes
  - – a bit slower than others
  - ○ see the class `ModularInverseFermat` implemented in *math/modular_inverse/modular_inverse_fermat.h*

- Using extended Euclidean algorithm
  - ∘ time $O(\log n)$, memory $O(1)$
  - \+ simple
  - \+ can work also for non-prime modulus as long as the inverse exists
  - \+ quite fast in practice
  - ∘ see the class `ModularInverseGcd` implemented in *math/modular_inverse/modular_inverse_gcd.h*

	Precomputation	Fermat	GCD
$p = 48611$	19ms	134ms	46ms
$p = 195413$	93ms	626ms	203ms
$p = 888061$	460ms	3140ms	1020ms
$p = 15485863$	9.5s	61.8s	21.5s
$p = 982451653$ range $10^6 \ldots 2 * 10^6$	-	4.9s	1.3s

Figure 4.1: Time to compute inverses of all numbers modulo p

Based on the benchmarks shown in figure 4.1, we recommend to use Extended Euclidean algorithm for modular inverse implementation as it is simple to implement and mainly because it can work also with modulus that is not a prime number. If you need to compute the whole table and it should be computed very fast, you may consider also using linear-time precomputation algorithm, which is twice as fast and relatively simple to implement too.

4.1.3 Primality testing

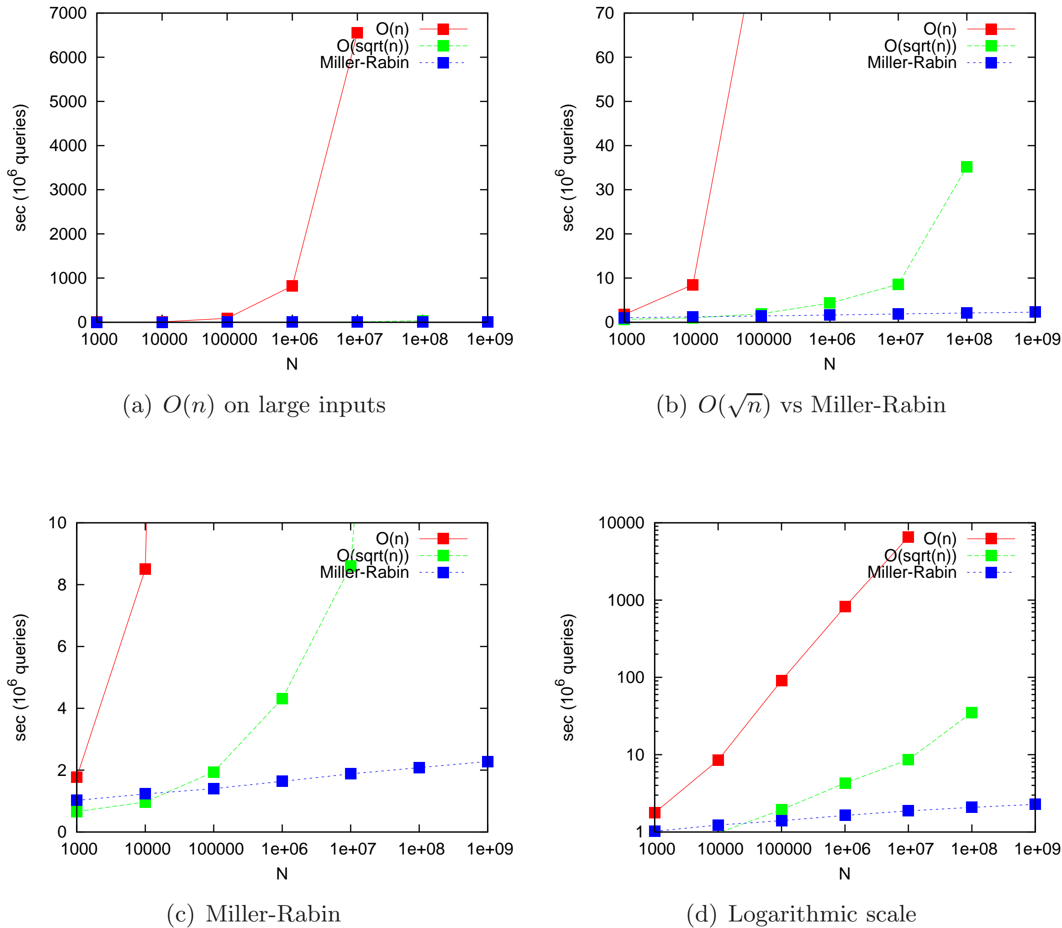
In this section we will discuss algorithms for primality testing and enumeration of primes. We start with the basics which everyone should know:

Definition 4.1.1 *A positive integer p is prime iff $x|p$ implies $x = 1$ or $x = p$. Specially, 1 is not a prime.*

Lemma 4.1.1 *An integer $p \geq 2$ is prime iff p is not divisible by any number from $\{2, 3, 4, \ldots, \sqrt{p}\}$*

In this thesis, we implemented several primality-testing algorithms.

- standard algorithm based on definition 4.1.1 of primality
  - − time: $O(N)$
  - \+ easy to implement
  - ∘ see the class `PrimesSlow` implemented in *math/primes/primes_slow.h*
- optimization of the previous algorithm that tests up to $\sqrt{n}$ based on lemma 4.1.1
  - − time: $O(\sqrt{n})$
  - \+ easy to implement
  - ∘ see the class `PrimesBasic` implemented in *math/primes/primes_basic.h*
- deterministic variant of the Miller-Rabin primality test
  - ∘ based on [Jae93], [PSSSW80], [ZT03]



(a) $O(n)$ on large inputs

(b) $O(\sqrt{n})$ vs Miller-Rabin

(c) Miller-Rabin

(d) Logarithmic scale

Figure 4.2: Benchmark of different isPrime() implementations

+ implementation works up to 2^{31} (when using standard modular exponentiation) or up to $3.4 * 10^{14}$ if using `PowermodExtended`
+ time for integers up to 2^{31}: $O(\log n)$, for bigger integers $O(\log^2 n)$
− harder to implement
∘ see the class `PrimesFast<PowermodAlgorithm>` implemented in *math/primes/primes_fast.h*

The first two algorithms are rather trivial to implement, the Miller-Rabin is a bit trickier. To help the reader decide which algorithm is best suited for him/her (the compromise of difficulty of implementation versus performance), we have done some benchmarks as you may see in figure 4.2.

From these benchmarks, we can conclude that the $O(n)$ implementation is practically useless, as $O(\sqrt{n})$ is much faster on all inputs and the difference between the implementations is minimal. However, usage of the Miller-Rabin algorithm may be a good idea, if you are going to do a lot of computations of numbers exceeding 10^7. For smaller numbers, the decision is solely whether you need very fast (and a bit complicated) implementation.

While implementing these algorithms, there are few standard implementation errors you should look out for:

- general
  - not checking whether the input is nonnegative
  - 1 is a special case and must be handled

- $O(\sqrt{n})$ algorithm
  - using `(int) sqrt((double)n)` as upper limit may not give correct results. Sqrt calculation is usually done at the processor instruction level and there is no guarantee that it won't miss the last significant bit. On current processors it will work though.
  - using `(int) sqrt((double)n) + 1` as upper limit tries to solve the above problem, but introduces a new special case: 2.
  - using `sqrt((double) n)` generally does not work on platforms where integer is 64 bits long, because the conversion to `double` will lose some digits. Using `long double` may help, but this type may not be available on all platforms.
- Miller-Rabin
  - implementation of powermod (integer overflows)

However, primality testing algorithms are not the only ones you may use. There is a big family of algorithms with the purpose of enumerating all primes up to some fixed number n. These algorithms are usually called "sieves" and we implemented some of them:

- Eratosthenes sieve
  - ∘ classical implementation of the algorithm
  - \+ time $O(n \log\log n)$, memory N bits
  - ∘ see the class `EratosthenesBasic` implemented in *math/prime_sieve/eratorsthenes_basic.h*
- Optimized Eratosthenes sieve
  - \+ time $O(n \log\log n)$, memory $N/2$ bits
  - \+ implementation of the Eratosthenes sieve which stores and calculates with only odd numbers, thus achieving faster speed and smaller memory footprint.
  - ∘ see the class `EratosthenesOptimized` implemented in *math/prime_sieve/eratorsthenes_optimized.h*
- Segmented Sieve
  - \+ time $O(n \log\log n)$, memory $O(\sqrt{n})$ (however, primes are not stored, only printed out)
  - ∘ implementation of the sieve from [BH77].
  - ∘ useful for computing some statistics on primes, not intended to be queryable.
  - \+ can be easily parallelized.
  - ∘ see the class `SegmentedSieve` implemented in *math/prime_sieve/segmented_sieve.h*

We considered several other algorithms for sieving, the nice list can be found in the paper [Sor92]. Some of these algorithms are linear and some run even in $o(n)$. However, the results of this paper suggest that there is no big speedup in reality and probably it is not worth implementing unless you need super-fast sieving.

4.1.4 Factorizing integers

After considering primality testing, another great mathematical task involving primes is integer factorization. The goal of integer factorization is to factor the input into set of several primes. A nice list of factorizing algorithms may be found in [Bar04], although the article omits fast algorithms like quadratic sieve or even general number sieve. For this thesis, we selected the following three algorithms:

- naive factorization by trial division
  - − running time $O(\sqrt{n})$.
  - ○ see the class `FactorizeNaive` implemented in *math/factorize/factorize_naive.h*
- Pollard-rho method
  - ○ implementation based on [Pol75]
  - + running time $O(n^{1/4})$.
  - − needs a fast `isPrime()` implementation
  - ○ see the class `FactorizeWithOracle<OraclePollard>` implemented in *math/factorize/factorize_with_oracle.h, math/factorize/oracle_pollard.h*
- Pollard-rho method with Brent's cycle detection
  - ○ implementation based on [Pol75], [Bar04]
  - + running time $O(n^{1/4})$.
  - − needs a fast `isPrime()` implementation
  - ○ see the class `FactorizeWithOracle<OracleBrent>` implemented in *math/factorize/factorize_with_oracle.h, math/factorize/oracle_brent.h*

The usual problems with $O(n^{1/4})$ algorithms are

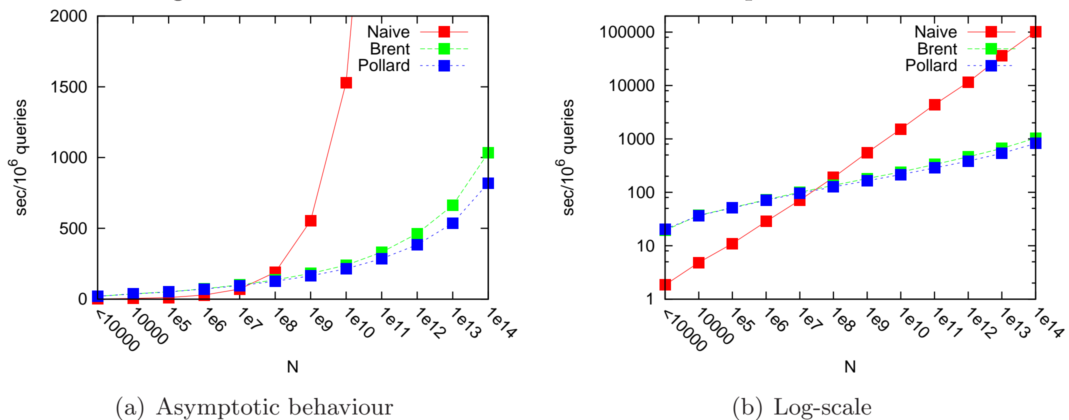
- special cases – for example $n = 4$ cannot be factored with these algorithms, because there is no cycle of convenient length
- similarly, primes cannot be factored and the algorithms may loop on them.
- algorithms must try several times to randomize the `advance()` function, otherwise they might loop with small probability

To help the reader decide if he/she needs to implement a naive method or one of the more complex algorithms, we benchmarked the implementations on sets of numbers. The results of the benchmark may be found in figure 4.3

Although the Pollard factorization method is asymptotically a lot faster than the naive method, the practical experiments showed that the overhead is quite big. The Pollard method was faster starting with $n = 10^8$. In fact, the difference even for $n = 10^9$ is sufficiently small. Thus, our recommendation is that to factor anything which will fit into 32-bit integer, you can use the naive method. Factorization of bigger integers could be more challenging and that would indeed require implementation of Pollard, Brent or Shanks. One of the surprises is that Brent's method is actually slower than Pollard's. This is consistent with the article [Bar04] but it is contradiction with Brent's original article.

If the reader needs to factor numbers which are much bigger (say 64 bits and more), we would recommend implementing the Quadratic Sieve method, or even better using some well-tested implementation.

Figure 4.3: Benchmark of different factorize implementations



(a) Asymptotic behaviour (b) Log-scale

4.1.5 Binary search, root of a function, convex function minimum

Binary search is a classic algorithm that may be found in STL. The problem is that the provided implementation is bogus, see section 2.4.2 for more information about integer overflows during binary search. Thus, we provide our own implementation, which checks for these problems.

- `lower_bound()`
  - ○ finds first position where element can be inserted
  - \+ time $O(\log n)$
  - \+ checks for possible overflows
  - ○ see the function `lower_bound` implemented in *math/binsearch/int_binsearch.h*
- `upper_bound()`
  - ○ finds last position where element can be inserted
  - \+ time $O(\log n)$
  - \+ checks for possible overflows
  - ○ see the function `upper_bound` implemented in *math/binsearch/int_binsearch.h*

But binary search algorithm in sorted arrays is not the only application of binary search. You may use binary search to find the root of a function. This is usually slower than finding root by the Newton's method, but it does not need to compute function derivatives. In practice, it is usually enough to use binary search. Also, there is a ternary-search modification for finding the minimum of a convex function.

We provide an implementation of two methods

- `root()` of a smooth function.
  - – classical binary search
  - \+ time $O(\log(range/precision))$
  - \+ does not loop forever
  - – does not guarantee the required precision
- `convex_min()` – minimum of a convex function.
  - – ternary seach

+ time $O(\log(range/precision))$
+ does not loop forever
− does not guarantee the required precision

For the actual code, see the class `FunctionBinsearch` implemented in *math/binsearch/function_binsearch.h* .

The complications with the implementations are

- the required precision of the result cannot be achieved. This may happen if we require high precision, but the results are big and thus the difference between two consecutive floating-point numbers is bigger than precision. Moreover, the code `while (right - left > precision)` may loop forever.
- precision of the function result itself may be limited. If the function we are interested in is complicated, it may happen that the results will not be correct to all available decimal places. This may violate the preconditions, for example convex function $ax^2 + bx$ may not be convex when computed in floating-point numbers.

4.1.6 Rational numbers

For other algorithms, notably the Gaussian elimination and geometric algorithms, rational numbers are needed (if we want perfect results). Thus, we needed to implement such numbers.

The general problem with rational numbers is their tendency to overflow. Even when we start with decent small numbers up to 100, after several multiplications/divisions and then additions of fractions, we can reach the limits of integers very fast. Thus, the major problem with the implementations is overflow-checking. The number of such overflow-checks in the implementation is enormous. We therefore decided to use a well tested third-party library for that purpose. We used `SafeInt` library from [Mic] This library checks integer overflows by defining a new type `SafeInt<T>` which encapsulates integer of type `T`. Using the `SafeInt` class, the code of rational numbers is more or less straightforward. Special care should be taken to the tests themselves, however. They should check that the `Rational` class is using `SafeInt` in a correct manner and that we are throwing exceptions on all types of overflows.

If the reader is interested in our implementation, he/she may see the class `Rational<IntegerType>` implemented in *math/rational/rational.h*

4.2 Computational geometry

4.2.1 General problems with geometry

The major problem with geometry is the "real-number" character of it. Many mathematically correct formulas are encountering severe problems when used in computer's floating-point arithmetics. For more information, see section 2.5 of this thesis. We will go through the same problems and show them in a geometric context.

Line representation

The classical representation of the line in form

$$y = ax + b$$

for some constants a, b may cause trouble with precision. First of all, there is no convenient way how to represent a vertical line. Moreover, the line is usually entered in a form of two points, rather than the parameters a, b. And the computation of a and b from the point

representation involves division. During this division, small rounding errors may occur and the computation is generally very unstable with near-vertical lines.

Angle comparison

Another very problematic geometry task is angle comparison. Angle comparison is used in several sweep-line geometric algorithms in the stage of sorting the points around some center by increasing angle. Moreover, in several of these algorithms, the order of the points is very important and if there are small discrepancies between the sorted angles, the algorithm may produce different results.

The straightforward way is to use `atan2` function[1] to get the angles and after that sort the angles. Note that the sorting stage can be problematic too, see section 2.2 for that.

But let us talk about the angle comparison again. Program [4.1] with its output [4.2] may suggest that the angles are indeed problematic around the x-axis.

Code Listing 4.1: Angle comparison

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main() {
5     printf("%lf %lf\n",
6         atan2(0.0, -1.0),
7         atan2(-0.0, -1.0));
8 }
```

Code Listing 4.2: Angle comparison output

```
1 3.141593 -3.141593
```

The explanation of such behaviour may be found in the manual of the function `atan2`. The manual suggests that the result greatly depends on the sign of the operands. We may quote the manual of the function `atan2`:

> special values:
>
> - `atan2(+-0, -0)` returns $\pm\pi$.
> - `atan2(+-0, +0)` returns ± 0.
> - `atan2(+-0, x)` returns $\pm\pi$ for $x < 0$.
> - $\cdots$

We may see that the result in case of zero arguments is "random", depending on the sign of the argument. But we cannot guarantee the correct sign of the argument, especially after several floating-point operations. Moreover, the function `atan2` might have problems with infinite arguments and other special cases. And finally, the result of `atan2` will probably be calculated in extended precision and thus comparison during sorting may yield other results than comparison after sorting.

Solution of the problems

The major problem with geometry computations is the presence of division operation, which may produce extremely high numbers when dividing by numbers close to zero. The same applies for functions with discontinuities like `atan2`.

[1] Function `atan2(y, x)` returns an angle ϕ between the point (x, y) and the x-axis.

Most of the problems can be solved though. If the task itself can be specified with only integer arguments, almost all computations can be done in integers as well and we may get rid of the problems. Even in case we must use floating-point numbers, the solution discussed here will probably end up with higher precision.

The key point is to change the representation of the line or line segment. There exists an amazingly natural representation – we may represent a line or line segment as two points (endpoints in case of a segment and any two points in case of a line). The question now is how to implement other operations with lines.

First of all, there is the question of intersection. In the $ax + b$ representation, the intersection of lines could be calculated very easily. Now, the problem is more challenging. If we ask for the point of intersection, the answer can't fit into integer numbers and it would be needed to implement rational numbers or use floating-points (with the knowledge that the result might be rounded). But if we ask only the question "do the lines/line segments intersect?", we may calculate the exact answer with only integers.

We can represent points as "vectors" from point $(0, 0)$ to point (x, y). And we already know the two useful operations on vectors - scalar product and cross product. Now, if we assume that we are dealing with 2D geometry, the cross product is exactly determined. We may use these two simple operations (which need only multiplication and addition) for many simple questions. A positive dot product means that the angle of two vectors is less than 180 degrees. A positive cross product means that the second vector is on the right (seen by the observer directed by first vector). And so on. The cross product may be used to determine the area of a triangle in much more stable way than the Heron's formula. Even the angle comparison may be done with these two products.

Overall, the cross and dot product are the basis of computational geometry and have many uses.

4.2.2 Line segment intersections

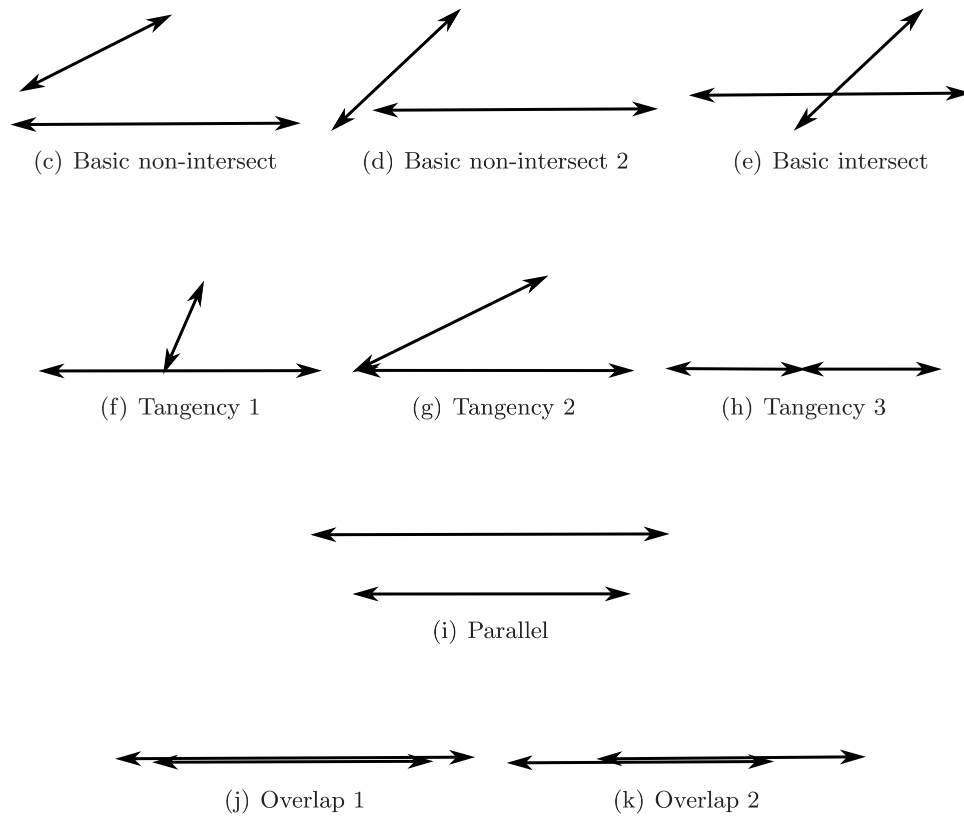
Determining whether line segments intersect may be a quite difficult task. Even if we are using exact integer arithmetics the implementation is hard to get right. The problem arises from many special cases. To show the reader the complexity of this problem we compiled a list of various types of two line segments intersection types. This list is shown in figure 4.4.

The reader may see that instead of two simple cases – the segments intersect or do not intersect, we must deal with much more complicated ones. Tangency is one of the problems. But the most problematic cases are overlaps. And even the overlaps are of different types. Therefore to exactly classify the type of the intersection we need to check many possibilities and the implementation is a bit complicated. We provide the following functions in *geometry/two_d/intersect.h*:

- `bool pointOnLine(Point, Line)`
- `bool pointOnLineSegment(Point, LineSegment)`
- `IntersectType intersectLineLineSegment(Line, LineSegment)`
- `IntersectType intersectLineSegmentLineSegment(LineSegment, LineSegment)`

4.2.3 Angle comparison

We have implemented a reliable angle comparison function. This function uses only multiplications and additions/subtractions. Thus, in integer arithmetics it is exact. Moreover, in floating-point arithmetics it is more precise than the transcendental `atan2` function. We have also implemented a fast function which can determine the quadrant of the point, in a very elegant way using an array of constants.



(c) Basic non-intersect (d) Basic non-intersect 2 (e) Basic intersect

(f) Tangency 1 (g) Tangency 2 (h) Tangency 3

(i) Parallel

(j) Overlap 1 (k) Overlap 2

Figure 4.4: Line segment intersecion cases

- quadrant detection
  - + elegant implementation
  - ◦ see the function `getQuadrant` implemented in *geometry/two_d/angle.h*
- angle comparison
  - ◦ see the function `angleLess` implemented in *geometry/two_d/angle.h*

4.2.4 Convex hull

One of the basic problems in geometry is finding a smallest convex superset of points. Such set is called convex hull. We decided to implement a reasonably fast algorithm for this task. The algorithm is called Monotone Chain Convex Hull and it constructs the upper and lower half of the convex hull separately.

- Monotone Chain convex hull
  - ◦ time $O(n \log n)$, memory $O(n)$
  - + easy to write (almost no special conditions or cases in implementation)
  - ◦ returns points of convex hull in sorted order
  - + removes any unnecessary points from the hull (duplicates, points lying on the edges)
  - ◦ see the class `ConvexHull<T>` implemented in *geometry/two_d/convex_hull.h*

In this implementation, there are no particular issues, but the reader may try to implement some of the other algorithms like Divide and Conquer or Graham Scan algorithms to see a big amount of corner cases and special scenarios.

4.3 Algorithms on strings/sequences

4.3.1 String search

String matching is probably the most important string problem. There are plenty of algorithms that can search for patterns. For the thesis we selected two well known algorithms KMP and Rabin-Karp and less known but extremely fast suffix arrays.

- Knuth-Moris-Pratt
  - ◦ implementation based on [KMJP77]
  - + fast, time $O(n+p)$
  - + deterministic
  - ◦ see the class `Kmp` implemented in *strings/search_kmp/kmp.h*
- Rabin-Karp
  - ◦ implementation based on [KR87]
  - − probabilistic
  - + time $O(n+p)$ if false positives are allowed (with small probability)
  - − worst-case $O(np)$ if the check for false positives is in place
  - ◦ see the class `RabinKarp` implemented in *strings/search_rabin_karp/rabin_karp.h*
- Suffix array search
  - ◦ implementation based on [MM90]
  - + very fast for fixed text after building the suffix array
  - + time $O(\log p)$
  - − building the suffix array is relatively hard to implement
  - ◦ more on suffix arrays in section 4.3.3

Common problems with these implementations are

- special cases – pattern of length 1 may cause buffer overflow in some implementations of KMP
- KMP might be a bit tricky to get it right, especially the automata backlink to epsilon state
- bad hashing function for Rabin-Karp algorithm – see next section

We recommend implementing the KMP algorithm for almost any occasion. The Rabin-Karp algorithm is slow because the hash function computation involves several arithmetic operations (versus fast memory lookups in KMP) and it may produce false positives (if not checked). Rabin-Karp might be a good idea only in case one needs to match several patterns of the same length. In that case, it is probably easier to implement than Aho-Corasick (a generalization of KMP to more patterns). The suffix arrays are extremely fast, if they are pre-built for a specific text, but their building is quite complex to implement. The reader may consult section 4.3.3 for more information about possible algorithms for building suffix arrays.

Testcase	KMP	RabinKarp	Suffix array build[1]	suffix array search
100000 random letters, small patterns	1.3ms	25ms	120ms	0.007ms
10^6 digits of pi, small patterns	15ms	300ms	2.6s	0.007ms
bible, small patterns	64ms	1.1s	36s	0.008ms
bible, big patterns	240ms	2s	36s	0.009ms
16MB apache logs, small patterns	210ms	4.2s	150s	0.01ms
60MB of human genome, 0 small matches	0.9s	15s	1400s	0.01ms
60MB of human genome, 500000 small matches	0.9s	17s	1400s	9ms
60MB of human genome, 500000 small matches	0.9s	17s	1400s	9ms
60MB of human genome, 650000 long matches	0.9s	17s / 630s[2]	1400s	42ms
100MB PHP source code tar, long patterns	1.3s	26s	2200s	1ms

1 – using our $O(n \log^2 n)$ implementation of Manber-Myers idea, see the section 4.3.3. Note that there are much faster implementations available.
2 – with checks for false positives

Figure 4.5: String search benchmark

4.3.2 Theoretical analysis of the rolling hash

In the Rabin-Karp algorithm [KR87], the concept of a rolling hash function is used. Basically, a rolling hash function is a hash function for a window of fixed length over the input. The concept of the rolling hash is shown on the figure 4.6. The basic property of the rolling hash

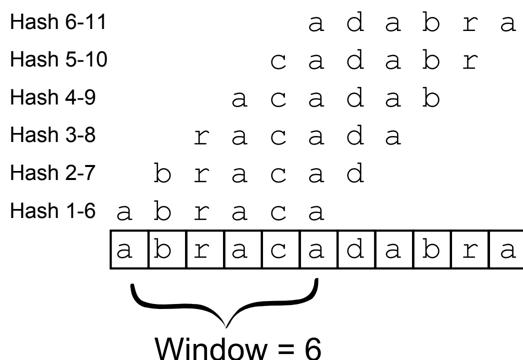


Figure 4.6: Rolling hash

is that there is is an easy way to recompute the hash when moving the window by one. This property makes it useful to detect fixed-size patterns in the text.

The math definition of the Rabin-Karp rolling hash is

$$h_{start,len}(x) = \sum_{i=start}^{start+len-1} c^{len-1-i} x[i] \pmod{m} \tag{4.1}$$

For the Rabin-Karp algorithm to be as fast as possible, there is an effort to maximize speed. The original Rabin-Karp algorithm is based on rolling hash modulo a random prime number p and under these conditions, it is guaranteed that the expected number of collisions is small.

In this part of the thesis, we theoretically examine the possibility of skipping the modulo p part and using overflows in integer arithmetics to do the modulo. This is considered to be a good optimization as we do not need to calculate the modulo operation, and also we do not need an extended data type (previously, if the hash was 32-bit long, we needed 64-bit operations during the computation). This variation of the rolling hash function has not been theoretically examined yet and some people errorneously believe that it is safe to use such modulo [DJ97].

The problem with the "optimization" is that the original average-case analysis of the collisions in [KR87] does not hold.

Bad behavior of rolling hash using modulo 2^n

In this section we will present our analysis of the rolling hash function computed modulo 2^n. We will start with the following lemma

Lemma 4.3.1 *Let $\{x[i]\}$, $\{y[i]\}$ be two sequences of the same length for which the rolling hash function collides. Then the hash of the sequence $\{x[i] - y[i]\}$ and the hash of the null sequence (of the same length) collides too.*

The result of this lemma is that we just need to find collisions of sequences with the null sequence. In other words, we need to find only sequences with a zero hash value.

Lemma 4.3.2 *Let constant c in the rolling hash function be even and $m = 2^n$. Then the rolling hash of two sequences of same length collides iff the rolling hash of their last n characters collides.*

Proof: We know that $c^n = 0 \pmod{m}$ because with each multiplication the number of ending zeroes in binary representation of c^i increases at least by one. □

We can see that setting c even results in very undesirable behaviour as we are completely ignoring almost everything from the sequences itself.

Moreover, the probability of a collision is also higher than it should be. At least, with c even, the hash function cannot have an odd result in 50% of cases as the parity of the result is dependent on the parity of the last character in the sequence. If c is divisible by a greater power of two, that may further increase the probability of a collision. The exact same thing holds for the value of sequence elements itself – if they are divisible by some power of two, they contribute with fewer bits to the final hash and the probability of finding a collision (with the zero sequence) is again increasing.

We have seen that even values of c have big undesired effects and therefore should be avoided. The usual solution is thus to avoid even values and use only odd values (which are coprime to m). This approach turns to contain problems as well.

Our main result about odd values of c is formulated in the following theorem:

Theorem 4.3.1 *Let $m = 2^n$. If $(k+1)(k+2) \geq 2n$, then then there exists a sequence of length 2^k over values $x_i \in \{-1, 1\}$ with the zero hash value* **for any odd value of c.**

The fact that the collision exists is self-evident, because hash function is much longer than the length of the hash itself. The interesting part is, that we need only plus/minus one. But the most important fact is that this is a collision **for any** *c and that the sequence is extremely easy to construct. Also, the constructed sequence will have some quite interesting properties.*
The consequence of this is that even when we use randomization of constant c to avoid collisions, bad adversary can supply a sequence which will have many collisions. This is especially important in case we are relying on the fact that collisions are improbable and we do not check for false positives in the algorithm to provide linear worst-case time.

Proof: The basis of our proof lies in the following algebraic identities:

$$\begin{aligned}
1-c &= 1-c \\
1-c^2 &= (1-c)(1+c) \\
1-c^4 &= (1-c^2)(1+c^2) = (1-c)(1+c)(1+c^2) \\
1-c^8 &= (1-c^4)(1+c^4) = (1-c)(1+c)(1+c^2)(1+c^4) \\
&\vdots \\
1-c^{2^k} &= (1-c)(1+c)(1+c^2)\cdots(1+c^{2^k-1})
\end{aligned}$$

The reader may have noticed that because c is odd, all expressions in parentheses are even. If we now take the expression

$$(1-c)(1-c^2)\cdots(1-c^{2^k}) \tag{4.2}$$

we have multiplication of even numbers. The count of even numbers in the result is $1+2+3+...+(k+1) = (k+1)(k+2)/2$. Therefore for k such that $(k+1)(k+2) \geq 2n$ we get

$hash = 0$.

Moreover, the expression 4.2 is defined quite nicely as a recurrent sequence:

$$\begin{aligned}
x_1 &= (1) \\
x_{1..2} &= (1, -1) = (x_1, -x_1) \\
x_{1..4} &= (1, -1, -1, 1) = (x_{1..2}, -x_{1..2}) \\
x_{1..8} &= (1, -1, -1, 1, -1, 1, 1, -1) = (x_{1..4}, -x_{1..4}) \\
&\vdots \\
x_{1..2^{k+1}} &= (x_{1..2^k}, -x_{1..2^k})
\end{aligned}$$

□

In real-world scenario, $n = 32$ or $n = 64$. Taking the second as an example, we have $11 * 12 > 2 * 62$ and thus $k = 10$ is sufficient. To summarize our result – we have sequence of length 1024 which collides with zero-sequence for **arbitrary odd** c.

High probability of shorter collision sequences

Our results can be improved even more.

Suppose we consider setting $k = 6$. In that case, the hash of a sequence of length $2^k = 64$ will lose $(k+1)*(k+2)/2 = 28$ bits of security. For $n = 32$ this may be a pretty big loss. But in fact, this analysis is only the "best-case" analysis. If we assume that (for fixed c) amongst $(k + 1) * (k + 2)/2$ even numbers around one half is divisible by 4, one fourth is divisible by 8, etc. we will come to the result that expected number of zeroes at the end of the hash is $(k + 1) * (k + 2) - 1$. Thus for $k = 7$ a collision will occur with high probability.

To sum up – we have found a nice recursively defined sequence of length 128, for which there is a collision with the zero sequence for almost all c. The variant of the rolling hash without using modular division by a prime number is therefore not resistant against collisions and should be avoided in Rabin-Karp algorithm.

4.3.3 Suffix arrays

While Knuth-Moriss-Pratt, Aho-Corasick and Rabin-Karp algorithms both preprocessed the pattern and then could be used to search for a pattern in the text, some problems require the opposite way. If the text is fixed, but there may be many patterns, it would make sense to preprocess the text into some structure and use that structure for faster searches later. The example of such scenarios is DNA sequencing – we want to preprocess the whole big sequence and then there will be many online queries for localising parts of the sequence.

There are several structures that support such operations, for example suffix trees, suffix arrays and some special types of indexes. In this section we will discuss the suffix arrays which are easier to implement and less memory-intensive than suffix trees.

Suffix arrays were first introduced by Udi Manber and Gene Myers in [MM90] as a simple replacement for suffix trees. After this paper there were several improvements made and many of the searching problems today rely on this structure.

Currently, there exist several different algorithms for creating suffix arrays from the string in linear time, but such are quite complicated. Moreover, the difference in running times of various algorithms may differ significantly [Mor10]. After consulting implementations on page [NW09] the decision was made to implement the simplest $O(n \log n)$ algorithm from article [MM90]. Another very nice implementation can be found in the article [KSB06]. We did not

reimplement this algorithm, as the algorithm in the electronic attachment to the article is quite great. In fact, if you are looking for a nicely-written and moderately fast suffix array implementation with small memory footprint, that implementation would be great for you.

If you need a bleeding-edge fast algorithm, consult [Mor10], as this implementation is very well tuned and extremely fast.

In the thesis we implemented three algorithms for constructing a suffix array.

- naive implementation
  - − running time $O(n^2 \log n)$ worst-case
  - + uses $4n$ bytes of memory
  - + if the data contains relatively short longest common prefixes, the algorithm may be relatively fast
  - − the problem is that the worst-case may be reached with real datasets (DNA sequences with lots of unknown bases)
  - + very easy to implement
  - ◦ see the class `NaiveSuffixArray` implemented in *strings/suffix_array_naive/naive.h*
- $O(n \log n)$ by Manber Myers
  - ◦ This is the implementation of the (first ever) suffix array construction from the article [MM90]
  - + $O(n \log n)$ worst case
  - − uses $12n$ bytes of memory (according to the article, this can be further reduced to $8n$ bytes of memory, but the article is not very clear about how to implement it)
  - ◦ see the class `ManberMyers` implemented in *strings/suffix_array_myers/manber_myers.h*
- $O(n \log^2 n)$ modification of Manber Myers idea
  - ◦ The previous algorithm used count-sort to sort the data, this algorithms takes the idea from there, but uses standard sorting function from STL.
  - + $O(n \log^2 n)$ worst-case
  - − uses $16n$ bytes of memory
  - + slightly less complicated to implement than Manber-Myers
  - ◦ see the class `ManberMyersLog2` implemented in *strings/suffix_array_log2/manber_myers_log2.h*

The results of benchmarks are in figure 4.7. Based on the results of the benchmark, we can conclude that the naive suffix array implementation is quite fast, if the text is not degenerated and does not have long common prefixes. It is a very good choice if you need a simple and relatively fast algorithm and you know that the data does not have long common prefixes. The $O(n \log^2 n)$ version seems to be faster than $O(n \log n)$ for the big inputs, probably because of smaller cache-miss count. Overall, our implementations are several times slower than fast suffix array implementations running in linear time. Thus, we recommend using our suffix array implementations only on smaller sequences (maximum of several megabytes) or when you need to build the suffix array only once in a while and you do not care for additional time.

data file	**size**	**time (sec)**					
		Naive	**MM**	**MM** $n \log^2 n$	**BK**	**LS**	**YM**
Alphabet – A..Z repeated many times	30 000	12.83	0.08	0.05	-	-	-
Random – 10^5 random characters	100 000	0.16	0.08	0.31	-	-	-
Pi – 10^6 digits of number π	1 000 000	1.72	1.81	4.36	-	-	-
Factbook – Text of The World Factbook 1992	2 473 400	5.48	12.11	13.81	-	-	-
Bible – Text of the bible	4 047 392	9.47	22.91	23.39	-	-	-
E.Coli – Genome of E.Coli	4 638 690	11.66	31.84	29.66	-	-	-
Apache logs – server logs	16 780 691	82.1	90.4	123.2	24.9	21.9	5.1
Chromosome Y – human genome	60 561 044	$>$ 5hours	806	598	102	203	16.3
PHP – tar of source codes	93 655 040	1460	1038	797	121	184	31.0

Shortcuts: MM – Manber Myers, BK – Burkhardt Karkkäinnen, LS – Larsson Sadakane, YM – Yuta Mori

Figure 4.7: Benchmarks of various suffix array implementations

data file				**results**		
File	Size	Avg. LCP	max LCP	Naive	Kasai	Manzini
random.txt	1000000	2.12	5	< 0.1s	< 0.1s	< 0.1s
pi.txt	1000000	5.31	12	< 0.1s	0.34s	0.5s
bible.txt	4047392	13.97	551	0.83s	1.4s	2.4s
world192.txt	2473400	22.01	559	0.45s	0.8s	1.3s
E.coli	4638690	17.38	2815	1.1s	1.9s	3s
alphabet.small	30000	14975	29974	2.1s	< 0.1s	< 0.1s
access.log	16780691	103.89	1422	10.7s	4.4s	8.8s
chrY.fa	60561044	7808302.5	30599949	> 5hours	18.4s	29.3s
php-5.3.5.tar	93655040	1370.4	230400	615s	26s	63s

Figure 4.8: LCP array calculation

Longest Common Prefix

Suffix array search and several other algorithms that query the suffix array may be improved with additional information about the suffixes. This information is called longest common prefix (LCP) and it is an array of sizes of prefixes of every two adjacent suffixes in the suffix array. There are simple, but sophisticated algorithms for LCP computation and we implemented the following of them

- naive LCP computation
  - + easy to implement
  - − worst-case $O(n^2)$ ($\Theta(L)$ where L is sum of LCP array).
  - + no additional memory needed
  - ∘ see the class `LCPNaive` implemented in *strings/suffix_array_lcp_naive/lcp_naive.h*
- fast computation by Kasai et al.
  - ∘ implementation based on [LKL+06]
  - + time $O(n)$
  - − $4n$ bytes of additional memory needed
  - ∘ see the class `LCPKasai` implemented in *strings/suffix_array_lcp_kasai/lcp_kasai.h*
- fast computation by Manzini
  - + enhancement of previous algorithm for less memory
  - ∘ implementation based on [Man04]
  - + time $O(n)$
  - + $O(\sigma)$ bytes of additional memory needed
  - ∘ see the class `LCPKasai` implemented in *strings/suffix_array_lcp_manzini/lcp_manzini.h*

The recommended implementation is Kasai's algorithm, because it is easy and fast. If you need an implementation of LCP calculation with tight memory limitations or you are processing very big sequences, the improvement made by Manzini might be a good idea.

Checking

After building the array or reading it from some storage, it is good to check its consistency. The trivial algorithm checks suffixes in the same way as naive LCP computation, thus the running time is worst-case $O(n^2)$. There is however a very simple and effective algorithm available. We implemented it in order to check the consistency of our own suffix array algorithms.

- fast suffix array checking
  - ∘ based on paper [BK03]
  - \+ simple and fast
  - \+ time $O(n)$
  - ∘ see the class `SuffixArrayChecker<T>` implemented in *strings/suffix_array_check/suffix_array_check.h*

4.3.4 Longest common subsequence

One of the standard problems in string/sequence matching is determining the longest common subsequence of two sequences. The algorithm may be also used to determine the edit distance between strings or to align two strings. The LCS problem can be solved by dynamic programming in quadratic time, there is however a complication with memory, because it too is quadratic. Hirschberg in [Hir75] introduced a trick how to decrease this memory just to linear. There are other optimizations (mainly reducing the running time by $\log n$), but they are impractical, as the speedup is not so big and they are quite complex. In the thesis, we implemented the basic algorithms:

- dynamic programming (returns only length)
  - ∘ simple
  - ∘ running time $O(nm)$
  - \+ memory $O(n+m)$
  - − see the class `LCS` implemented in *strings/lcs/lcs.h*
- dynamic programming (returns also subsequence)
  - ∘ simple
  - ∘ running time $O(nm)$
  - − memory $O(nm)$
  - − see the class `LCS` implemented in *strings/lcs/lcs.h*
- Hirschberg's algorithm (returns also subsequence)
  - ∘ based on [Hir75]
  - ∘ running time $O(nm)$
  - \+ memory $O(n+m)$
  - − see the class `LCSHirschberg` implemented in *strings/lcs/lcs_hirschberg.h*

We strongly recommend to use the first implementation, if possible, because it is straightforward. In case the reader really needs also the subsequence and not only its length, the decision depends solely on the sizes of sequences. Hirschberg's algorithm is quite useful for $n, m > 5000$.

4.3.5 Minimal cyclic shift

In case of cyclic sequences, it is usually important to compare them effectively. The problem with basic comparison is that there are n possible rotations of the string that need to be taken into account. One convenient way how to solve this problem is "normalizing" the sequences. The natural normalized representation of a cyclic sequence is such a rotation that is lexicographically minimal. It is therefore crucial to be able to compute such rotation very fast. For this purpose, we implemented algorithms from [Duv83] which are solving this and similar problems. The reader may wish to see the class `Duval` implemented in *strings/cyclic/duval.h* .

If the reader wants to implement Duval's algorithms, we have a warning for him/her. We have found (and mentioned earlier in section 2.1) that pseudocode from the original paper has several buffer overflows/out-of-bounds errors.

4.4 Balanced structures

4.4.1 Interval trees and balanced data structures

Interval trees

In algorithm competitions, there is a big need of data structures that can do some operations on whole intervals. Such data structures are usually variants of the Interval Tree – a binary tree representing intervals. In such trees, "statistic" operations like sum over the whole interval or maximum over it can be done efficiently in $O(\log n)$ time. Some of the interval trees can even support data-altering operations on whole intervals efficiently. These structures are however tightly tied for their purpose and cannot be easily generalized. In this thesis, we implemented two very efficient versions of simple interval trees which can update one position and query whole interval.

- Simple interval tree
  - ○ computes maximums over intervals
  - \+ can set/update one value in $O(\log n)$ time
  - \+ can query interval $[x, y)$ in $O(\log n)$ time
  - \+ implementation without recursion
  - ○ see the class `SimpleMaxTree` implemented in *interval_trees/simple/simple_max.h*
- Fenwick interval tree
  - ○ computes sum/maximum over intervals
  - ○ based on [Fen94]
  - \+ can set/update one value in $O(\log n)$ time
  - \+ can query interval $[0, x)$ (or $[x, n)$ depending on the variant of the implementation) in $O(\log n)$ time
  - \+ extremely easy to implement, implementation takes only several lines
  - \+ very fast in practice
  - ○ see the class `FenwickTree` implemented in *interval_trees/fenwick/fenwick.h*

Our recommendation is to write Fenwick tree if it is possible. It does not possess the full power of interval trees as it can only query ranges with one fixed end, but this is in many cases sufficient. This disadvantage is, however, greatly overshadowed by its simplicity – the core of the algorithm takes only a few lines and there is almost no room for a mistake. Moreover, the algorithm is extremely fast, as can be seen in figure 4.9

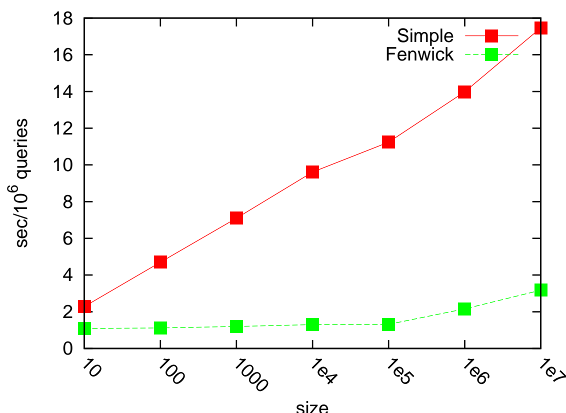


Figure 4.9: Benchmark of interval trees

4.4.2 Skiplist

One of the most useful data structures available in STL is `std::set`. A `set` can hold a set of items, storing them in a balanced Red-Black tree. The advantage of the `set` is the time complexity needed by various operations. Operations like insert, delete and find all run in $O(\log n)$ where n is the size of the set. These operations are quite useful, but the `std::set` is missing two important operations, which are sometimes required. They are

- find k-th element
- inverse of k-th element. Given an element (iterator), return the position in the set.

Both of these functions can be implemented in Red-Black tree in $O(\log n)$ but the implementation of the tree is difficult. Therefore we provide an alternative data structure called skiplist, which is a randomized data structure with all operations expected to be $O(\log n)$.

Our implementation supports the following functions, all in $O(\log n)$

- `insert(value)`
- `find(value)`
- `erase(iterator)`
- `kth(position)`
- `xth(iterator)`
- iterator functions: `begin(), end(), iterator++, iterator--`

Thus, we provide a full alternative for `std::set` with additional functions. The comparison of `std::set` and `Skiplist` is available in figure 4.10. Based on these results, we may conclude that our implementation is roughly 3 times slower and takes 2.5 times more memory than `std::set`. But the presence of the new functions may prove useful in some cases. For details see the class `Skiplist` implemented in *balanced_structures/skiplist/skiplist.h*

operation ($2.5 * 10^6$ elements)	set	skiplist
memory per element, 64-bit	44 bytes	108.2 bytes
memory per element, 32-bit	20 bytes	52.3 bytes
insert random	10s	37s
delete random	11s	55s
insert ascending	9s	14s
delete ascending	3s	11s
insert descending	8s	7s
delete descending	2s	20s
find	9s	36s
kth	N/A	15s
xth	N/A	3s

Figure 4.10: Skiplist benchmark

Chapter 5

Conclusion

The goal of the thesis was to create a set of implementations of various algorithms. We fulfilled this goal by providing a rich set of algorithms which are well-documented. The different algorithms are tested against each other and tested for special cases. Moreover, we tried to minimize the descriptive complexity and optimize performance. In the thesis, we also created a comprehensive list of potential problems and issues. The list helped us with the implementation task and we hope it will help other programmers as well – it should be a good reminder of what can go wrong with implementations. We hope that the listing of the attached programs will help programmers to learn how to write robust error-free code and that it will help ACM contestants by providing a set of well-tested and easy to write algorithms needed in this competition. Thus we conclude that the thesis fulfills all the goals.

But our work should not end here. We recognize that there are many useful algorithms not covered in the thesis. The future work should therefore include a variety of new algorithms. Also, in the future we should improve the existing implementations – there is still plenty of issues which need to be addressed (many tests for special cases, better documentation of some algorithms, introducing new tricks to reduce descriptive complexity).

Source code of selected algorithms

.1 Benchmarking

Code Listing 1: benchmark.h

```
#ifndef H_BENCHMARK
#define H_BENCHMARK

/**
 * @file contains BENCHMARK and AUTO_BENCHMARK macros
 */

#include <limits>
#include "color.h"
#include "utils/timer/timer.h"
#include "utils/static_assert/static_assert.h"

namespace utils {
namespace benchmark {

/**
 * Minimum time (in seconds) for benchmark
 * to have usedul results.
 *
 * Note: current implementation of benchmark is using
 * system timer to determine running time.
 * This have the best resolution of 16ms and may
 * wildly change, the present constant is
 * conservative for reliable benchmarking
 */
const double MIN_BENCHMARK_TIME = 1.5;

/**
 * @param times How many times the test was run
 * @param run_time_sec Run time of the test in seconds
 * @param function_str string containing the name
 *         of the function and arguments
 */
void printBenchmarkResults(
    long long int times,
    double run_time_sec,
    const char* function_str)
{
  bool isLongEnough = run_time_sec > MIN_BENCHMARK_TIME;

  color::Color color = isLongEnough ? color::CYAN : color::PINK;
  double avg_time_sec = run_time_sec / times;
  double times_per_sec = 1 / avg_time_sec;

  color::colorPrintf(color,
    "BENCHMARK: time %6.2lfs avg.time %10.5lfms "
```

```
47      "%12.5lf times/ms, %10lldx %s\n",
48      run_time_sec, avg_time_sec*1000, times_per_sec/1000, times, function_str);
49  }
50
51
52  /**
53   * Macro for benchmarking code.
54   *
55   * The benchmark runs specified \a times the code
56   * and prints nicely formatted report to stdout.
57   * Usage:
58   * <code>
59   *   BENCHMARK(100, my_function());
60   * </code>
61   * calls 100 times function my_function.
62   *
63   * Note: "function" does not need to be the function,
64   * in fact, it may be sequence of statements of event the {} block.
65   *
66   * @warning Sometimes compiler optimizes out parts of you code!
67   *
68   * @param times number of times the \a code should be run
69   * @param code code that should be run
70   *
71   */
72  #define BENCHMARK(times, code) { \
73      STATIC_ASSERT(std::numeric_limits<typeof(times)>::is_integer, "") \
74      ::utils::timer::Timer timer = ::utils::timer::Timer(); \
75      for (typeof(times) q__= 0; q__ < times; q__++) { \
76          code;\
77      } \
78      ::utils::benchmark::printBenchmarkResults(\
79          times, timer.elapsed_time_sec(), #code); \
80    }
81
82  /**
83   * Macro for benchmarking the code
84   *.
85   * The benchmark runs the code several times depenging
86   * on the speed of execution and prints nicely formatted report to stdout.
87   * Usage:
88   * <pre>
89   *   AUTO_BENCHMARK(my_function());
90   * </pre>
91   * calls the function my_function for approximately
92   * \a MIN_BENCHMARK_TIME seconds.
93   *
94   * Note: This macro uses exponential decay to guess the correct
95   * number of iterations, and so that instrumentation is not big.
96   *
97   * @warning Sometimes compiler optimizes out parts of you code!
98   *
99   * @param code code that should be run
100  */
101 #define AUTO_BENCHMARK(code) { \
102     ::utils::timer::Timer timer = ::utils::timer::Timer(); \
103     long long int times = 0; \
104     int i = 0; \
105     while ((timer.elapsed_time_sec() < ::utils::benchmark::MIN_BENCHMARK_TIME)
          && \
106         (i < std::numeric_limits<long long int>::digits)) { \
107       long long int t = 1LL << i; \
108         for (long long q__ = 0; q__ < t; q__++) { \
```

```
            code; \
        } \
      times += t; \
      i++; \
    } \
    ::utils::benchmark::printBenchmarkResults(\
        times, timer.elapsed_time_sec(), #code); \
  }

} // namespace benchmark
} // namespace utils
#endif
```

Example output:

```
BENCHMARK: time   1.20s avg.time   12.00000ms      0.08333 times/ms,       100
    x SegmentedSieve::findPrimes(1 * Mi, &c)
BENCHMARK: time   1.18s avg.time  118.00000ms      0.00847 times/ms,        10
    x SegmentedSieve::findPrimes(10 * Mi, &c)
BENCHMARK: time   1.21s avg.time 1210.00000ms      0.00083 times/ms,         1
    x SegmentedSieve::findPrimes(100 * Mi, &c)
```

.2 Preconditions

Code Listing 2: int_binsearch.h

```
#ifndef H_PRECONDITIONS
#define H_PRECONDITIONS

#include <stdexcept>
#include <stdio.h>
#include "utils/branch_predict/branch_predict.h"

// Note: Preconditions is in global namespace!
/**
 * Preconditions is a helper class containing static methods that
 * can check the validity of function arguments (function pre-conditions).
 * In case there is a failure, Preconditions will throw an
 * std::invalid_argument error.
 *
 * Note: Do not overuse the throwing/catching exceptions. Failed preconditions
 * are running much slower because of branch prediction and slowness of
 * exceptions.
 */
class Preconditions {
 public:
  /**
   * Checks whether the expression is true
   *
   * @param expression value that is expected to be true
   *
   * @throws invalid_argument if expression is false
   */
  static void check(bool expression) {
    Preconditions::check(expression, "Precondition failed");
  }

  /**
   * Checks whether the expression is true
   *
   * @param expression value that is expected to be true
```

```
36     * @param message of the thrown expection if case of failure
37     *
38     * @throws invalid_argument if expression is false
39     */
40   static void check(bool expression, const char* message) {
41     if (UNLIKELY(message == NULL)) { // self-check
42       throw std::invalid_argument("message shoudn't be NULL");
43     }
44
45     if (UNLIKELY(!expression)) {
46       throw std::invalid_argument(message);
47     }
48   }
49
50   /**
51    * Check that index is in range [0, size)
52    *
53    * Warning: never ever override this template for using
54    * two different types - it may end up with
55    * nasty results because of casting like
56    * checkRange(unsigned int, int) and check will be done in ints!
57    *
58    * @throws invalid_argument in case of an error
59    */
60   template <typename T>
61   static void checkRange(T index, T size) {
62     // Note: conversion of zero to T is required!
63     Preconditions::checkRange(index, (T)0, size);
64   }
65
66   /**
67    * Check that index is in range [low, high)
68    *
69    * @see warnings for checkRange(index, size) implementation
70    *
71    * @throws invalid_argument in case of an error
72    */
73   template<typename T>
74   static void checkRange(T index, T low, T high) {
75     Preconditions::check(low < high, "Bad range!");
76     Preconditions::check(index >= low, "Index out of range");
77     Preconditions::check(index < high, "Index out of range");
78   }
79
80   /**
81    * Checks that pointer is not NULL.
82    *
83    * @throws invalid_argument in case of an error
84    */
85   template<typename T>
86   static void checkNotNull(const T* ptr) {
87     Preconditions::check(ptr != NULL, "Variable can't be null pointer!");
88   }
89 };
90
91 #endif
```

.3 Sample code – binsearch

Code Listing 3: preconditions.h

```
1  #ifndef H_MATH_BINSEARCH_INT_BINSEARCH
2  #define H_MATH_BINSEARCH_INT_BINSEARCH
3  /**
4   * @file Contains lower_bound and upper_bound
5   * functions similar to std::lower_bound and std::upper_bound
6   */
7  #include "utils/preconditions/preconditions.h"
8  #include "utils/static_assert/static_assert.h"
9  #include <stdexcept>
10
11 namespace math {
12 namespace binsearch {
13
14 /**
15  * Finds the middle of the range <i> [left, right) </i>
16  *
17  * Middle is defined as <i> floor((left + right) / 2) </i>
18  *
19  * @precondition \a T is integral type
20  * @precondition <i> (right-left) </i> is representable in type \a T
21  *
22  * @param left start of the interval
23  * @param right first index after the end of the interval
24  *
25  * @returns midde of the interval
26  */
27 template <typename T>
28 T range_middle(T left, T right) {
29   STATIC_ASSERT(std::numeric_limits<T>::is_integer,
30       "T should be of integral type");
31   Preconditions::check(left < right, "Invalid range");
32   T len = right - left;
33   if (len <= 0) {
34     throw std::overflow_error("Too big range!");
35   }
36   return left + len / 2;
37 }
38
39 /**
40  * Find first index in array range <i> [left, right) </i>
41  * where the value may be inserted without violating the ordering
42  *
43  * Note that the definition is same as
44  * ``index of first element which is >= value''
45  * except that the result is <i> right </i> if no such value exists
46  *
47  * Example:
48  * <pre>
49  *      a = 1 1 2 2 2 3 5
50  * lb(1) = ^
51  * lb(2) =     ^
52  * lb(4) =           ^
53  * lb(6) =             ^ (==right)
54  * </pre>
55  *
56  * @precondition sorted array
57  * @precondition \a SizeType is integral type
58  * @precondition <i>(right-left)</i> will fit into type \a SizeType
59  *
60  * @param left start of the interval
61  * @param right index after the end of the interval
62  *
63  * @returns index of the binsearched value
```

```
64  *
65  */
66 template <typename ValueType, typename SizeType>
67   SizeType lower_bound(ValueType pole[], SizeType left, SizeType right,
       ValueType value)
68 {
69   STATIC_ASSERT(std::numeric_limits<SizeType>::is_integer,
70       "SizeType should be of integral type");
71   Preconditions::check(left <= right, "Invalid range");
72   if (left == right) {
73     return left;
74   }
75   if (std::numeric_limits<SizeType>::is_signed) {
76     // Checks for signed values overflow.
77     // Note that right - left == 0 could be also caused by overflow!
78     if (right - left < 0) {
79       throw std::overflow_error("Too big range!");
80     }
81   }
82   while (right - left > 0) {
83     SizeType middle = range_middle(left, right);
84     if (pole[middle] < value) {
85       left = middle + 1;
86     } else {
87       right = middle;
88     }
89   }
90   return left;
91 }
92
93 /**
94  * Finds last position from range <i>[left, right) </i>
95  * where the value may be inserted without violating ordering
96  *
97  * Note that the definitions is the same as
98  * ``index of first element that is greater than value''
99  * except that the result is <i> right </i> if no such value exists
100  *
101  * Example:
102  * <pre>
103  *        1 1 2 2 3 5
104  * ub(6)              ^ (== right)
105  * ub(2)          ^
106  * ub(1)      ^
107  * ub(0)  ^
108  * </pre>
109  *
110  * @precondition <i> (right-left) </i> should fit into ValueType
111  * @precondition \a SizeType should be integral type
112  *
113  * @param left start of the interval
114  * @param right index after the end of the interval
115  *
116  * @returns index of the binsearched value
117  */
118 template <typename ValueType, typename SizeType>
119 SizeType upper_bound(ValueType pole[], SizeType left, SizeType right, ValueType
      value) {
120   Preconditions::check(left <= right, "Invalid range");
121   if (left == right) {
122     return left;
123   }
124   if (std::numeric_limits<SizeType>::is_signed) {
```

```
125      // Checks for signed values overflow.
126      // Note that right - left == 0 could be also caused by overflow!
127      if (right - left <= 0) {
128        throw std::overflow_error("Too big range");
129      }
130    }
131    while (right - left > 0) {
132      SizeType middle = range_middle(left, right);
133      if (value < pole[middle]) {
134        right = middle;
135      } else {
136        left = middle + 1;
137      }
138    }
139    return left;
140  }
141
142  } // namespace binsearch
143  } // namespace math
144  #endif
```

.4 Sample test – heap operations

Code Listing 4: heap_unittest.cpp

```
1   #include "heap.h"
2   #include "gtest/gtest.h"
3
4   namespace heap {
5   TEST(HeapOperations, left) {
6     EXPECT_THROW(left(0), std::invalid_argument);
7     EXPECT_EQ(2, left(1));
8     EXPECT_EQ(4, left(2));
9     EXPECT_EQ(6, left(3));
10    EXPECT_EQ(8, left(4));
11    EXPECT_EQ(10, left(5));
12    EXPECT_EQ(12, left(6));
13    EXPECT_EQ(14, left(7));
14  }
15
16  TEST(HeapOperations, right) {
17    EXPECT_THROW(right(0), std::invalid_argument);
18    EXPECT_EQ(3, right(1));
19    EXPECT_EQ(5, right(2));
20    EXPECT_EQ(7, right(3));
21    EXPECT_EQ(9, right(4));
22    EXPECT_EQ(11, right(5));
23    EXPECT_EQ(13, right(6));
24    EXPECT_EQ(15, right(7));
25  }
26
27  TEST(HeapOperations, parent) {
28    EXPECT_THROW(parent(0), std::invalid_argument);
29    EXPECT_THROW(parent(1), std::invalid_argument);
30
31    EXPECT_EQ(1, parent(2));
32    EXPECT_EQ(1, parent(3));
33
34    EXPECT_EQ(2, parent(4));
35    EXPECT_EQ(2, parent(5));
36
```

```
37   EXPECT_EQ(3, parent(6));
38   EXPECT_EQ(3, parent(7));
39 
40   EXPECT_EQ(4, parent(8));
41   EXPECT_EQ(4, parent(9));
42 
43   EXPECT_EQ(5, parent(10));
44   EXPECT_EQ(5, parent(11));
45 
46   EXPECT_EQ(6, parent(12));
47   EXPECT_EQ(6, parent(13));
48 
49   EXPECT_EQ(7, parent(14));
50   EXPECT_EQ(7, parent(15));
51 }
52 
53 TEST(HeapOperations, sibling) {
54   EXPECT_THROW(sibling(0), std::invalid_argument);
55   EXPECT_THROW(sibling(1), std::invalid_argument);
56 
57   EXPECT_EQ(3, sibling(2));
58   EXPECT_EQ(2, sibling(3));
59 
60   EXPECT_EQ(5, sibling(4));
61   EXPECT_EQ(4, sibling(5));
62 
63   EXPECT_EQ(7, sibling(6));
64   EXPECT_EQ(6, sibling(7));
65 
66   EXPECT_EQ(9, sibling(8));
67   EXPECT_EQ(8, sibling(9));
68 
69   EXPECT_EQ(11, sibling(10));
70   EXPECT_EQ(10, sibling(11));
71 
72   EXPECT_EQ(13, sibling(12));
73   EXPECT_EQ(12, sibling(13));
74 
75   EXPECT_EQ(15, sibling(14));
76   EXPECT_EQ(14, sibling(15));
77 }
78 
79 TEST(NextPowerOfTwo, badInput)
80 {
81   EXPECT_THROW(nextPowerOfTwo(0), std::invalid_argument);
82 }
83 
84 TEST(NextPowerOfTwo, overflow)
85 {
86   EXPECT_THROW(nextPowerOfTwo(
87                    std::numeric_limits<size_t>::max()),
88                std::overflow_error);
89 
90   EXPECT_THROW(nextPowerOfTwo(
91                   std::numeric_limits<size_t>::max() / 2 + 2),
92                std::overflow_error);
93 
94   EXPECT_NO_THROW(nextPowerOfTwo(
95                      std::numeric_limits<size_t>::max() / 2 + 1));
96 
97   EXPECT_NO_THROW(nextPowerOfTwo(
98                      std::numeric_limits<size_t>::max() / 2));
99 }
```

```
100
101 TEST(NextPowerOfTwo, powersOfTwo)
102 {
103   EXPECT_EQ(1, nextPowerOfTwo(1));
104   EXPECT_EQ(2, nextPowerOfTwo(2));
105   EXPECT_EQ(4, nextPowerOfTwo(4));
106   EXPECT_EQ(8, nextPowerOfTwo(8));
107   EXPECT_EQ(16, nextPowerOfTwo(16));
108   EXPECT_EQ(32, nextPowerOfTwo(32));
109
110   EXPECT_EQ(1u<<10, nextPowerOfTwo(1u<<10));
111   EXPECT_EQ(1u<<20, nextPowerOfTwo(1u<<20));
112   EXPECT_EQ(1u<<30, nextPowerOfTwo(1u<<30));
113 }
114
115 TEST(NextPowerOfTwo, powersMinusOne)
116 {
117   EXPECT_EQ(4, nextPowerOfTwo(3));
118   EXPECT_EQ(8, nextPowerOfTwo(7));
119   EXPECT_EQ(16, nextPowerOfTwo(15));
120   EXPECT_EQ(32, nextPowerOfTwo(31));
121
122   EXPECT_EQ(1u<<10, nextPowerOfTwo((1u<<10) - 1));
123   EXPECT_EQ(1u<<20, nextPowerOfTwo((1u<<20) - 1));
124   EXPECT_EQ(1u<<30, nextPowerOfTwo((1u<<30) - 1));
125 }
126
127 TEST(NextPowerOfTwo, powersPlusOne)
128 {
129   EXPECT_EQ(4, nextPowerOfTwo(3));
130   EXPECT_EQ(8, nextPowerOfTwo(5));
131   EXPECT_EQ(16, nextPowerOfTwo(9));
132   EXPECT_EQ(32, nextPowerOfTwo(17));
133   EXPECT_EQ(64, nextPowerOfTwo(33));
134
135   EXPECT_EQ(1u<<11, nextPowerOfTwo((1u<<10) + 1));
136   EXPECT_EQ(1u<<21, nextPowerOfTwo((1u<<20) + 1));
137   // this must be unsigned!
138   EXPECT_EQ(1u<<31, nextPowerOfTwo((1u<<30) + 1));
139 }
140
141 TEST(NextPowerOfTwo, random)
142 {
143   unsigned int testdata[][2] = {
144     {804334813, 1073741824},
145     {7888, 8192},
146     {327463, 524288},
147     {436, 512},
148     {4706125, 8388608},
149     {13721806, 16777216},
150     {0, 0}
151   };
152   for (int i = 0; testdata[i][0] != 0; i++) {
153     EXPECT_EQ(testdata[i][1], nextPowerOfTwo(testdata[i][0]));
154   }
155 }
156
157 }
```

Bibliography

[Bar04] C. Barnes. Integer factorization algorithms. *Oregon State University*, 2004.

[BH77] Carter Bays and Richard H. Hudson. The segmented sieves of eratosthenes and primes in arithmetic progressions to 10^{12}. *BIT NUMERICAL MATHEMATICS*, 17(2):121–127, 1977.

[BK03] Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. *COMBINATORIAL PATTERN MATCHING, Lecture Notes in Computer Science*, 2676/2003:55–69, 2003.

[c9905] Iso/iec 9899:tc2 (c99 standard), May 6 2005.

[Daw06] Bruce Dawson. Comparing floating point numbers. `http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm`, 2004-2006. Online accessed 11.1.2010.

[Dea11] Jeff Dean. Building software systems at google and lessons learned. `http://www.stanford.edu/class/ee380/Abstracts/101110-slides.pdf`, November 2011. Online accessed 13.4.2011.

[DJ97] Ellard Daniel J. S-q course book. `http://www.eecs.harvard.edu/~ellard/Q-97/HTML/root/node43.html`, July 1997. Online accessed 10.4.2011.

[Duv83] Jean Pierre Duval. Factorizing words over an ordered alphabet. *Journal of Algorithms*, 4(issue 4):363–381, December 1983.

[Fen94] Peter M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24:327–336, 1994.

[Goo] Google. Google code style guide. `http://code.google.com/p/google-styleguide/`.

[GS05] Brian J. Gough and Richard M. Stallman. An introduction to gcc - for the gnu compilers gcc and g++. `http://www.network-theory.co.uk/docs/gccintro/gccintro_70.html`, August 2005. Online accessed, 11.1.2010.

[Hig93] NJ Higham. The accuracy of floating point summation. *SIAM Journal on Scientific Computing*, 1993.

[Hir75] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(Issue 6):341–343, June 1975.

[iee85] Ieee standard for binary floating-point arithmetic, 1985.

[Int04] Intel. Fdiv replacement program. statistical analysis of floating point flaw: Intel white paper. `http://www.intel.com/support/processors/pentium/sb/CS-013007.htm`, Jul 2004. Online accessed 11.4.2011.

[Jae93] Gerhard Jaeschke. On strong pseudoprimes to several bases. *mathematics of computation*, 61:915–926, October 1993.

[KMJP77] D.E. Knuth, J.H. Morris Jr, and V.R. Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6:323, 1977.

[Koc96] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. *ADVANCES IN CRYPTOLOGY — CRYPTO '96, Lecture Notes in Computer Science*, 1109/1996:104–113, 1996.

[KR87] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31:249–260, March 1987.

[KSB06] Juha Käarkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6), November 2006.

[LeB08] David LeBlanc. Ptrdiff_t is evil. `http://blogs.msdn.com/b/david_leblanc/archive/2008/09/02/ptrdiff-t-is-evil.aspx`, September 2008. Online accessed 19.1.2011.

[LIO96] J. L. LIONS. Ariane 5 flight 501 failure. Technical report, 1996. Online accessed 26.2.2010 `http://www.di.unito.it/~damiani/ariane5rep.html`.

[LKL+06] Gad Landau, Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. 2089:181–192, 2006. 10.1007/3-540-48194-X_17.

[Man04] Giovanni Manzini. Two space saving tricks for linear time lcp array computation. 3111:372–383, 2004. 10.1007/978-3-540-27810-8_32.

[mem] Soft errors in electronic memory – a white paper. `http://www.tezzaron.com/about/papers/soft_errors_1_1_secure.pdf`.

[Mic] Microsoft. Safeint library. `http://msdn.microsoft.com/en-us/library/dd570023.aspx`.

[MM90] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *SODA '90 Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, 1990.

[Mon08] David Monniaux. The pitfalls of verifying floating-point computation. *ACM Transactions on Programming Languages and Systems*, 30, 2008.

[Mor10] Yuta Mori. Sais - an implementation of the induced sorting algorithm. `http://sites.google.com/site/yuta256/sais`, 2008-2010. Online accessed 4.12.2010.

[NW09] Michał Nowak and Dawid Weiss. jsuffixarrays: Suffix arrays for java. `http://labs.carrotsearch.com/jsuffixarrays.html`, 2008-2009. Online accessed 17.2.2011.

[O'N08] Melissa E. O'Neill. The genuine sieve of eratosthenes. *Journal of Functional Programming*, October 2008. Published online by Cambridge University Press 9 `http://www.cs.hmc.edu/~oneill/papers/Sieve-JFP.pdf`.

[Pol75] J. M. Pollard. A monte carlo method for factorization. *BIT Numerical Mathematics*, 15:331–334, 1975. 10.1007/BF01933667.

[PSSSW80] Carl Pomerance, J. L. Selfridge, and Jr Samuel S. Wagstaff. The pseudoprimes to $25 * 10^9$. *Mathematics of Computation*, 35(151):1003–1026, July 1980.

[Sor92] Jonathan Sorenson. An introduction to prime number sieves. Technical report, University of Winsconsin-Madison, 1992.

[Wik11] Wikipedia. 64-bit — wikipedia, the free encyclopedia, 2011. [Online; accessed 18-April-2011].

[ZT03] Zhenxiang Zhang and Min Tang. Finding strong pseudoprimes to several bases. ii. *Math. Comp.*, 72:2085–2097, 2003.