Computing convex hulls in the plane: Collected algorithms in C++

Ask Neve Gamby[1] and Jyrki Katajainen[1,2]

[1] *Department of Computer Science, University of Copenhagen, Universitetsparken 5, 2100 Copenhagen East, Denmark*
[2] *Jyrki Katajainen and Company, 3390 Hundested, Denmark;*
jyrki@di.ku.dk

Abstract. In the two-dimensional convex-hull problem we are given a multiset S of points and the task is find those points of S that are the vertices of the minimum-area convex polygon enclosing all the points of S. In the output the vertices must be reported in the order in which they appear along the boundary of the polygon—either in clockwise or counterclockwise order. Further, when the coordinates of the points are integers, the computed output should always be correct.

This collection contains our implementations of the following convex-hull algorithms: PLANE-SWEEP, DIVIDE & CONQUER, QUICKHULL, BUCKETING, THROW-AWAY, POLES-FIRST, and TORCH. For an input of size n, none of our implementations use more than $O(\sqrt{n})$ extra words of memory and all run in linear expected time when the input points are randomly distributed.

As far as we know, this collection contains the best known algorithms—both with respect to time and space usage. If you know a better algorithm—or a better implementation written in C++—and think that it should be in this collection, let us know. Also remember that we promise a reward of 2.56 brownie points for every person who is the first to report an error in one of the programs released in this collection.

Keywords. Computational geometry, convex hull, algorithm collection, quality of implementation, robustness

CPH STL Report 2018-1, February 2018. Revised March 2018

Contents

1. Introduction

The problem of computing the convex hull for a multiset of points in the Euclidean plane is well studied in computational geometry. One may think that this problem is easy and solved, but many of the descriptions and programs one can find from the Internet are of low quality. Actually, it is a non-trivial task to write a program that handles all the special cases correctly. In particular, we want to solve the problem without the usual simplifying assumptions about the input: there can be duplicates among the input and the points need not be in a general position.

For a proper introduction to the topic, we refer to the textbook by Cormen et al. [2, Section 33.3]. This book makes good job in explaining Graham's ROTATIONAL-SWEEP algorithm, but this algorithm is not very efficient in practice. We have meticulously analysed several algorithms (see [3, 4, 5, 6]). Based on this work, we have selected the best algorithms known by us to this collection. We have tried to implement each algorithm in the best possible way. But, unfortunately, we have to admit that we have made many embarrassing errors on the journey. The implementations you find here are known to be fast. Some of them can even be implemented space-optimally if needed. But there is certainly still room for improvements. Do not hesitate to contact us, if you have any constructive feedback.

Our hope is that people start to use these programs as baselines in their future studies. If you can implement an algorithm that beats all the algorithms in this collection, we can just congratulate you. In this case, it is quite probable that you have something that could be published.

Recall that a point q of a convex set X is called an *extreme point* if no two other points p and r exist in X such that q lies on the line segment $\overline{pr}$. Hence, the vertices of the convex polygon P describing the convex hull are precisely the extreme points of the smallest convex set enclosing the points of S.

Let us assume that the input points are given in a sequence, i.e. in a `std::vector`. An *in-place* convex-hull algorithm (see, for example, [1]) partitions this sequence into two parts: (1) the first part contains all the extreme points in clockwise or counterclockwise order of their appearance on P and (2) the second part contains all the remaining points that are inside P or on the periphery of P.

All the algorithms in this collection rearrange the points in the input sequence in place. In addition to this side effect, the algorithms return an iterator to the first interior point. Thus, for two iterators `i` and `k`, if the range $[\![\texttt{i}\,..\,\texttt{k}]\!)$ specifies the input and `j` is the returned output, the range $[\![\texttt{i}\,..\,\texttt{j}]\!)$ contains the extreme points in circular order and the range $[\![\texttt{j}\,..\,\texttt{k}]\!)$ contains the interior points eliminated during the computation.

Each algorithm is implemented in its own namespace which provides the function `solve` that can be used to compute the convex hull for a given sequence, and the function `check` that can be used to verify the correctness of the corresponding solver. If `I` is the iterator type used by the input

sequence, the signatures of these two functions are as follows:

```
template<typename I>
using solver = I (*)(I, I);

template<typename I>
using checker = bool (*)(I, I);
```

That is, the signature of an in-place solver is similar to that of the generic function `std::partition` in the C++ standard library.

To summarize, a *solver* takes two iterators specifying the first and the past-the-end positions of the input sequence and, after reordering the points, the return value specifies the end of the computed convex hull and the beginning of all interior points in the very same sequence. Even if the users think that the computation is done in place, the solvers may use some temporary storage to speed up the computation.

A *checker* should have no side effects. Therefore, it takes a copy of the input, solves the problem with a solver, and uses the tools in our test framework to check that we still have the same points in the output, that the output is actually a convex polygon, and that all the input points are inside or on the boundary of the computed convex polygon.

2. Simple use case

Consider now a simple use case where we want to compute the convex hull for a multiset of four points (0, 0), (0, 1), (0, 2), and (0, 1). Let us use the PLANE-SWEEP algorithm to do the computation.

```
#include <cassert> // assert macro
#include <iostream> // std streams
#include "plane_sweep.h++" // plane_sweep::solve
#include "point.h++" // point
#include <vector> // std::vector

int main() {
  using P = point<int>;
  using S = std::vector<P>;
  using I = typename S::iterator;

  S bag{P(0, 0), P(0, 1), P(0, 2), P(0, 1)};

  I rest = plane_sweep::solve(bag.begin(), bag.end());
  auto h = rest - bag.begin();
  assert(h == 2);
  for (I i = bag.begin(); i != rest; ++i) {
    std::cout << *i << " ";
  }
  std::cout << "\n";
}
```

As shown below, when this program was run on a terminal, it printed out two points: (0, 0) (0, 2). In this case, a checker would have accepted the output even if the extreme points were reported in opposite order.

```
shell> g++ -O3 -std=c++17 -x c++ -Wall -Wextra use-case.c++
shell> ./a.out
(0, 0) (0, 2)
```

If you want to perform the experiments with the multi-precision integers, you have to install the CPH STL integers [8] and the CPH MPL [7]. Since both of these library components rely on concepts, you should compile the code with the option `-fconcepts`. We had these tools in the parent directory, so we also had to use the compiler option `-I..`.

3. Use of the `makefile`

You can use `make` to redo the experiments done in the papers [4, 5, 6]. For each convex-hull algorithm X, the `makefile` provides the following facilities. For the sake of concreteness, we fix X to be QUICKHULL.

`quickhull.check`. Run a unittest to check that the code compiles.

`quickhull.test`. Run the program through all our test cases.

`quickhull.square`. Run the CPU benchmark for the square data set.

`quickhull.disc`. Run the CPU benchmark for the disc data set.

`quickhull.bell`. Run the CPU benchmark for the bell data set—i.e. the points are drawn according to a discrete normal distribution.

`quickhull.sorted`. Run the CPU benchmark for a presorted data set—i.e. the points are given in sorted order according to their x-coordinates.

`quickhull.universe`. Run the CPU benchmark for the saturated-universe data set.

`quickhull.special`. Run the CPU benchmark for the special data set where there are four poles and many duplicates in the centre.

`quickhull.parabola`. Run the CPU benchmark for the parabola data set.

`quickhull.turn`. Measure how many times an orientation test is called.

`quickhull.comp`. Measure how many coordinate comparisons are performed.

`quickhull.move`. Measure how many coordinate moves are performed.

`quickhull.benchmark`. Redo all the experiments for QUICKHULL in one go. For algorithm X, the results will be put on the log file `x.log`.

If you want to run the benchmarks for the largest data set ($n = 2^{30}$), you have to enable it in the `makefile`. Each of these huge experiments took about 10 minutes in our Linux computer.

As an example, let us run the PLANE-SWEEP algorithm for the square data set:

```
shell> make plane_sweep.square
g++ -O3 -std=c++17 -x c++ -Wall -Wextra -fconcepts -DNDEBUG -I.. -
    ↪ DNAME=plane_sweep driver.c++
1024     55.1
```

```
32768    52.58
1048576 57.52
33554432        80.37
```

The same `makefile` can also handle the micro-benchmarks since these are packaged in the same way as the convex-hull algorithms. Each micro-benchmark is defined its own namespace—the name of this namespace is the same as the name of the file—and the driver will run the function `solve` in this namespace. To give an example, let us see how fast INTROSORT can sort a sorted sequence of points according to their x-coordinates:

```
shell> make sort.sorted
g++ -O3 -std=c++17 -x c++ -Wall -Wextra -fconcepts -DNDEBUG -I.. -
    ↪ DSORTED -DNAME=sort driver.c++
1024     6.718
32768    9.787
1048576 13.44
33554432        18.41
```

It is amazing that, even in this case, BUCKETSORT can speed things up (but TWO-PHASE BUCKETSORT makes things slower—try it yourself). In the PLANE-SWEEP algorithm, TWO-PHASE BUCKETSORT is used by default, but you can change this by disabling the macro definition `TWO_PHASE_BUCKETSORT`.

```
shell> make bsort.sorted
g++ -O3 -std=c++17 -x c++ -Wall -Wextra -fconcepts -DNDEBUG -I.. -
    ↪ DSORTED -DNAME=bsort driver.c++
1024     7.204
32768    8.488
1048576 8.341
33554432        12.65
```

A. Source code

A.1 Copyright notice

A.2 Release date

2018-03-08

B. Convex-hull algorithms

B.1 plane_sweep.h++

```
/*
  Performance Engineering Laboratory © 2017–2018
*/

#ifndef __PLANE_SWEEP__
#define __PLANE_SWEEP__

#include <algorithm> // std::sort std::partition ...
#include <cassert> // assert macro
#include <cstdlib> // std::size_t
#include <functional> // std::less std::greater
#include <iterator> // std::iterator_traits
#include "point.h++" // left_turn right_turn
#include "two_phase_bucketsort.h++" // two_phase_bucketsort::sort
#include <utility> // std::pair std::get ...
#include "validation.h++" // same_multiset convex_polygon all_inside

//#define TWO_PHASE_BUCKETSORT

namespace plane_sweep {

  template<typename variable>
  void ignore_warning(variable) {
  }

  // move *st <- *rd and *nd <- *th by swaps in parallel

  template<typename I>
  void parallel_iter_swap(I st, I nd, I rd, I th) {
    assert(st != nd and rd != th);
    std::iter_swap(st, rd);
    if (th == st) {
      std::iter_swap(nd, rd);
    }
    else {
      std::iter_swap(nd, th);
    }
  }

  template<typename I>
  std::pair<I, I> find_poles(I first, I past) {
    using P = typename std::iterator_traits<I>::value_type;
    auto pair = std::minmax_element(first, past,
      [](P const& a, P const& b) -> bool {
        return (a.x < b.x) or (a.x == b.x and a.y < b.y);
      });
    return pair;
  }
```

```
49
50  template<typename I>
51  I partition_upper_lower(I pole, I antipole, I first, I past) {
52    using P = typename std::iterator_traits<I>::value_type;
53    using T = typename P::coordinate;
54    T upper_limit = std::max((*pole).y, (*antipole).y);
55    T lower_limit = std::min((*pole).y, (*antipole).y);
56    I i = std::partition(first, past,
57      [&](P const& q) -> bool {
58        if (q.y ≥ upper_limit) {
59          return true;
60        }
61        else if (q.y ≤ lower_limit) {
62          return false;
63        }
64        return not left_turn(*pole, q, *antipole);
65      });
66    return i;
67  }
68
69  template<typename I, typename C>
70  std::pair<I, I> clean(I pole, I past, C compare) {
71    assert(pole ≠ past);
72    using P = typename std::iterator_traits<I>::value_type;
73    I k = pole + 1;
74    while (k ≠ past and (*k).x == (*pole).x) {
75      ++k;
76    }
77    if (k == pole + 1) {
78      return std::make_pair(pole, pole + 1); // (top, next)
79    }
80    I top = std::max_element(pole + 1, k,
81      [&](P const& p, P const& q) -> bool {
82        return compare(p.y, q.y);
83      });
84    if ((*top).y == (*pole).y) {
85      return std::make_pair(pole, k);
86    }
87    std::iter_swap(pole + 1, top);
88    return std::make_pair(pole + 1, k);
89  }
90
91  template<typename I>
92  I swap_blocks(I source, I past_the_end, I target) {
93    if (source == target or source == past_the_end) {
94      return past_the_end;
95    }
96    using P = typename std::iterator_traits<I>::value_type;
97    P p = *target;
98    I const last = past_the_end - 1;
99    while (true) {
100     *target = *source;
```

```
101         ++target;
102         if (source == last) {
103           break;
104         }
105         *source = *target;
106         ++source;
107       }
108       *source = p;
109       return target;
110     }
111
112     template<typename I, typename C, typename K>
113     void sort(I first, I past, C compare, K key) {
114
115   #ifdef TWO_PHASE_BUCKETSORT
116
117       two_phase_bucketsort::sort(first, past, compare, key);
118
119   #else
120
121       ignore_warning(key);
122       std::sort(first, past, compare);
123
124   #endif
125
126     }
127
128     template<typename I>
129     I scan(I first, I top, I next, I past) {
130       assert(first != past);
131       for (I i = next; i != past; ++i) {
132         while (top != first and not right_turn(*(top - 1), *top, *i))
      ↪ {
133           --top;
134         }
135         ++top;
136         std::iter_swap(i, top);
137       }
138       return ++top;
139     }
140
141     template<typename I>
142     I scan_one_more(I first, I top, I extra) {
143       while (top != first and not right_turn(*(top - 1), *top,
      ↪ *extra)) {
144         --top;
145       }
146       return ++top;
147     }
148
149     template<typename I, typename C>
150     I chain(I pole, I rest, I antipole, C compare) {
```

```
151     using P = typename std::iterator_traits<I>::value_type;
152     using T = typename P::coordinate;
153     if (std::distance(pole, rest) == 1) {
154       return rest;
155     }
156     sort(pole + 1, rest,
157       [&](P const& p, P const& q) → bool {
158         return compare(p.x, q.x);
159       },
160       [](P const& p) → T {
161         return p.x;
162       });
163     std::pair<I, I> pair = clean(pole, rest, compare);
164     I top = std::get<0>(pair);
165     I next = std::get<1>(pair);
166     I interior = scan(pole, top, next, rest);
167     interior = scan_one_more(pole, interior − 1, antipole);
168     return interior;
169   }
170
171   template<typename I>
172   I solve(I first, I past) {
173     using P = typename std::iterator_traits<I>::value_type;
174     using T = typename P::coordinate;
175     std::size_t n = past − first;
176     if (n < 2) {
177       return past;
178     }
179     std::pair<I, I> pair = find_poles(first, past);
180     I west = first;
181     I east = past − 1;
182     parallel_iter_swap(west, east, std::get<0>(pair),
      ↪ std::get<1>(pair));
183     if (*east == *west) {
184       return first + 1;
185     }
186     I middle = partition_upper_lower(west, east, first + 1, past −
      ↪ 1);
187     I rest1 = chain(west, middle, east, std::less<T>());
188     std::iter_swap(middle, east);
189     east = middle;
190     I rest2 = chain(east, past, west, std::greater<T>());
191     I interior = swap_blocks(east, rest2, rest1);
192     return interior;
193   }
194
195   template<typename I>
196   bool check(I first, I past) {
197     using P = typename std::iterator_traits<I>::value_type;
198     using S = std::vector<P>;
199     using J = typename S::iterator;
200     S data;
```

```
201       std::size_t n = past - first;
202       data.resize(n);
203       std::copy(first, past, data.begin());
204       J rest = solve(data.begin(), data.end());
205       bool ok = validation::same_multiset(data.begin(), data.end(),
          ↪ first, past) and validation::convex_polygon(data.begin(),
          ↪ rest) and validation::all_inside(rest, data.end(),
          ↪ data.begin(), rest);
206       return ok;
207     }
208   }
209
210   #endif
```

B.2 divide_and_conquer.h++

```
1   /*
2     Performance Engineering Laboratory © 2018
3   */
4
5   #ifndef __DIVIDE_AND_CONQUER__
6   #define __DIVIDE_AND_CONQUER__
7
8   #include <algorithm> // std::sort std::partition ...
9   #include <cassert> // assert macro
10  #include <cstdlib> // std::size_t
11  #include <iterator> // std::iterator_traits
12  #include "point.h++" // left_turn right_turn
13  #include <utility> // std::pair std::get ...
14  #include "validation.h++" // same_multiset convex_polygon all_inside
15
16  namespace divide_and_conquer {
17
18    // move *st <- *rd and *nd <- *th by swaps in parallel
19
20    template<typename I>
21    void parallel_iter_swap(I st, I nd, I rd, I th) {
22      assert(st ≠ nd and rd ≠ th);
23      std::iter_swap(st, rd);
24      if (th == st) {
25        std::iter_swap(nd, rd);
26      }
27      else {
28        std::iter_swap(nd, th);
29      }
30    }
31
32    template<typename I>
33    std::pair<I, I> find_poles(I first, I past) {
34      using P = typename std::iterator_traits<I>::value_type;
35      auto pair = std::minmax_element(first, past,
```

```
36        [](P const& a, P const& b) -> bool {
37          return (a.x < b.x) or (a.x == b.x and a.y < b.y);
38        });
39      return pair;
40    }
41
42    template<typename I>
43    I partition_upper_lower(I first, I past) {
44      assert(first != past);
45      using P = typename std::iterator_traits<I>::value_type;
46      using T = typename P::coordinate;
47      std::pair<I, I> pair = find_poles(first, past);
48      I west = first;
49      I east = past - 1;
50      parallel_iter_swap(west, east, std::get<0>(pair),
        ↪ std::get<1>(pair));
51
52      T upper_limit = std::max((*west).y, (*east).y);
53      T lower_limit = std::min((*west).y, (*east).y);
54      I i = std::partition(west + 1, east,
55        [&](P const& q) -> bool {
56          if (q.y ≥ upper_limit) {
57            return true;
58          }
59          else if (q.y ≤ lower_limit) {
60            return false;
61          }
62          return not left_turn(*west, q, *east);
63        });
64      std::iter_swap(i, east);
65      return i + 1;
66    }
67
68    template<typename I, typename C>
69    std::pair<I, I> clean(I pole, I rest, C compare) {
70      assert(pole != rest);
71      I k = pole + 1;
72      while (k != rest and (*k).x == (*pole).x) {
73        ++k;
74      }
75      if (k == pole + 1) {
76        return std::make_pair(pole, pole + 1); // (top, next)
77      }
78      I bottom = std::min_element(pole, k, compare);
79      std::iter_swap(pole, bottom);
80      I top = std::max_element(pole + 1, k, compare);
81      if ((*top).y == (*pole).y) {
82        return std::make_pair(pole, k);
83      }
84      std::iter_swap(pole + 1, top);
85      return std::make_pair(pole + 1, k);
86    }
```

```
87
88    template<typename I>
89    void swap_blocks(I source, I past_the_end, I target) {
90      // retains the order in [source, past_the_end)
91      if (source == target or source == past_the_end) {
92        return;
93      }
94      using P = typename std::iterator_traits<I>::value_type;
95      I hole = target;
96      P p = *target;
97      I const last = past_the_end - 1;
98      while (true) {
99        *hole = *source;
100       ++hole;
101       if (source == last) {
102         break;
103       }
104       *source = *hole;
105       ++source;
106     }
107     *source = p;
108   }
109
110   template<typename I>
111   I scan(I first, I top, I next, I past) {
112     assert(first != past);
113     for (I i = next; i != past; ++i) {
114       while (top != first and not right_turn(*(top - 1), *top, *i))
      ↪ {
115         --top;
116       }
117       ++top;
118       std::iter_swap(i, top);
119     }
120     return ++top;
121   }
122
123   template<typename I>
124   I scan_one_more(I first, I top, I extra) {
125     while (top != first and not right_turn(*(top - 1), *top,
      ↪ *extra)) {
126       --top;
127     }
128     return ++top;
129   }
130
131   template<typename I, typename C>
132   I brute_force(I first, I past, C compare) {
133     std::size_t n = past - first;
134     assert(n ≤ 4);
135     if (n ≤ 1) {
136       return past;
```

```
137     }
138     else if (n == 2) {
139       if (*first == *(first + 1)) {
140         return first + 1;
141       }
142       if (not compare(*first, *(first + 1))) {
143         std::iter_swap(first, first + 1);
144       }
145       return past;
146     }
147     if (n == 3) {
148       I west = std::min_element(first, past, compare);
149       std::iter_swap(first, west);
150       I east = std::max_element(first + 1, past, compare);
151       if (*first == *east) {
152         return first + 1;
153       }
154       std::iter_swap(first + 2, east);
155       if (right_turn(*first, *(first + 1), *(first + 2))) {
156         return first + 3;
157       }
158       std::iter_swap(first + 1, first + 2);
159       return first + 2;
160     }
161     // n == 4
162     I west = std::min_element(first, past, compare);
163     std::iter_swap(first, west);
164     I east = std::max_element(first + 1, past, compare);
165     if (*first == *east) {
166       return first + 1;
167     }
168     std::iter_swap(first + 3, east);
169     if (not compare(*(first + 1), *(first + 2))) {
170       std::iter_swap(first + 1, first + 2);
171     }
172     I rest = scan(first, first, first + 1, past);
173     return rest;
174   }
175
176   template<typename I, typename C>
177   I recurse(I first, I past, C compare) {
178     std::size_t n = past - first;
179     if (n ≤ 4) {
180       return brute_force(first, past, compare);
181     }
182     I middle = first + n / 2;
183     I rest1 = recurse(first, middle, compare);
184     I rest2 = recurse(middle, past, compare);
185     std::size_t h2 = rest2 - middle;
186     I rest = rest1 + h2;
187     swap_blocks(middle, rest2, rest1);
188     std::inplace_merge(first, rest1, rest, compare);
```

```
189     std::pair<I, I> pair = clean(first, rest, compare);
190     I top = std::get<0>(pair);
191     I next = std::get<1>(pair);
192     rest = scan(first, top, next, rest);
193     return rest;
194   }
195
196   template<typename I>
197   I solve(I first, I past) {
198     using P = typename std::iterator_traits<I>::value_type;
199     std::size_t n = past - first;
200     if (n < 2) {
201       return past;
202     }
203     I middle = partition_upper_lower(first, past);
204     I rest_1 = recurse(first, middle,
205       [](P const& a, P const& b) {
206         return a.x < b.x or (a.x == b.x and a.y < b.y);
207       });
208     if (middle == past) {
209       return rest_1;
210     }
211     I west = first;
212     std::iter_swap(rest_1 - 1, middle - 1);
213     I east = middle - 1;
214     I rest_2 = recurse(east, past,
215       [](P const& a, P const& b) {
216         return a.x > b.x or (a.x == b.x and a.y > b.y);
217       });
218     if (rest_2 != east) {
219       rest_2 = scan_one_more(east, rest_2 - 1, west);
220     }
221     std::size_t h_2 = rest_2 - east;
222     I rest = rest_1 + (h_2 - 1);
223     swap_blocks(east, rest_2, rest_1 - 1);
224     return rest;
225   }
226
227   template<typename I>
228   bool check(I first, I past) {
229     using P = typename std::iterator_traits<I>::value_type;
230     using S = std::vector<P>;
231     using J = typename S::iterator;
232     S data;
233     std::size_t n = past - first;
234     data.resize(n);
235     std::copy(first, past, data.begin());
236     J rest = solve(data.begin(), data.end());
237     bool ok = validation::same_multiset(data.begin(), data.end(),
      ↪ first, past) and validation::convex_polygon(data.begin(),
      ↪ rest) and validation::all_inside(rest, data.end(),
      ↪ data.begin(), rest);
```

```
238       return ok;
239     }
240   }
241
242   #endif
```

B.3 quickhull.h++

```
1   /*
2     Performance Engineering Laboratory © 2017–2018
3   */
4
5   #ifndef __QUICKHULL__
6   #define __QUICKHULL__
7
8   #include "point.h++" // signed_area left_turn
9
10  #include <algorithm> // std::minmax_element std::partitioning ...
11  #include <cassert> // assert macro
12  #include <cstdlib> // std::size_t
13  #include <iterator> // std::iterator_traits
14  #include <utility> // std::pair std::make_pair ...
15  #include "validation.h++" // same_multiset convex_polygon all_inside
16  #include <vector> // std::vector
17
18  namespace quickhull {
19
20    template <typename variable>
21    void ignore_warning(variable) {
22    }
23
24    // move *st <- *rd and *nd <- *th by swaps in parallel
25
26    template<typename I>
27    void parallel_iter_swap(I st, I nd, I rd, I th) {
28      assert(st != nd and rd != th);
29      std::iter_swap(st, rd);
30      if (th == st) {
31        std::iter_swap(nd, rd);
32      }
33      else {
34        std::iter_swap(nd, th);
35      }
36    }
37
38    template<typename I>
39    std::pair<I, I> find_poles(I first, I past) {
40      using P = typename std::iterator_traits<I>::value_type;
41      auto pair = std::minmax_element(first, past,
42        [](P const& a, P const& b) -> bool {
43          return (a.x < b.x) or (a.x == b.x and a.y < b.y);
```

```
44        });
45      return pair;
46    }
47
48    template<typename I>
49    I find_furthest(I first, I past, I antipole) {
50      assert(first ≠ past);
51      I pole = first;
52      I answer = pole;
53      using U = decltype(signed_area(*pole, *pole, *pole));
54      U best = 0;
55      if ((*antipole).x == (*pole).x) { // vertical
56        for (I i = first + 1; i ≠ past; ++i) {
57          U Φ = signed_area(*pole, *antipole, *i);
58          if (Φ > best or (Φ == best and (*i).y < (*answer).y)) {
59            answer = i;
60            best = Φ;
61          }
62        }
63      }
64      else {
65        for (I i = first + 1; i ≠ past; ++i) {
66          U Φ = signed_area(*pole, *antipole, *i);
67          if (Φ > best or (Φ == best and (*i).x < (*answer).x)) {
68            answer = i;
69            best = Φ;
70          }
71        }
72      }
73      return answer;
74    }
75
76    template<typename I>
77    I partition_left_right(I first, I past, I antipole) {
78      assert(first ≠ past);
79      using P = typename std::iterator_traits<I>::value_type;
80      I pole = first;
81      I middle = std::partition(pole + 1, past,
82        [&](P const& q) → bool {
83          return not left_turn(*pole, q, *antipole);
84        });
85      return middle;
86    }
87
88    template <typename I>
89    void swap_blocks(I source, I past_the_end, I target) {
90      if (source == target or source == past_the_end) {
91        return;
92      }
93      using P = typename std::iterator_traits<I>::value_type;
94      I hole = target;
95      P p = *target;
```

```
96      I const last = past_the_end - 1;
97      while (true) {
98        *hole = *source;
99        ++hole;
100       if (source == last) {
101         break;
102       }
103       *source = *hole;
104       ++source;
105     }
106     *source = p;
107   }
108
109   template <typename I>
110   void move_away(I here, I rest, I past) {
111     if (here == rest or rest == past) {
112       return;
113     }
114     if (rest - here < past - rest) {
115       swap_blocks(here, rest, past - (rest - here));
116     }
117     else {
118       swap_blocks(rest, past, here);
119     }
120   }
121
122   template<typename I>
123   I recurse(I pole, I past, I antipole) {
124     std::size_t n = std::distance(pole, past);
125     if (n == 1) {
126       return past;
127     }
128     if (n == 2) {
129       if (no_turn(*(pole + 1), *pole, *antipole)) {
130         return pole + 1;
131       }
132       else {
133         return past;
134       }
135     }
136     I pivot = find_furthest(pole, past, antipole);
137     if (no_turn(*pivot, *pole, *antipole)) {
138       return pole + 1;
139     }
140     I last = past - 1;
141     std::iter_swap(pivot, last); // pivot at the end
142     I mid = partition_left_right(pole, last, last);
143     I eliminated = recurse(pole, mid, last);
144     std::iter_swap(mid, last);
145     std::iter_swap(eliminated, mid); // pivot at its final place
146     pivot = eliminated;
147     std::size_t m = past - mid;
```

```
148       ++mid;
149       ++eliminated;
150       move_away(eliminated, mid, past);
151       I interior = partition_left_right(pivot, pivot + m, antipole);
152       eliminated = recurse(pivot, interior, antipole);
153       return eliminated;
154     }
155
156     template<typename I>
157     I solve(I first, I past) {
158       std::size_t n = past - first;
159       if (n < 2 or (n == 2 and *first ≠ *(first + 1))) {
160         return past;
161       }
162       std::pair<I, I> pair = find_poles(first, past);
163       I west = first;
164       I east = past - 1;
165       parallel_iter_swap(west, east, std::get<0>(pair),
          ↪ std::get<1>(pair));
166       if (*west == *east) {
167         return first + 1;
168       }
169       I middle = partition_left_right(west, east, east);
170       std::size_t m = past - middle;
171       I eliminated = recurse(first, middle, east);
172       std::iter_swap(middle, east);
173       std::iter_swap(eliminated, middle); // east at its final place
174       east = eliminated;
175       ++middle;
176       ++eliminated;
177       move_away(eliminated, middle, past);
178       eliminated = recurse(east, east + m, west); // downunder
179       return eliminated;
180     }
181
182     template<typename I>
183     bool check(I first, I past) {
184       using P = typename std::iterator_traits<I>::value_type;
185       using S = std::vector<P>;
186       using J = typename S::iterator;
187       S data;
188       std::size_t n = past - first;
189       data.resize(n);
190       std::copy(first, past, data.begin());
191       J rest = solve(data.begin(), data.end());
192       bool ok = validation::same_multiset(data.begin(), data.end(),
          ↪ first, past) and validation::convex_polygon(data.begin(),
          ↪ rest) and validation::all_inside(rest, data.end(),
          ↪ data.begin(), rest);
193       return ok;
194     }
195   }
```

```
196
197 #endif
```

B.4 bucketing.h++

```
1  /*
2    Performance Engineering Laboratory © 2017–2018
3  */
4
5  #ifndef __BUCKETING__
6  #define __BUCKETING__
7
8  #include <cmath> // std::sqrt std::ceil
9  #include <cstdlib> // std::size_t
10 #include <iterator> // std::iterator_traits
11 #include <limits> // std::numeric_limits
12 #include "plane_sweep.h++" // plane_sweep::solve
13 #include "point.h++"
14 #include <utility> // std::pair
15 #include <vector> // std::vector
16
17 namespace bucketing {
18
19   template<typename V>
20   class identity {
21   public:
22
23     V operator()(V const& e) const {
24       return e;
25     }
26
27     V& operator()(V& e) const {
28       return e;
29     }
30   };
31
32   template<typename D, typename V, typename K = identity<V>>
33   class formula {
34   public:
35
36     formula(D m, V const& min, V const& max, K key = K())
37       : a(double(m - 1) / (double(key(max)) - double(key(min)))),
38         b(double(m - 1) * double(key(min)) / (double(key(max)) -
    ↪ double(key(min)))),
39         key(key) {
40     }
41
42     D operator()(V const& x) {
43       return D(a * double(key(x)) - b);
44     }
45
```

```
46    private:
47
48      double a;
49      double b;
50      K key;
51    };
52
53    // move *st <- *rd and *nd <- *th by swaps in parallel
54
55    template<typename I>
56    void parallel_iter_swap(I st, I nd, I rd, I th) {
57      assert(st ≠ nd and rd ≠ th);
58      std::iter_swap(st, rd);
59      if (th == st) {
60        std::iter_swap(nd, rd);
61      }
62      else {
63        std::iter_swap(nd, th);
64      }
65    }
66
67    // all points on the same vertical line
68
69    template<typename I>
70    I vertical_case(I first, I past) {
71      assert(std::distance(first, past) ≥ 2);
72      using P = typename std::iterator_traits<I>::value_type;
73      std::pair<I, I> pair = std::minmax_element(first, past,
74        [](P const& p, P const& q) → bool {
75          return (p.y < q.y);
76        });
77      I south = std::get<0>(pair);
78      I north = std::get<1>(pair);
79      if ((*south).y == (*north).y) {
80        std::iter_swap(first, south);
81        return first + 1;
82      }
83      parallel_iter_swap(first, first + 1, south, north);
84      return first + 2;
85    }
86
87    template<typename I>
88    I prune(I first, I past) {
89      using P = typename std::iterator_traits<I>::value_type;
90      using T = typename P::coordinate;
91      using D = typename std::iterator_traits<I>::difference_type;
92      D n = std::distance(first, past);
93      if (n < 2) {
94        return past;
95      }
96
97      // find the extreme values in the ±x directions
```

```
98
99      std::pair<I, I> pair = std::minmax_element(first, past,
100       [](P const& p, P const& q) → bool {
101         return p.x < q.x;
102       });
103     I west = std::get<0>(pair);
104     I east = std::get<1>(pair);
105     if ((*west).x == (*east).x) {
106       return vertical_case(first, past);
107     }
108     D m = std::ceil(std::sqrt(n)); // divided by 2 made the program
        ↪ faster
109     formula slab(m, (*west).x, (*east).x);
110
111     T* slab_min = new T[m];
112     T* slab_max = new T[m];
113     for (D j = 0; j ≠ m; ++j) {
114       slab_max[j] = std::numeric_limits<T>::min();
115       slab_min[j] = std::numeric_limits<T>::max();
116     }
117
118     // calculate the slab extrema
119
120     for (I j = first; j ≠ past; ++j) {
121       D i = slab((*j).x);
122       if ((*j).y < slab_min[i]) {
123         slab_min[i] = (*j).y;
124       }
125       if ((*j).y > slab_max[i]) {
126         slab_max[i] = (*j).y;
127       }
128     }
129
130     // find the extreme values in the ±y directions
131
132     D bottom = 0;
133     D top = 0;
134     T ymin = std::numeric_limits<T>::max();
135     T ymax = std::numeric_limits<T>::min();
136     for (D i = 0; i ≠ m; ++i) {
137       if (slab_max[i] > ymax) {
138         ymax = slab_max[i];
139         top = i;
140       }
141       if (slab_min[i] < ymin) {
142         ymin = slab_min[i];
143         bottom = i;
144       }
145     }
146
147     // determine the boundaries for the extreme values
148
```

```
149     T roof = (*east).y; // Q1 go backwards upstairs
150     T previous = roof;
151     for (D j = m - 1; j >= top; --j) {
152       previous = roof;
153       roof = std::max(roof, slab_max[j]);
154       slab_max[j] = previous;
155     }
156
157     T floor = (*east).y; // Q2 go backwards downstairs
158     previous = floor;
159     for (D j = m - 1; j >= bottom; --j) {
160       previous = floor;
161       floor = std::min(floor, slab_min[j]);
162       slab_min[j] = previous;
163     }
164
165     floor = (*west).y; // Q3 go downstairs
166     previous = floor;
167     for (D j = 0; j < bottom; ++j) {
168       previous = floor;
169       floor = std::min(floor, slab_min[j]);
170       slab_min[j] = previous;
171     }
172     slab_min[bottom] = std::max(previous, slab_min[bottom]);
173
174     roof = (*west).y; // Q4 go upstairs
175     previous = roof;
176     for (D j = 0; j < top; ++j) {
177       previous = roof;
178       roof = std::max(roof, slab_max[j]);
179       slab_max[j] = previous;
180     }
181     slab_max[top] = std::min(previous, slab_max[top]);
182
183     // partition the input
184
185     I o = first;
186     for (I j = first; j != past; ++j) {
187       D column = slab((*j).x);
188       if ((*j).y >= slab_max[column] or (*j).y <= slab_min[column]) {
189         std::iter_swap(j, o);
190         ++o;
191       }
192     }
193     delete[] slab_min;
194     delete[] slab_max;
195     return o;
196   }
197
198   template<typename I>
199   I solve(I first, I past) {
200     I rest = prune(first, past);
```

```
201       I interior = plane_sweep::solve(first, rest);
202       return interior;
203     }
204
205     template<typename I>
206     bool check(I first, I past) {
207       using P = typename std::iterator_traits<I>::value_type;
208       using S = std::vector<P>;
209       using J = typename S::iterator;
210       S data;
211       std::size_t n = past - first;
212       data.resize(n);
213       std::copy(first, past, data.begin());
214       J rest = solve(data.begin(), data.end());
215       bool ok = validation::same_multiset(data.begin(), data.end(),
          ↪ first, past) and validation::convex_polygon(data.begin(),
          ↪ rest) and validation::all_inside(rest, data.end(),
          ↪ data.begin(), rest);
216       return ok;
217     }
218   }
219
220   #endif
```

B.5 throw_away.h++

```
1   /*
2     Performance Engineering Laboratory © 2017–2018
3
4     Find the extrema in eight directions and eliminate all points
5     inside the convex hull of these points.
6   */
7
8   #ifndef __THROW_AWAY__
9   #define __THROW_AWAY__
10
11  #include <algorithm> // std::sort std::min std::minmax_element
12  #include <cassert> // assert macro
13  #include <cmath> // std::sqrt
14  #include "cphmpl/functions.h++" // cphmpl::width
15  #include "cphstl/integers.h++" // cphstl::ℤ
16  #include <cstddef> // std::size_t
17  #include <iterator> // std::iterator_traits
18  #include <limits> // std::numeric_limits
19  #include "plane_sweep.h++"
20  #include "point.h++"
21  #include <utility> // std::pair std::get
22  #include "validation.h++" // same_multiset convex_polygon all_inside
23  #include <vector> // std::vector
24
25  namespace throw_away {
```

```
26
27    template<typename I>
28    std::vector<I> find_extrema(I first, I past) {
29      assert(first ≠ past);
30      using P = typename std::iterator_traits<I>::value_type;
31      using T = typename P::coordinate;
32  #if defined(MEASURE_MOVES) or defined(MEASURE_COMPS)
33  #define CONVERSION (int)
34  #else
35  #define CONVERSION
36  #endif
37      constexpr std::size_t w = cphmpl::width<T>;
38      using Z = cphstl::ℤ<w + 2>;
39      std::vector<I> max_position(8, first);
40      T xmin = (*first).x;
41      T xmax = (*first).x;
42      T ymin = (*first).y;
43      T ymax = (*first).y;
44      Z ne = Z(CONVERSION xmin) + Z(CONVERSION ymin);
45      Z se = Z(CONVERSION xmin) - Z(CONVERSION ymin);
46      Z sw = -(Z(CONVERSION xmin) + Z(CONVERSION ymin));
47      Z nw = Z(CONVERSION ymin) - Z(CONVERSION xmin);
48      for (I i = first; i ≠ past; ++i) {
49        T x = (*i).x;
50        T y = (*i).y;
51        if (x < xmin) {
52          xmin = x;
53          max_position[0] = i;
54        }
55        if (x > xmax) {
56          xmax = x;
57          max_position[1] = i;
58        }
59        if (y < ymin) {
60          ymin = y;
61          max_position[2] = i;
62        }
63        if (y > ymax) {
64          ymax = y;
65          max_position[3] = i;
66        }
67  #if defined(MEASURE_MOVES)
68
69        ++moves; ++moves; ++moves;
70
71  #endif
72        Z dot = Z(CONVERSION x) + Z(CONVERSION y);
73  #if defined(MEASURE_COMPS)
74
75        ++comps;
76
77  #endif
```

```
78        if (dot > ne) {
79  #if defined(MEASURE_MOVES)
80
81        ++moves;
82
83  #endif
84          ne = dot;
85          max_position[4] = i;
86        }
87  #if defined(MEASURE_MOVES)
88
89        ++moves; ++moves; ++moves;
90
91  #endif
92        dot = Z(CONVERSION x) - Z(CONVERSION y);
93  #if defined(MEASURE_COMPS)
94
95        ++comps;
96
97  #endif
98        if (dot > se) {
99  #if defined(MEASURE_MOVES)
100
101       ++moves;
102
103 #endif
104         se = dot;
105         max_position[5] = i;
106       }
107 #if defined(MEASURE_MOVES)
108
109       ++moves; ++moves; ++moves;
110
111 #endif
112       dot = -(Z(CONVERSION x) + Z(CONVERSION y));
113 #if defined(MEASURE_COMPS)
114
115       ++comps;
116
117 #endif
118       if (dot > sw) {
119 #if defined(MEASURE_MOVES)
120
121       ++moves;
122
123 #endif
124         sw = dot;
125         max_position[6] = i;
126       }
127 #if defined(MEASURE_MOVES)
128
129       ++moves; ++moves; ++moves;
```

```
130
131 #endif
132       dot = Z(CONVERSION y) - Z(CONVERSION x);
133 #if defined(MEASURE_COMPS)
134
135       ++comps;
136
137 #endif
138       if (dot > nw) {
139 #if defined(MEASURE_MOVES)
140
141       ++moves;
142
143 #endif
144         nw = dot;
145         max_position[7] = i;
146       }
147     }
148     return max_position;
149   }
150
151   template<typename S>
152   void remove_duplicates(S& sequence) {
153     assert(sequence.size() != 0);
154     assert(std::is_sorted(sequence.begin(), sequence.end()));
155     std::size_t i = 0;
156     std::size_t j = 1;
157     auto v = sequence[0];
158     while (j != sequence.size()) {
159       if (not (sequence[j] == v)) {
160         std::swap(sequence[i], v);
161         ++i;
162         v = sequence[j];
163       }
164       ++j;
165     }
166     std::swap(sequence[i], v);
167     ++i;
168     std::size_t leftovers = sequence.size() - i;
169     while (leftovers != 0) {
170       sequence.pop_back();
171       --leftovers;
172     }
173   }
174
175   template<typename P, typename I>
176   bool inside_convex_polygon(P const& p, I upper, I lower, I past) {
177     assert(past - upper > 2);
178     assert(*upper == *(past - 1));
179     assert(*upper < *(upper + 1));
180     assert(*upper < *(past - 2));
181     I i = upper;
```

```
182     do {
183       ++i;
184     } while (i != lower and *i < p);
185     if (left_turn(*(i - 1), *i, p)) {
186       return false;
187     }
188     I last = past - 1;
189     I j = lower;
190     do {
191       ++j;
192     } while (j != last and *j > p);
193     if (left_turn(*(j - 1), *j, p)) {
194       return false;
195     }
196     return true;
197   }
198
199   template<typename I>
200   I eliminate_inner_points(I first, I past, I polygon, I rest) {
201     assert(polygon != rest);
202     I i = first;
203     if (rest - polygon == 1) {
204       for (I j = first; j != past; ++j) {
205         if (not (*j == *polygon)) {
206           std::iter_swap(i, j);
207           ++i;
208         }
209       }
210       return i;
211     }
212     if (rest - polygon == 2) {
213       for (I j = first; j != past; ++j) {
214         if (not (on_line_segment(*polygon, *(polygon + 1), *j))) {
215           std::iter_swap(i, j);
216           ++i;
217         }
218       }
219       return i;
220     }
221     // rest - polygon > 2
222     using P = typename std::iterator_traits<I>::value_type;
223     std::vector<P> approximate_hull;
224     for (I k = polygon; k != rest; ++k) {
225       approximate_hull.push_back(*k);
226     }
227     approximate_hull.push_back(*polygon);
228     auto upper = approximate_hull.begin();
229     auto lower = upper;
230     auto end = approximate_hull.end();
231     for (auto k = upper + 1; k != end - 1; ++k) {
232       if (*lower < *k) {
233         lower = k;
```

```
234         }
235       }
236       for (I j = first; j != past; ++j) {
237         if (not inside_convex_polygon(*j, upper, lower, end)) {
238           std::iter_swap(i, j);
239           ++i;
240         }
241       }
242       return i;
243     }
244
245     template<typename I>
246     I prune(I first, I past) {
247       if (past - first < 2) {
248         return past;
249       }
250       // Step 1
251       std::vector<I> extrema = find_extrema(first, past);
252       std::sort(extrema.begin(), extrema.end());
253       remove_duplicates(extrema);
254       for (std::size_t i = 0; i != extrema.size(); ++i) {
255         std::iter_swap(first + i, extrema[i]);
256       }
257       // Step 2
258       I rest = plane_sweep::solve(first, first + extrema.size());
259       // Step 3
260       I interior = eliminate_inner_points(rest, past, first, rest);
261       return interior;
262     }
263
264     template<typename I>
265     I solve(I first, I past) {
266       I rest = prune(first, past);
267       return plane_sweep::solve(first, rest);
268     }
269
270     template<typename I>
271     bool check(I first, I past) {
272       using P = typename std::iterator_traits<I>::value_type;
273       using S = std::vector<P>;
274       using J = typename S::iterator;
275       S data;
276       std::size_t n = past - first;
277       data.resize(n);
278       std::copy(first, past, data.begin());
279       J rest = solve(data.begin(), data.end());
280       bool ok = validation::same_multiset(data.begin(), data.end(),
          ↪ first, past) and validation::convex_polygon(data.begin(),
          ↪ rest) and validation::all_inside(rest, data.end(),
          ↪ data.begin(), rest);
281       return ok;
282     }
```

```
283 }
284
285 #endif
```

B.6 torch.h++

```
1  /*
2    Performance Engineering Laboratory © 2017–2018
3  */
4
5  #ifndef __TORCH__
6  #define __TORCH__
7
8  #include <algorithm> // std::sort std::partition ...
9  #include <cassert> // assert macro
10 #include <cstdlib> // std::size_t
11 #include <iterator> // std::iterator_traits
12 #include "point.h++" // right_turn
13 #include <utility> // std::pair std::get
14 #include "validation.h++" // same_multiset convex_polygon all_inside
15
16 namespace torch {
17
18   // move *st <- *rd and *nd <- *th by swaps in parallel
19
20   template<typename I>
21   void parallel_iter_swap(I st, I nd, I rd, I th) {
22     assert(st != nd and rd != th);
23     std::iter_swap(st, rd);
24     if (th == st) {
25       std::iter_swap(nd, rd);
26     }
27     else {
28       std::iter_swap(nd, th);
29     }
30   }
31
32   template<typename I>
33   I go_upstairs(I first, I past) {
34     if (first == past) {
35       return past;
36     }
37     using P = typename std::iterator_traits<I>::value_type;
38     using T = typename P::coordinate;
39     T max = (*first).y;
40     I i = first + 1;
41     for (I j = first + 1; j != past; ++j) {
42       if ((*j).y >= max) {
43         max = (*j).y;
44         std::iter_swap(i, j);
45         ++i;
```

```
46        }
47      }
48      return i;
49    }
50
51    template<typename I>
52    I go_downstairs(I first, I past) {
53      if (first == past) {
54        return past;
55      }
56      using P = typename std::iterator_traits<I>::value_type;
57      using T = typename P::coordinate;
58      T min = (*first).y;
59      I i = first + 1;
60      for (I j = first + 1; j != past; ++j) {
61        if ((*j).y <= min) {
62          min = (*j).y;
63          std::iter_swap(i, j);
64          ++i;
65        }
66      }
67      return i;
68    }
69
70    template<typename I>
71    I scan(I first, I past) {
72      assert(first != past);
73      I top = first;
74      I next = first + 1;
75      for (I i = next; i != past; ++i) {
76        while (top != first and not right_turn(*(top - 1), *top, *i))
      ↪ {
77          --top;
78        }
79        ++top;
80        std::iter_swap(i, top);
81      }
82      return ++top;
83    }
84
85    template<typename I>
86    I scan_one_more(I first, I top, I extra) {
87      while (top != first and not right_turn(*(top - 1), *top,
      ↪ *extra)) {
88        --top;
89      }
90      return ++top;
91    }
92
93    template<typename I>
94    I solve(I first, I past) {
95      using P = typename std::iterator_traits<I>::value_type;
```

```
96      using T = typename P::coordinate;
97      std::size_t n = past - first;
98      if (n < 2) {
99        return past;
100     }
101
102     // find the west and east poles
103
104     std::pair<I, I> pair = std::minmax_element(first, past,
105       [](P const& a, P const& b) -> bool {
106         return (a.x < b.x) or (a.x == b.x and a.y < b.y);
107       });
108     I west = std::get<0>(pair);
109     I east = std::get<1>(pair);
110
111     // all points equal?
112
113     if (*west == *east) {
114       return first + 1;
115     }
116
117     // all points on a vertical line?
118
119     if ((*west).x == (*east).x) {
120       parallel_iter_swap(first, first + 1, west, east);
121       return first + 2;
122     }
123
124     // move west to its final place and east temporarily to the end
125
126     I last = past - 1;
127     parallel_iter_swap(first, last, west, east);
128     west = first;
129     east = last;
130
131     // find the south and north poles
132
133     pair = std::minmax_element(first, past,
134       [](P const& a, P const& b) -> bool {
135         return (a.y < b.y) or (a.y == b.y and a.x < b.x);
136       });
137     I south = std::get<0>(pair);
138     I north = std::get<1>(pair);
139     bool east_north_same = (*east == *north);
140     bool west_south_same = (*west == *south);
141     bool west_north_same = (*west == *north);
142     bool east_south_same = (*east == *south);
143
144     // recall the quadrant borders
145
146     T west_y = (*west).y;
147     T east_y = (*east).y;
```

```
148     T north_x = (*north).x;
149     T south_x = (*south).x;
150
151     // north–west corner
152
153     I done = west + 1;
154     if (not west_north_same) {
155       I rest = std::partition(done, last,
156         [&](P const& a) → bool {
157           return (a.x ≤ north_x) and (a.y ≥ west_y);
158         });
159       if (rest ≠ done) {
160         std::sort(done, rest,
161           [](P const& a, P const& b) → bool {
162             return (a.x < b.x);
163           });
164         rest = go_upstairs(west, rest);
165         rest = scan(west, rest);
166         done = scan_one_more(west, rest − 1, east);
167       }
168     }
169
170     // north–east corner
171
172     if (not east_north_same) {
173       I top = done − 1;
174       I rest = std::partition(done, last,
175         [&](P const& a) → bool {
176           return (a.x ≥ north_x) and (a.y ≥ east_y);
177         });
178       if (rest ≠ done) {
179         std::sort(top, rest,
180           [](P const& a, P const& b) → bool {
181             return (a.x > b.x);
182           });
183         rest = go_upstairs(top, rest);
184         std::reverse(top, rest);
185         rest = scan(top, rest);
186         done = scan_one_more(top, rest − 1, east);
187       }
188     }
189
190     // move east to its final place
191
192     std::iter_swap(done, east);
193     east = done;
194     done = done + 1;
195
196     // south–east corner
197
198     if (not east_south_same) {
199       I rest = std::partition(done, past,
```

```
200         [&](P const& a) → bool {
201           return (a.x ≥ south_x) and (a.y ≤ east_y);
202         });
203       if (rest ≠ done) {
204         std::sort(done, rest,
205           [](P const& a, P const& b) → bool {
206             return (a.x > b.x);
207           });
208         rest = go_downstairs(east, rest);
209         rest = scan(east, rest);
210         done = scan_one_more(east, rest − 1, west);
211       }
212     }
213
214     // south–west corner
215
216     if (not west_south_same) {
217       I bottom = done − 1;
218       I rest = std::partition(done, past,
219         [&](P const& a) → bool {
220           return (a.x ≤ south_x) and (a.y ≤ west_y);
221         });
222       if (rest ≠ done) {
223         std::sort(bottom, rest,
224           [](P const& a, P const& b) → bool {
225             return (a.x < b.x);
226           });
227         rest = go_downstairs(bottom, rest);
228         std::reverse(bottom, rest);
229         rest = scan(bottom, rest);
230         done = scan_one_more(bottom, rest − 1, west);
231       }
232     }
233     return done;
234   }
235
236   template<typename I>
237   bool check(I first, I past) {
238     using P = typename std::iterator_traits<I>::value_type;
239     using S = std::vector<P>;
240     using J = typename S::iterator;
241     S data;
242     std::size_t n = past − first;
243     data.resize(n);
244     std::copy(first, past, data.begin());
245     J rest = solve(data.begin(), data.end());
246     bool ok = validation::same_multiset(data.begin(), data.end(),
      ↪ first, past) and validation::convex_polygon(data.begin(),
      ↪ rest) and validation::all_inside(rest, data.end(),
      ↪ data.begin(), rest);
247     return ok;
248   }
```

```
249 }
250
251 #endif
```

B.7 poles_first.h++

```
1  /*
2    Performance Engineering Laboratory © 2017–2018
3
4    Find the extrema in four directions and eliminate all points inside
5    the convex hull of these points.
6  */
7
8  #ifndef __POLES_FIRST__
9  #define __POLES_FIRST__
10
11 #include <algorithm> // std::sort std::partition ...
12 #include <cassert> // assert macro
13 #include <cstdlib> // std::size_t
14 #include <iterator> // std::iterator_traits
15 #include "plane_sweep.h++" // plane_sweep::solve
16 #include "point.h++" // left_turn
17 #include <utility> // std::iter_swap std::pair std::get ...
18 #include "validation.h++" // same_multiset convex_polygon all_inside
19 #include <vector> // std::vector
20
21 namespace poles_first {
22
23   template<typename I>
24   std::vector<I> find_poles(I first, I past) {
25     assert(first != past);
26     using P = typename std::iterator_traits<I>::value_type;
27     std::vector<I> position;
28     position.resize(4);
29     std::pair<I, I> pair = std::minmax_element(first, past,
30       [](P const& a, P const& b) -> bool {
31         return (a.x < b.x) or (a.x == b.x and a.y < b.y);
32       });
33     enum {west = 0, east = 1, south = 2, north = 3};
34     position[west] = std::get<0>(pair);
35     position[east] = std::get<1>(pair);
36     pair = std::minmax_element(first, past,
37       [](P const& a, P const& b) -> bool {
38         return (a.y < b.y) or (a.y == b.y and a.x < b.x);
39       });
40     position[south] = std::get<0>(pair);
41     position[north] = std::get<1>(pair);
42     return position;
43   }
44
45   template<typename S>
```

```
46  void remove_duplicates(S& sequence) {
47    assert(sequence.size() != 0);
48    assert(std::is_sorted(sequence.begin(), sequence.end()));
49    std::size_t i = 0;
50    std::size_t j = 1;
51    auto v = sequence[0];
52    while (j != sequence.size()) {
53      if (not (sequence[j] == v)) {
54        std::swap(sequence[i], v);
55        ++i;
56        v = sequence[j];
57      }
58      ++j;
59    }
60    std::swap(sequence[i], v);
61    ++i;
62    std::size_t leftovers = sequence.size() - i;
63    while (leftovers != 0) {
64      sequence.pop_back();
65      --leftovers;
66    }
67  }
68
69  template<typename P, typename I>
70  bool inside_quadrilateral(P const& p, I upper, I lower, I past) {
71    assert(past - upper > 2);
72    assert(*upper == *(past - 1));
73    assert(*upper < *(upper + 1));
74    assert(*upper < *(past - 2));
75    I i = upper;
76    do {
77      ++i;
78    } while (i != lower and *i < p);
79    if (left_turn(*(i - 1), *i, p)) {
80      return false;
81    }
82    I last = past - 1;
83    I j = lower;
84    do {
85      ++j;
86    } while (j != last and *j > p);
87    if (left_turn(*(j - 1), *j, p)) {
88      return false;
89    }
90    return true;
91  }
92
93  template<typename I>
94  I remove_inner_part(I first, I past, I polygon, I rest) {
95    assert(polygon != rest);
96    I i = first;
97    if (rest - polygon == 1) {
```

```
98        for (I j = first; j != past; ++j) {
99          if (not (*j == *polygon)) {
100           std::iter_swap(i, j);
101           ++i;
102         }
103       }
104       return i;
105     }
106     if (rest - polygon == 2) {
107       for (I j = first; j != past; ++j) {
108         if (not (on_line_segment(*polygon, *(polygon + 1), *j))) {
109           std::iter_swap(i, j);
110           ++i;
111         }
112       }
113       return i;
114     }
115     // rest - polygon > 2
116     using P = typename std::iterator_traits<I>::value_type;
117     std::vector<P> approximate_hull;
118     for (I k = polygon; k != rest; ++k) {
119       approximate_hull.push_back(*k);
120     }
121     approximate_hull.push_back(*polygon);
122     auto upper = approximate_hull.begin();
123     auto lower = upper;
124     auto end = approximate_hull.end();
125     for (auto k = upper + 1; k != end - 1; ++k) {
126       if (*lower < *k) {
127         lower = k;
128       }
129     }
130     for (I j = first; j != past; ++j) {
131       if (not inside_quadrilateral(*j, upper, lower, end)) {
132         std::iter_swap(i, j);
133         ++i;
134       }
135     }
136     return i;
137   }
138
139   template<typename I>
140   I prune(I first, I past) {
141     if (past - first < 2) {
142       return past;
143     }
144     std::vector<I> poles = find_poles(first, past);
145     std::sort(poles.begin(), poles.end());
146     remove_duplicates(poles);
147     std::size_t n = poles.size();
148     for (std::size_t i = 0; i != n; ++i) {
149       std::iter_swap(first + i, poles[i]);
```

```
150       }
151       I rest = plane_sweep::solve(first, first + n);
152       I eliminated = remove_inner_part(rest, past, first, rest);
153       return eliminated;
154     }
155
156     template<typename I>
157     I solve(I first, I past) {
158       I rest = prune(first, past);
159       return plane_sweep::solve(first, rest);
160     }
161
162     template<typename I>
163     bool check(I first, I past) {
164       using P = typename std::iterator_traits<I>::value_type;
165       using S = std::vector<P>;
166       using J = typename S::iterator;
167       S data;
168       std::size_t n = past - first;
169       data.resize(n);
170       std::copy(first, past, data.begin());
171       J rest = solve(data.begin(), data.end());
172       bool ok = validation::same_multiset(data.begin(), data.end(),
          ↪ first, past) and validation::convex_polygon(data.begin(),
          ↪ rest) and validation::all_inside(rest, data.end(),
          ↪ data.begin(), rest);
173       return ok;
174     }
175   }
176
177   #endif
```

C. Micro-benchmarks

C.1 *multi_precision.h++*

```
/*
  This dummy performs Graham's scan
*/

#define MULTI_PRECISION

#include <functional>
#include <iterator>
#include "plane_sweep.h++"
#include "point.h++"
#include <utility>

namespace multi_precision {

  template<typename I>
  I solve(I first, I past) {
    using P = typename std::iterator_traits<I>::value_type;
    using T = typename P::coordinate;
    std::pair<I, I> pair = plane_sweep::clean(first, past,
    ↪ std::less<T>());
    I top = std::get<0>(pair);
    I next = std::get<1>(pair);
    I interior = plane_sweep::scan(first, top, next, past);
    return interior;
  }
}
```

C.2 *double_precision.h++*

```
/*
  This dummy performs Graham's scan
*/

#define DOUBLE_PRECISION

#include <functional>
#include <iterator>
#include "plane_sweep.h++"
#include <utility>

namespace double_precision {

  template<typename I>
  I solve(I first, I past) {
    using P = typename std::iterator_traits<I>::value_type;
    using T = typename P::coordinate;
```

```
18      std::pair<I, I> pair = plane_sweep::clean(first, past,
        ↪ std::less<T>());
19      I top = std::get<0>(pair);
20      I next = std::get<1>(pair);
21      I interior = plane_sweep::scan(first, top, next, past);
22      return interior;
23    }
24  }
```

C.3 *floating_point_filter.h++*

```
1   /*
2     This dummy performs Graham's scan
3   */
4
5   #define FLOATING_POINT_FILTER
6
7   #include <functional>
8   #include <iterator>
9   #include "plane_sweep.h++"
10  #include <utility>
11
12  namespace floating_point_filter {
13
14    template<typename I>
15    I solve(I first, I past) {
16      using P = typename std::iterator_traits<I>::value_type;
17      using T = typename P::coordinate;
18      std::pair<I, I> pair = plane_sweep::clean(first, past,
        ↪ std::less<T>());
19      I top = std::get<0>(pair);
20      I next = std::get<1>(pair);
21      I interior = plane_sweep::scan(first, top, next, past);
22      return interior;
23    }
24  }
```

C.4 *minmax_element.h++*

```
1   /*
2     This dummy finds the two extreme points on the left and the right
3   */
4
5   #include <algorithm>
6   #include <utility>
7
8   namespace minmax_element {
9
10    template<typename I>
11    I solve(I first, I past) {
```

```
12      using P = typename std::iterator_traits<I>::value_type;
13      std::pair<I, I> pair = std::minmax_element(first, past,
14        [](P const& a, P const& b) -> bool {
15          return (a.x < b.x) or (a.x == b.x and a.y < b.y);
16        });
17      return std::get<0>(pair);
18    }
19  }
```

C.5 *partition.h++*

```
1   /*
2     This dummy partitions the sequence into two by the line which
3     passes through two points
4   */
5
6   #include <iterator>
7   #include "point.h++"
8
9   namespace partition {
10
11    template<typename I>
12    I solve(I first, I past) {
13      using P = typename std::iterator_traits<I>::value_type;
14      I west = first;
15      I east = past - 1;
16      I middle = std::partition(west + 1, past,
17        [&](P const& q) -> bool {
18          return not left_turn(*west, q, *east);
19        });
20      return middle;
21    }
22  }
```

C.6 *copy.h++*

```
1   /*
2     This dummy copies the points from a vector to an array
3   */
4
5   #include <algorithm>
6   #include <iterator>
7   #include <vector>
8
9   namespace copy {
10
11    template<typename I>
12    I solve(I first, I past) {
13      using point = typename std::iterator_traits<I>::value_type;
14      auto n = std::distance(first, past);
```

```
15      point* copy = new point[n];
16      std::copy(first, past, &copy[0]);
17      delete[] copy;
18      return past;
19    }
20  }
```

C.7 *sort.h++*

```
1   /*
2     This dummy sorts the points according to their x-coordinates
3   */
4
5   #include <algorithm> // std::sort
6   #include <iterator> // std::iterator_traits
7
8   namespace sort {
9
10    template<typename I>
11    I solve(I first, I past) {
12      using P = typename std::iterator_traits<I>::value_type;
13      std::sort(first, past,
14        [](P const& p, P const& q) {
15          return p.x < q.x;
16        });
17      return past;
18    }
19  }
```

C.8 *lexicographic_sort.h++*

```
1   /*
2     This dummy sorts the points in ascending lexicographic order
3   */
4
5   #include <algorithm> // std::sort
6   #include <iterator> // std::iterator_traits
7
8   namespace lexicographic_sort {
9
10    template<typename I>
11    I solve(I first, I past) {
12      using P = typename std::iterator_traits<I>::value_type;
13      std::sort(first, past,
14        [](P const& a, P const& b) -> bool {
15          return (a.x < b.x) or (a.x == b.x and a.y < b.y);
16        });
17      return past;
18    }
19  }
```

C.9 *angular_sort.h++*

```
/*
  Sort all the points around o lexicographically by polar angle and
  distance from o
*/

#include <algorithm> // std::sort
#include <cstddef> // std::size_t
#include <iterator> // std::iterator_traits

#define MULTI_PRECISION

#include "point.h++"

namespace angular_sort {

  template<typename I>
  I solve(I first, I past) {
    using P = typename std::iterator_traits<I>::value_type;
    I west = std::min_element(first, past,
      [](P const& p, P const& q) -> bool {
        return (p.x < q.x) or (p.x == q.x and p.y < q.y);
      });
    P o = *west;
    std::sort(first, past,
      [&](P const& p, P const& q) -> bool {
        int turn = orientation(o, p, q);
        if (turn == 0) {
          return (L_∞distance(o, p) < L_∞distance(o, q));
        }
        return (turn == +1);
      });
    return past;
  }
}
```

C.10 *bsort.h++*

```
/*
  This dummy sorts the points according to their x–coordinates
*/

#include "bucketsort.h++"
#include <iterator> // std::iterator_traits

namespace bsort {

  template<typename I>
  I solve(I first, I past) {
    using P = typename std::iterator_traits<I>::value_type;
```

```
13      using T = typename P::coordinate;
14      bucketsort::sort(first, past,
15        [](P const& p, P const& q) -> bool {
16          return p.x < q.x;
17        },
18        [](P const& p) -> T {
19          return p.x;
20        });
21      return past;
22    }
23  }
```

C.11 tsort.h++

```
1   /*
2     This dummy sorts the points according to their x-coordinates
3   */
4
5   #include <iterator> // std::iterator_traits
6   #include "two_phase_bucketsort.h++"
7
8   namespace tsort {
9
10    template<typename I>
11    I solve(I first, I past) {
12      using P = typename std::iterator_traits<I>::value_type;
13      using T = typename P::coordinate;
14      two_phase_bucketsort::sort(first, past,
15        [](P const& p, P const& q) -> bool {
16          return p.x < q.x;
17        },
18        [](P const& p) -> T {
19          return p.x;
20        });
21      return past;
22    }
23  }
```

C.12 shuffle.h++

```
1   /*
2     This dummy shuffles the points randomly
3   */
4
5   #include <algorithm> // std::shuffle
6   #include <random> // random-number generators
7
8   using N = unsigned long long;
9   using linear_congruential_engine =
```

```
    ↪ std::linear_congruential_engine<N, 6364136223846793005U,
    ↪ 1442695040888963407U, 0U>;
constexpr N init = 10164167376618180197U;
linear_congruential_engine generator{init};

namespace shuffle {

  template<typename I>
  void solve(I first, I past) {
    std::shuffle(first, past, generator);
  }
}
```

D. Helpers

D.1 point.h++

```
#ifndef __POINT__
#define __POINT__

#include <algorithm> // std::min std::max
#include <cassert> // assert macro
#include <cstddef> // std::size_t
#include <limits> // std::numeric_limits
#include <string> // std::string std::to_string

#if not defined(MULTI_PRECISION) and not
    ↪ defined(FLOATING_POINT_FILTER)
#define DOUBLE_PRECISION
#endif

#if defined(MULTI_PRECISION) or defined(FLOATING_POINT_FILTER)

#include "cphmpl/functions.h++" // cphmpl::width
#include "cphstl/integers.h++" // cphstl::ℤ

#endif

template<typename T>
class point {
public:

  using self = point<T>;
  using coordinate = T;

  T x;
  T y;

  explicit point()
    : x(0), y(0) {
  }

  point(T x_coordinate, T y_coordinate)
    : x(x_coordinate), y(y_coordinate) {
  }

  std::string to_string() const {
    return "(" + std::to_string(x) + ", " + std::to_string(y) + ")";
  }

  bool operator==(point<T> const& other) const {
    return x == other.x and y == other.y;
  }

  bool operator≠(point<T> const& other) const {
```

```
48      return not (*this == other);
49    }
50
51    bool operator<(point<T> const& other) const {
52      return (x < other.x) or (x == other.x and y < other.y);
53    }
54
55    bool operator>(point<T> const& other) const {
56      return other < *this;
57    }
58
59    friend std::ostream& operator<<(std::ostream& os, point<T>
        ↪ const& p) {
60      return os << p.to_string();
61    }
62  };
63
64  #if defined(MULTI_PRECISION) or defined(FLOATING_POINT_FILTER)
65
66  namespace multi_precision {
67
68    template<typename P>
69    bool left_turn(P const& p, P const& q, P const& r) {
70      using N = std::size_t;
71      using T = typename P::coordinate;
72      constexpr N w = cphmpl::width<T>;
73      using Z = cphstl::ℤ<2 * w + 2>;
74      Z p_x = Z(p.x);
75      Z p_y = Z(p.y);
76      Z lhs = (Z(q.x) - p_x) * (Z(r.y) - p_y);
77      Z rhs = (Z(r.x) - p_x) * (Z(q.y) - p_y);
78      return lhs > rhs;
79    }
80
81    template<typename P>
82    bool no_turn(P const& p, P const& q, P const& r) {
83  #ifdef MEASURE_TURNS
84
85      ++turns;
86
87  #endif
88      using N = std::size_t;
89      using T = typename P::coordinate;
90      constexpr N w = cphmpl::width<T>;
91      using Z = cphstl::ℤ<2 * w + 2>;
92      Z p_x = Z(p.x);
93      Z p_y = Z(p.y);
94      Z lhs = (Z(q.x) - p_x) * (Z(r.y) - p_y);
95      Z rhs = (Z(r.x) - p_x) * (Z(q.y) - p_y);
96      return lhs == rhs;
97    }
98
```

```
99    // compute the signed area of the parallelogram spanned by
100   // $\overrightarrow{pq}$ and $\overrightarrow{pr}$.
101
102   template<typename P>
103   auto signed_area(P const& p, P const& q, P const& r) {
104     using N = std::size_t;
105     using T = typename P::coordinate;
106     constexpr N w = cphmpl::width<T>;
107     using Z = cphstl::ℤ<2 * w + 3>;
108 #ifdef MEASURE_TURNS
109
110     ++turns;
111
112 #endif
113     Z p_x = Z(p.x);
114     Z p_y = Z(p.y);
115     Z dx_1 = Z(q.x) - p_x;
116     Z dx_2 = Z(r.x) - p_x;
117     Z dy_1 = Z(q.y) - p_y;
118     Z dy_2 = Z(r.y) - p_y;
119     Z result = dx_1 * dy_2 - dx_2 * dy_1;
120     return result;
121   }
122
123   // return the square of the distance between p and q
124
125   template<typename P>
126   auto distance_squared(P const& p, P const& q) {
127     using N = std::size_t;
128     using T = typename P::coordinate;
129     constexpr N w = cphmpl::width<T>;
130     using Z = cphstl::ℤ<2 * w + 3>;
131     Z p_x = Z(p.x);
132     Z p_y = Z(p.y);
133     Z q_x = Z(q.x);
134     Z q_y = Z(q.y);
135     return (p_x - q_x) * (p_x - q_x) + (p_y - q_y) * (p_y - q_y);
136   }
137
138   template<typename P>
139   auto L_∞distance(P const& p, P const& q) {
140     using N = std::size_t;
141     using T = typename P::coordinate;
142     constexpr N w = cphmpl::width<T>;
143     using Z = cphstl::ℤ<w + 2>;
144     Z dx = cphstl::abs(Z(q.x) - Z(p.x));
145     Z dy = cphstl::abs(Z(q.y) - Z(p.y));
146     return std::max(dx, dy);
147   }
148
149   template<typename P>
150   auto L_1_distance(P const& p, P const& q) {
```

```
151       using N = std::size_t;
152       using T = typename P::coordinate;
153       constexpr N w = cphmpl::width<T>;
154       using Z = cphstl::ℤ<w + 3>;
155       Z dx = cphstl::abs(Z(q.x) - Z(p.x));
156       Z dy = cphstl::abs(Z(q.y) - Z(p.y));
157       return dx + dy;
158     }
159
160     // return the orientation when moving from p to r via q
161     // +1: left turn
162     //  0: p, q, and r are collinear
163     // −1: right turn
164
165     template<typename P>
166     int orientation(P const& p, P const& q, P const& r) {
167       using N = std::size_t;
168       using T = typename P::coordinate;
169       constexpr N w = cphmpl::width<T>;
170       using Z = cphstl::ℤ<2 * w + 2>;
171       Z p_x = Z(p.x);
172       Z p_y = Z(p.y);
173       Z lhs = (Z(q.x) - p_x) * (Z(r.y) - p_y);
174       Z rhs = (Z(r.x) - p_x) * (Z(q.y) - p_y);
175       if (lhs == rhs) {
176         return 0;
177       }
178       return (lhs > rhs) ? +1 : −1;
179     }
180   }
181
182   #endif
183
184   #if defined(MULTI_PRECISION)
185
186   using namespace multi_precision;
187
188   #endif
189
190   #if defined(DOUBLE_PRECISION)
191
192   namespace double_precision {
193
194     template<typename P>
195     bool left_turn(P const& p, P const& q, P const& r) {
196
197   #ifdef MEASURE_TURNS
198
199       ++turns;
200
201   #endif
202
```

```
203     using T = typename P::coordinate;
204     using Z = long long;
205     Z mid_x = Z(q.x);
206     Z mid_y = Z(q.y);
207     Z dx_1 = mid_x - Z(p.x);
208     Z dx_2 = Z(r.x) - mid_x;
209     Z dy_1 = mid_y - Z(p.y);
210     Z dy_2 = Z(r.y) - mid_y;
211
212     /* We have 31 bits + 1 for the sign in int, and
213        we may need (31 (+1 from minus)) * 2 (from mult) = 64 bits
214        in total, but we only have 63 bits + 1 for the sign.
215        If we overflow the same direction, the order is preserved. */
216     bool check1 = Z(T(dx_1)) != dx_1 and Z(T(dy_2)) != dy_2;
217     bool check2 = Z(T(dx_2)) != dx_2 and Z(T(dy_1)) != dy_1;
218     if (check1 or check2) {
219       // overflow happening
220       if (check1) {
221         if ((dx_1 > 0) == (dy_2 > 0)) {
222           // first part overflows above
223           if (check2 and ((dx_2 > 0) == (dy_1 > 0))) {
224             // both overflow above
225             return dx_1 * dy_2 > dx_2 * dy_1;
226           }
227           else {
228             return true; // only first term overflows above
229           }
230         }
231         else {
232           // first term overflows below
233           if (check2 and ((dx_2 > 0) != (dy_1 > 0))) {
234             // second term also overflows below
235             return dx_1 * dy_2 > dx_2 * dy_1;
236           }
237           else {
238             // only first term overflows below
239             return false;
240           }
241         }
242       }
243       else {
244         // second term overflows; result is inverse of the direction
245         return (dx_2 > 0) != (dy_1 > 0);
246       }
247       return dx_1 * dy_2 > dx_2 * dy_1;
248     }
249     else {
250       // standard setting
251       return dx_1 * dy_2 > dx_2 * dy_1;
252     }
253   }
254
```

```
255   template<typename P>
256   bool no_turn(P const& p, P const& q, P const& r) {
257
258 #ifdef MEASURE_TURNS
259
260     ++turns;
261
262 #endif
263
264     using T = typename P::coordinate;
265     using Z = long long;
266     Z mid_x = Z(q.x);
267     Z mid_y = Z(q.y);
268     Z dx_1 = mid_x - Z(p.x);
269     Z dx_2 = Z(r.x) - mid_x;
270     Z dy_1 = mid_y - Z(p.y);
271     Z dy_2 = Z(r.y) - mid_y;
272
273     bool check1 = Z(T(dx_1)) != dx_1 and Z(T(dy_2)) != dy_2;
274     bool check2 = Z(T(dx_2)) != dx_2 and Z(T(dy_1)) != dy_1;
275     if (check1 or check2) {
276       // if they do not both overflow, they must be different
277       if (check1 and check2) {
278         // if overflow direction not the same, they are different
279         if (((dx_1 > 0) == (dy_2 > 0)) == ((dx_2 > 0) == (dy_1 >
    ↪ 0))) {
280           // the non-overflow part must also be the same
281           return dx_1 * dy_2 == dx_2 * dy_1;
282         }
283       }
284       return false;
285     }
286     else {
287       return dx_1 * dy_2 == dx_2 * dy_1;
288     }
289   }
290
291   // may do controlled overflow on very large values
292
293   template<typename P>
294   auto signed_area(P const& p, P const& q, P const& r) {
295     assert(not right_turn(p, q, r));
296
297 #ifdef MEASURE_TURNS
298
299     ++turns;
300
301 #endif
302
303     using Z = long long;
304     Z mid_x = Z(q.x);
305     Z mid_y = Z(q.y);
```

```
306       Z dx1 = midx - Z(p.x);
307       Z dx2 = Z(r.x) - midx;
308       Z dy1 = midy - Z(p.y);
309       Z dy2 = Z(r.y) - midy;
310       return (unsigned long long) (dx1 * dy2 - dx2 * dy1);
311     }
312   }
313
314   using namespace double_precision;
315
316   #endif
317
318   #if defined(FLOATING_POINT_FILTER)
319
320   namespace floating_point_filter {
321
322     template<typename P>
323     bool left_turn(P const& p, P const& q, P const& r) {
324       using R = double;
325       constexpr R u = std::numeric_limits<R>::epsilon();
326       R lastx = R(r.x);
327       R lasty = R(r.y);
328       R lhs = (R(p.x) - lastx) * (R(q.y) - lasty);
329       R rhs = (R(p.y) - lasty) * (R(q.x) - lastx);
330       R det = lhs - rhs;
331       R detsum = 0.0;
332       if (lhs > 0.0) {
333         if (rhs <= 0.0) {
334           return lhs > rhs;
335         }
336         else {
337           detsum = lhs + rhs;
338         }
339       }
340       else if (lhs < 0.0) {
341         if (rhs >= 0.0) {
342           return lhs > rhs;
343         }
344         else {
345           detsum = -lhs - rhs;
346         }
347       }
348       else {
349         return lhs > rhs;
350       }
351       R errbound = (3 * u + 16 * u * u) * detsum;
352       if (det >= errbound or -det >= errbound) {
353         return lhs > rhs;
354       }
355       return multi_precision::left_turn(p, q, r);
356     }
357
```

```
358   template<typename P>
359   bool no_turn(P const& p, P const& q, P const& r) {
360     using R = double;
361     constexpr R u = std::numeric_limits<R>::epsilon();
362     R last_x = R(r.x);
363     R last_y = R(r.y);
364     R lhs = (R(p.x) - last_x) * (R(q.y) - last_y);
365     R rhs = (R(p.y) - last_y) * (R(q.x) - last_x);
366     R det = lhs - rhs;
367     R detsum = 0.0;
368     if (lhs > 0.0) {
369       if (rhs <= 0.0) {
370         return false;
371       }
372       else {
373         detsum = lhs + rhs;
374       }
375     }
376     else if (lhs < 0.0) {
377       if (rhs >= 0.0) {
378         return false;
379       }
380       else {
381         detsum = -lhs - rhs;
382       }
383     }
384     else {
385       return lhs == rhs;
386     }
387     R errbound = (3 * u + 16 * u * u) * detsum;
388     if (det >= errbound or -det >= errbound) {
389       return lhs == rhs;
390     }
391     return multi_precision::no_turn(p, q, r);
392   }
393 }
394
395 using namespace floating_point_filter;
396
397 #endif
398
399 template<typename P>
400 bool right_turn(P const& p, P const& q, P const& r) {
401   return left_turn(r, q, p);
402 }
403
404 // is p on the line segment {q, r}?
405
406 template<typename P>
407 bool on_line_segment(P const& p, P const& q, P const& r) {
408   P left = std::min(q, r);
409   P right = std::max(q, r);
```

```
410     if (p < left) {
411       return false;
412     }
413     if (p > right) {
414       return false;
415     }
416     return no_turn(p, q, r);
417   }
418
419   #endif
```

D.2 validation.h++

```
1   #ifndef __VALIDATION__
2   #define __VALIDATION__
3
4   #include <algorithm> // std::copy std::sort std::equal std::rotate
5   #include <cassert> // assert macro
6   #include <cstdlib> // std::size_t
7   #include <iostream> // std streams
8   #include <iterator> // std::iterator_traits
9   #include "point.h++" // left_turn right_turn
10  #include <vector> // std::vector
11
12  namespace validation {
13
14    template<typename I, typename J>
15    bool same_multiset(I p, I q, J r, J s) {
16      using P = typename std::iterator_traits<I>::value_type;
17      using S = std::vector<P>;
18      std::size_t n = q - p;
19      std::size_t m = s - r;
20      if (m != n) {
21        return false;
22      }
23      S backup;
24      backup.resize(n);
25      std::copy(p, q, backup.begin());
26      std::sort(backup.begin(), backup.end(),
27        [](P const& a, P const& b) -> bool {
28          return (a.x < b.x) or (a.x == b.x and a.y < b.y);
29        });
30      S other;
31      other.resize(n);
32      std::copy(r, s, other.begin());
33      std::sort(other.begin(), other.end(),
34        [](P const& a, P const& b) -> bool {
35          return (a.x < b.x) or (a.x == b.x and a.y < b.y);
36        });
37      return std::equal(backup.begin(), backup.end(), other.begin(),
38        [](P const& a, P const& b) -> bool {
```

```
39          return (a.x == b.x) and (a.y == b.y);
40        });
41    }
42
43    // [p, r) circular chain of vertices
44
45    template<typename I>
46    bool left_turns_only(I p, I r) {
47      std::size_t h = r - p;
48      if (h < 3) {
49        return true;
50      }
51      for (I q = p; q != r; ++q) {
52        I next = q + 1;
53        I next_after = next + 1;
54        if (q == r - 2) {
55          next_after = p;
56        }
57        else if (q == r - 1) {
58          next = p;
59          next_after = p + 1;
60        }
61        if (not left_turn(*q, *next, *next_after)) {
62          std::cerr << "Left-turn: " << *q << " " << *next << " "
63                    << *next_after << " failed\n";
64          return false;
65        }
66      }
67      return true;
68    }
69
70    // [p, r) circular chain of vertices
71
72    template<typename I>
73    bool right_turns_only(I p, I r) {
74      std::size_t h = r - p;
75      if (h < 3) {
76        return true;
77      }
78      for (I q = p; q != r; ++q) {
79        I next = q + 1;
80        I next_after = next + 1;
81        if (q == r - 2) {
82          next_after = p;
83        }
84        else if (q == r - 1) {
85          next = p;
86          next_after = p + 1;
87        }
88        if (not right_turn(*q, *next, *next_after)) {
89          std::cerr << "Right-turn: " << *q << " " << *next << " "
90                    << *next_after << " failed\n";
```

```
91          return false;
92        }
93      }
94      return true;
95    }
96
97    // [p, r) half–circular chain of vertices
98
99    template<typename I>
100   bool monotone(I p, I r) {
101     assert(p != r);
102     using P = typename std::iterator_traits<I>::value_type;
103     std::size_t h = r - p;
104     if (h == 0 or h == 1) {
105       return true;
106     }
107     I q = p;
108     while (q != r - 1) {
109       P a = *q;
110       P b = *(q + 1);
111       if (not (a.x < b.x or (a.x == b.x and a.y < b.y))) {
112         std::cerr << "Convex: " << a << " " << b
113                   << ": not monotone\n";
114         return false;
115       }
116       ++q;
117     }
118     return true;
119   }
120
121   // [p, r) circular chain of vertices
122
123   template<typename I>
124   bool convex_polygon(I p, I r) {
125     using P = typename std::iterator_traits<I>::value_type;
126     using S = std::vector<P>;
127     using J = typename S::iterator;
128     std::size_t h = r - p;
129     if (h == 0 or h == 1) {
130       return true;
131     }
132     if (h == 2) {
133       if (*p != *(p + 1)) {
134         return true;
135       }
136       else {
137         std::cerr << "Convex: h = 2: two equal points\n";
138         return false;
139       }
140     }
141     // h ≥ 3
142     bool clockwise = right_turn(*p, *(p + 1), *(p + 2));
```

```
143       if (clockwise and (not right_turns_only(p, r))) {
144         std::cerr << "Convex: h = " << h
145                   << ": not a spiral (cw)\n";
146         return false;
147       }
148       if (not clockwise and (not left_turns_only(p, r))) {
149         std::cerr << "Convex: h = " << h
150                   << ": not a spiral (ccw)\n";
151         return false;
152       }
153
154       using P = typename std::iterator_traits<I>::value_type;
155       using S = std::vector<P>;
156       using J = typename S::iterator;
157       S hull;
158       hull.resize(h);
159       std::copy(p, r, hull.begin());
160       J west = std::min_element(hull.begin(), hull.end(),
161         [](P const& a, P const& b) -> bool {
162           return (a.x < b.x) or (a.x == b.x and a.y < b.y);
163         });
164       (void) std::rotate(hull.begin(), west, hull.end());
165       hull.push_back(*west);
166       west = hull.begin();
167       J east = std::max_element(hull.begin(), hull.end() - 1,
168         [](P const& a, P const& b) -> bool {
169           return (a.x < b.x) or (a.x == b.x and a.y < b.y);
170         });
171       assert(west != east);
172       if (clockwise) {
173         if (not monotone(west, east + 1)) {
174           std::cerr << "Convex: h = " << h
175                     << ": upper hull not monotone (cw)\n";
176           return false;
177         }
178         std::reverse(east, hull.end());
179         if (not monotone(east, hull.end())) {
180           std::cerr << "Convex: h = " << h
181                     << ": lower hull not monotone (cw)\n";
182           return false;
183         }
184         return true;
185       }
186       // counterclockwise
187       if (not monotone(west, east + 1)) {
188         std::cerr << "Convex: h = " << h
189                   << ": lower hull not monotone (ccw)\n";
190         return false;
191       }
192       std::reverse(east, hull.end());
193       if (not monotone(east, hull.end())) {
194         std::cerr << "Convex: h = " << h
```

```
195                 << ": upper hull not monotone (ccw)\n";
196       return false;
197     }
198     return true;
199   }
200
201   // [p, r) upper hull from left to right
202
203   template<typename I, typename P>
204   bool above(I p, I r, P const& u) {
205     std::size_t h = r - p;
206     assert(h > 1);
207     I last = r - 1;
208     assert(u.x ≥ (*p).x and u.x ≤ (*last).x);
209     if ((*p).x == (*(p + 1)).x) {
210       ++p;
211     }
212     if (h ≠ 1 and (*last).x == (*(last - 1)).x) {
213       --r;
214     }
215     I q = std::lower_bound(p, r, u,
216       [](P const& a, P const& b) → bool {
217         return (a.x < b.x);
218       });
219     if (q == p) {
220       return u.y > (*p).y;
221     }
222     return left_turn(*(q - 1), *q, u);
223   }
224
225   // [p, r) lower hull from left to right
226
227   template<typename I, typename P>
228   bool below(I p, I r, P const& u) {
229     std::size_t h = r - p;
230     assert(h > 1);
231     I last = r - 1;
232     assert(u.x ≥ (*p).x and u.x ≤ (*last).x);
233     if ((*p).x == (*(p + 1)).x) {
234       ++p;
235     }
236     if (h ≠ 1 and (*last).x == (*(last - 1)).x) {
237       --r;
238     }
239     I q = std::lower_bound(p, r, u,
240       [](P const& a, P const& b) → bool {
241         return (a.x < b.x);
242       });
243     if (q == p) {
244       return u.y < (*p).y;
245     }
246     return right_turn(*(q - 1), *q, u);
```

```
247   }
248
249   // [r, s) multiset of points; [p, q) convex polygon
250
251   template<typename I, typename J>
252   bool all_inside(I r, I s, J p, J q) {
253     std::size_t h = q - p;
254     if (h == 0) {
255       if (r != s) {
256         std::cerr << "Inside: h = 0: no points can be inside\n";
257         return false;
258       }
259       return true;
260     }
261     if (h == 1) {
262       for (J i = r; i != s; ++i) {
263         if (*i != *p) {
264           std::cerr << "Inside: h = 1: all points not equal\n";
265           return false;
266         }
267       }
268       return true;
269     }
270     if (h == 2) {
271       for (J i = r; i != s; ++i) {
272         if (not on_line_segment(*i, *p, *(p + 1))) {
273           std::cerr << "Inside: h = 2: all points not collinear\n";
274           return false;
275         }
276       }
277       return true;
278     }
279     using P = typename std::iterator_traits<I>::value_type;
280     using S = std::vector<P>;
281     using K = typename S::iterator;
282     S hull;
283     hull.resize(h);
284     std::copy(p, q, hull.begin());
285     K west = std::min_element(hull.begin(), hull.end(),
286       [](P const& a, P const& b) -> bool {
287         return (a.x < b.x) or (a.x == b.x and a.y < b.y);
288       });
289     (void) std::rotate(hull.begin(), west, hull.end());
290     hull.push_back(*west);
291     west = hull.begin();
292     K east = std::max_element(hull.begin(), hull.end() - 1,
293       [](P const& a, P const& b) -> bool {
294         return (a.x < b.x) or (a.x == b.x and a.y < b.y);
295       });
296     assert(west != east);
297     for (I i = r; i != s; ++i) {
298       if ((*i).x < (*west).x) {
```

```
299           std::cerr << "Inside: " << *i
300                     << " on left of the output\n";
301           return false;
302         }
303         if ((*i).x > (*east).x) {
304           std::cerr << "Inside: " << *i
305                     << " on right of the output\n";
306           return false;
307         }
308       }
309       bool clockwise = right_turn(*west, *(west + 1), *(west + 2));
310       if (clockwise) {
311         for (I i = r; i != s; ++i) {
312           if (above(west, east + 1, *i)) {
313             std::cerr << "Inside: " << *i
314                       << " above the upper hull (cw)\n";
315             return false;
316           }
317         }
318         std::reverse(east, hull.end());
319         for (I i = r; i != s; ++i) {
320           if (below(east, hull.end(), *i)) {
321             std::cerr << "Inside: " << *i
322                       << " below the lower hull (cw)\n";
323             return false;
324           }
325         }
326       }
327       else { // counterclockwise
328         for (I i = r; i != s; ++i) {
329           if (below(west, east + 1, *i)) {
330             std::cerr << "Inside: " << *i
331                       << " below the lower hull (ccw)\n";
332             return false;
333           }
334         }
335         std::reverse(east, hull.end());
336         for (I i = r; i != s; ++i) {
337           if (above(east, hull.end(), *i)) {
338             std::cerr << "Inside: " << *i
339                       << " above the upper hull (ccw)\n";
340             return false;
341           }
342         }
343       }
344       return true;
345     }
346   }
347
348   #endif
```

D.3 bucketsort.h++

```
/*
  This program implements bucketsort

  Source: O. Nevalainen and T. Raita, An internal hybrid sort
  algorithm revisited, The Computer Journal 35,2 (1992), 177–183

  Author: Jyrki Katajainen © 2018

  Worst–case running time: O(n lg n)
  Parameter: m = min(n/5, 2^16)
  Average–case running time: O(n lg(n/m))
  Extra space: n elements, m + O(lg n) words

  Observed sorting time per n lg n [ns]; input: random permutation
      1024      1.07
     32768      0.82
   1048576      1.17
  33554432      2.04
*/

#include <algorithm> // std::sort std::copy
#include <cassert> // assert macro
#include <cstddef> // std::ptrdiff_t
#include <iterator> // std::distance std::iterator_traits
#include <utility> // std::pair

namespace bucketsort {

  constexpr std::ptrdiff_t threshold = 100;
  constexpr std::ptrdiff_t max_m = (1 << 16);

  template<typename V>
  class identity {
  public:

    V operator()(V const& e) const {
      return e;
    }

    V& operator()(V& e) const {
      return e;
    }
  };

  template<typename D, typename V, typename K = identity<V>>
  class bucket {
  public:

    bucket(D m, V const& min, V const& max, K key = K())
```

```
50        : a(double(m - 1) / (double(key(max)) - double(key(min)))),
51          b(double(m - 1) * double(key(min)) / (double(key(max)) -
      ↪ double(key(min)))),
52          key(key) {
53      }
54
55      D operator()(V const& x) {
56        return D(a * double(key(x)) - b);
57      }
58
59    private:
60
61      double a;
62      double b;
63      K key;
64    };
65
66    template<typename R, typename C, typename K = identity<typename
      ↪ std::iterator_traits<R>::value_type>>
67    void sort(R data, R past, C compare, K key = K()) {
68      using V = typename std::iterator_traits<R>::value_type;
69      using D = typename std::iterator_traits<R>::difference_type;
70      D const n = std::distance(data, past);
71
72      // small input
73
74      if (n < threshold) {
75        std::sort(data, past, compare);
76        return;
77      }
78      D m = std::min(n / 5, max_m);
79      V* copy = new V[n];
80      D* counter = new D[m + 1];
81
82      // initialization
83
84      std::pair<R, R> pair = std::minmax_element(data, past,
      ↪ compare);
85      R leftmost = std::get<0>(pair);
86      R rightmost = std::get<1>(pair);
87      if (key(*leftmost) == key(*rightmost)) {
88        return;
89      }
90      bucket formula(m, *leftmost, *rightmost, key);
91      for (D j = 0; j ≤ m; ++j) {
92        counter[j] = 0;
93      }
94
95      // determination of the number of elements in the buckets
96
97      for (D i = 0; i ≠ n; ++i) {
98        D j = formula(*(data + i));
```

```
99        counter[j] = counter[j] + 1;
100       copy[i] = *(data + i);
101     }
102     assert(counter[m] == 0);
103
104     // initialization of the cursors to the buckets
105
106     for (D j = 1; j ≤ m; ++j) {
107       counter[j] = counter[j − 1] + counter[j];
108     }
109
110     // construction of the buckets
111
112     for (D i = n − 1; i ≥ 0; −−i) {
113       D j = formula(copy[i]);
114       counter[j] = counter[j] − 1;
115       data[counter[j]] = copy[i];
116     }
117
118     // sorting the buckets
119
120     for (D j = 0; j ≠ m; ++j) {
121       std::sort(data + counter[j], data + counter[j + 1], compare);
122     }
123     delete[] copy;
124     delete[] counter;
125   }
126 }
```

D.4 two_phase_bucketsort.h++

```
1  /*
2    This program implements two–phase bucketsort
3
4    Author: Jyrki Katajainen © 2018
5
6    Worst–case running time: O(n lg n)
7    Parameter: m
8    Average–case running time: O(n)
9    Extra space: O(√n + m) words
10
11   Observed sorting time per n lg n [ns]; input: random permutation
12       1024      3.33
13      32768      1.92
14    1048576      2.09
15   33554432      2.41
16 */
17
18 #include <algorithm> // std::sort std::copy
19 #include "bucketsort.h++"
20 #include <cassert> // assert macro
```

```
21 #include <cmath> // std::sqrt
22 #include <cstddef> // std::ptrdiff_t
23 #include <iterator> // std::distance std::iterator_traits
24 #include <utility> // std::pair
25
26 namespace two_phase_bucketsort {
27
28   constexpr std::ptrdiff_t threshold = 100;
29
30   template<typename R, typename C, typename K =
       ↪ bucketsort::identity<typename
       ↪ std::iterator_traits<R>::value_type>>
31   void sort(R data, R past, C compare, K key = K()) {
32     using D = typename std::iterator_traits<R>::difference_type;
33     D const n = std::distance(data, past);
34
35     // small input
36
37     if (n < threshold) {
38       std::sort(data, past, compare);
39       return;
40     }
41     D m = D(std::sqrt(n));
42     D* counter = new D[m];
43     D* cursor = new D[m + 1];
44
45     // initialize the bucket formula
46
47     std::pair<R, R> pair = std::minmax_element(data, past,
       ↪ compare);
48     R leftmost = std::get<0>(pair);
49     R rightmost = std::get<1>(pair);
50     if (key(*leftmost) == key(*rightmost)) {
51       return;
52     }
53     bucketsort::bucket formula(m, *leftmost, *rightmost, key);
54     for (D j = 0; j != m; ++j) {
55       counter[j] = 0;
56     }
57
58     // determine the number of elements in the buckets
59
60     for (D i = 0; i != n; ++i) {
61       D j = formula(*(data + i));
62       counter[j] = counter[j] + 1;
63     }
64
65     // initialize the cursors to the buckets
66
67     cursor[0] = counter[0];
68     for (D j = 1; j != m; ++j) {
69       cursor[j] = cursor[j - 1] + counter[j];
```

```
70      }
71      cursor[m] = n;
72
73      // move the elements into the buckets
74
75      D home = m - 1;
76      for (D k = 0; k != n; ++k) {
77        while (counter[home] == 0) {
78          --home;
79        }
80        D j = formula(data[cursor[home] - 1]);
81        if (home != j) {
82          std::swap(data[cursor[home] - 1], data[cursor[j] - 1]);
83        }
84        counter[j] = counter[j] - 1;
85        cursor[j] = cursor[j] - 1;
86      }
87
88      // sort the buckets
89
90      for (D j = 0; j != m; ++j) {
91        bucketsort::sort(data + cursor[j], data + cursor[j + 1],
        ↪ compare, key);
92      }
93      delete[] counter;
94      delete[] cursor;
95    }
96  }
```

E. Drivers

E.1 check–driver.c++

```
#include "algorithm.h++" // NAME::solve NAME::check
#include <cassert> // assert macro
#include <iterator> // std::distance
#include "point.h++" // point
#include <vector> // std::vector

int main() {
  using P = point<int>;
  using S = std::vector<P>;
  using I = typename S::iterator;

  S bag{P(0, 0), P(0, 1), P(0, 2), P(0, 1)};
  I rest = NAME::solve(bag.begin(), bag.end());
  auto h = std::distance(bag.begin(), rest);
  assert(h == 2);
  assert(NAME::check(bag.begin(), bag.end()));
}
```

E.2 test–driver.c++

```
/*
  Performance Engineering Laboratory © 2017–2018

  Brownie points:
  11 Sep 2017: The test cases for overflow detection by
    ↪ Marius–Florin Cristian
*/

#include <algorithm> // std::copy std::equal
#include <cassert> // assert macro
#include <cstdlib> // std::rand std::size_t
#include <exception> // std::exception
#include <iostream> // std::cout std::cerr
#include <iterator> // std::iterator_traits
#include <limits> // std::numeric_limits
#include <random> // std::linear_congruential_engine
#include <vector> // std::vector

#include "algorithm.h++" // NAME::check
#include "point.h++" // point

using linear_congruential_engine =
    ↪ std::linear_congruential_engine<unsigned long long,
    ↪ 6364136223846793005U, 1442695040888963407U, 0U>;
unsigned long long const seed = 10164167376618180197U;
linear_congruential_engine random_number_generator{seed};

```

```
25 template<typename I>
26 using checker = bool (*)(I, I);
27
28 template<typename A>
29 using testcase = void (*)(A);
30
31 template<typename I>
32 void generate(I p, I r) {
33   using P = typename std::iterator_traits<I>::value_type;
34   using T = typename P::coordinate;
35   T t = 100;
36   std::uniform_int_distribution<T> distribution(-t, t);
37   for (I q = p; q != r; ++q) {
38     T x = distribution(random_number_generator);
39     T y = distribution(random_number_generator);
40     *q = P(x, y);
41   }
42 }
43
44 template<typename checker>
45 void empty_set(checker f) {
46   std::cout << "empty set\n";
47   using P = point<int>;
48   using S = std::vector<P>;
49   S input;
50   assert(f(input.begin(), input.end()));
51 }
52
53 template<typename checker>
54 void one_point(checker f) {
55   std::cout << "one point\n";
56   using P = point<int>;
57   using S = std::vector<P>;
58   S input;
59   P origo;
60   input.push_back(origo);
61   assert(f(input.begin(), input.end()));
62 }
63
64 template<typename checker>
65 void two_points(checker f) {
66   std::cout << "two points\n";
67   using P = point<int>;
68   using S = std::vector<P>;
69   S input;
70   P origo;
71   P another(1, 1);
72   input.push_back(origo);
73   input.push_back(another);
74   assert(f(input.begin(), input.end()));
75 }
76
```

```
77  template<typename checker>
78  void three_points_on_a_vertical_line(checker f) {
79    std::cout << "three points on a vertical line\n";
80    using P = point<int>;
81    P origo;
82    P st(0, 1);
83    P nd(0, 2);
84    using S = std::vector<P>;
85    S input;
86    input.push_back(origo);
87    input.push_back(st);
88    input.push_back(nd);
89    assert(f(input.begin(), input.end()));
90  }
91
92  template<typename checker>
93  void three_points_on_a_horizontal_line(checker f) {
94    std::cout << "three points on a horizontal line\n";
95    using P = point<int>;
96    P origo;
97    P up(1, 0);
98    P top(2, 0);
99    using S = std::vector<P>;
100   S input;
101   input.push_back(origo);
102   input.push_back(up);
103   input.push_back(top);
104   assert(f(input.begin(), input.end()));
105 }
106
107 template<typename checker>
108 void ten_equal_points(checker f) {
109   std::cout << "ten equal points\n";
110   using P = point<int>;
111   P p(1, 1);
112   using S = std::vector<P>;
113   S input;
114   for (std::size_t i = 0; i != 10; ++i) {
115     input.push_back(p);
116   }
117   assert(f(input.begin(), input.end()));
118 }
119
120 template<typename checker>
121 void four_poles_and_many_duplicates_inside(checker f) {
122   std::cout << "four poles and many duplicates inside\n";
123   using P = point<int>;
124   using S = std::vector<P>;
125   std::size_t n = 10;
126   P origin(0, 0);
127   P west(-1, 0);
128   P north(0, 1);
```

```
129   P east(1, 0);
130   P south(0, -1);
131   S input;
132   input.push_back(west);
133   input.push_back(north);
134   input.push_back(east);
135   input.push_back(south);
136   for (std::size_t i = 0; i != n; ++i) {
137     input.push_back(origin);
138   }
139   assert(f(input.begin(), input.end()));
140 }
141
142 template<typename checker>
143 void four_poles_with_duplicates(checker f) {
144   std::cout << "four poles with duplicates\n";
145   using P = point<int>;
146   P west(-1, 0);
147   P north(0, 1);
148   P east(1, 0);
149   P south(0, -1);
150   using S = std::vector<P>;
151   S input;
152   for (std::size_t i = 0; i < 5; ++i) {
153     input.push_back(south);
154     input.push_back(north);
155     input.push_back(west);
156     input.push_back(east);
157   }
158   assert(f(input.begin(), input.end()));
159 }
160
161 template<typename checker>
162 void quadrilateral_with_duplicates(checker f) {
163   std::cout << "quadrilateral with duplicates\n";
164   using P = point<int>;
165   P bottom_left(0, 0);
166   P top_left(1, 1);
167   P bottom_right(4, 0);
168   P top_right(3, 1);
169   using S = std::vector<P>;
170   S input;
171   for (std::size_t i = 0; i < 5; ++i) {
172     input.push_back(bottom_left);
173     input.push_back(top_left);
174     input.push_back(bottom_right);
175     input.push_back(top_right);
176   }
177   assert(f(input.begin(), input.end()));
178 }
179
180 template<typename checker>
```

```
181 void many_east_pole_candidates(checker f) {
182   std::cout << "many east-pole candidates\n";
183   using P = point<int>;
184   P west(0, 0);
185   P east(1, 4);
186   P three(1, 3);
187   P two(1, 2);
188   P one(1, 1);
189   P zero(1, 0);
190   using S = std::vector<P>;
191   S input;
192   for (std::size_t i = 0; i < 5; ++i) {
193     input.push_back(west);
194     input.push_back(east);
195     input.push_back(three);
196     input.push_back(two);
197     input.push_back(zero);
198     input.push_back(one);
199   }
200   assert(f(input.begin(), input.end()));
201 }
202
203 template<typename checker>
204 void line_quadrilateral_line(checker f) {
205   std::cout << "line quadrilateral line\n";
206   using P = point<int>;
207   using S = std::vector<P>;
208   S input;
209   input.push_back(P(0, 0));
210   input.push_back(P(1, 0));
211   input.push_back(P(2, 0));
212   input.push_back(P(2, 0));
213   input.push_back(P(3, 0));
214
215   input.push_back(P(4, 1));
216   input.push_back(P(5, 2));
217   input.push_back(P(6, 3));
218   input.push_back(P(6, 3));
219   input.push_back(P(7, 2));
220   input.push_back(P(8, 1));
221
222   input.push_back(P(4, -1));
223   input.push_back(P(5, -2));
224   input.push_back(P(6, -3));
225   input.push_back(P(6, -3));
226   input.push_back(P(7, -2));
227   input.push_back(P(8, -1));
228
229   input.push_back(P(9, 0));
230   input.push_back(P(10, 0));
231   input.push_back(P(11, 0));
232   input.push_back(P(12, 0));
```

```
233   assert(f(input.begin(), input.end()));
234 }
235
236 template<typename checker>
237 void many_big_numbers(checker f) {
238   std::cout << "many big numbers\n";
239   using T = int;
240   using P = point<int>;
241   T max = std::numeric_limits<T>::max();
242   P origo(0, 0);
243   P left_bottom(-max, -max);
244   P just_above(-max, -max + 1);
245   P right_top(max, max);
246   P left_top(-max, max);
247   P neighbour_below(-max, max - 1);
248   P neighbour_beside(-max - 1, max);
249   using S = std::vector<P>;
250   S input;
251   input.push_back(left_bottom);
252   input.push_back(just_above);
253   input.push_back(origo);
254   input.push_back(left_top);
255   input.push_back(right_top);
256   input.push_back(neighbour_below);
257   input.push_back(neighbour_beside);
258   assert(f(input.begin(), input.end()));
259 }
260
261 template<typename checker>
262 void noisy_box(checker f) {
263   std::cout << "noisy box\n";
264   using T = int;
265   using P = point<T>;
266   T max = std::numeric_limits<T>::max();
267   using S = std::vector<P>;
268   using I = typename S::iterator;
269   constexpr std::size_t n = 10;
270   S input(n);
271   std::uniform_int_distribution<T> distribution(-max + 1, max - 1);
272   for (I q = input.begin(); q != input.end(); ++q) {
273     T x = distribution(random_number_generator);
274     T y = distribution(random_number_generator);
275     *q = P(x, y);
276   }
277   input.push_back(P(max, max));
278   input.push_back(P(max, -max));
279   input.push_back(P(-max, -max));
280   input.push_back(P(-max, max));
281   assert(f(input.begin(), input.end()));
282 }
283
284 template<typename checker>
```

```
285 void random_points_on_a_parabola(checker f) {
286   std::cout << "random points on a parabola\n";
287   using P = point<int>;
288   using S = std::vector<P>;
289   using I = typename S::iterator;
290   std::size_t const n = 4;
291   S input(n);
292   int j = 0;
293   for (I q = input.begin(); q != input.end(); ++q) {
294     *q = P(j, j * j);
295     ++j;
296   }
297   std::shuffle(input.begin(), input.end(), random_number_generator);
298   assert(f(input.begin(), input.end()));
299 }
300
301 template<typename checker>
302 void random_points_in_a_square(checker f) {
303   std::cout << "random points in a square\n";
304   std::size_t const bign = 10000;
305   for (std::size_t n = 0; n <= bign; ++n) {
306     if (n % 100 == 0) {
307       std::cout << "." << std::flush;
308     }
309     using P = point<int>;
310     using S = std::vector<P>;
311     S input(n);
312     generate(input.begin(), input.end());
313     assert(f(input.begin(), input.end()));
314   }
315   std::cout << "\n" << std::flush;
316 }
317
318 template<typename checker>
319 void positive_overflow(checker f) {
320   std::cout << "positive overflow\n";
321   using T = int;
322   using P = point<T>;
323   T max = std::numeric_limits<T>::max();
324   P origo(100, 100);
325   P left_top(100, max);
326   P right_top(max, max);
327   P middle(100000, 100000);
328   P right_bottom(max, 100);
329   using S = std::vector<P>;
330   S input;
331   input.push_back(left_top);
332   input.push_back(right_bottom);
333   input.push_back(origo);
334   input.push_back(middle);
335   input.push_back(right_top);
336   assert(f(input.begin(), input.end()));
```

```
337 }
338
339 template<typename checker>
340 void negative_overflow(checker f) {
341   std::cout << "negative overflow\n";
342   using T = int;
343   using P = point<T>;
344   T min = std::numeric_limits<T>::min();
345   P origo;
346   P left_top(min, 0);
347   P left_bottom(min, min);
348   P right_bottom(0, min);
349   P middle(-10000, -10000);
350   using S = std::vector<P>;
351   S input;
352   input.push_back(left_top);
353   input.push_back(right_bottom);
354   input.push_back(origo);
355   input.push_back(middle);
356   input.push_back(left_bottom);
357   assert(f(input.begin(), input.end()));
358 }
359
360 template<typename checker>
361 void integer_overflow(checker f) {
362   std::cout << "integer overflow\n";
363   using T = int;
364   using P = point<T>;
365   T min = std::numeric_limits<T>::min();
366   T max = std::numeric_limits<T>::max();
367   P origo;
368   P left_top(min, max);
369   P left_bottom(min, min);
370   P right_top(max, max);
371   P right_bottom(max, min);
372   using S = std::vector<P>;
373   S input;
374   input.push_back(left_top);
375   input.push_back(right_bottom);
376   input.push_back(origo);
377   input.push_back(right_top);
378   input.push_back(left_bottom);
379   assert(f(input.begin(), input.end()));
380 }
381
382 template<typename checker>
383 void negative_coordinates(checker f) {
384   std::cout << "negative coordinates\n";
385   using P = point<int>;
386   P p1(-3, -3);
387   P p2(-3, -2);
388   P p3(0, 0);
```

```
389   P p4(1, -1);
390   using S = std::vector<P>;
391   S input;
392   input.push_back(p1);
393   input.push_back(p2);
394   input.push_back(p3);
395   input.push_back(p4);
396   assert(f(input.begin(), input.end()));
397 }
398
399 template<typename checker>
400 void degenerate_coordinates(checker f) {
401   std::cout << "degenerate coordinates\n";
402   using P = point<int>;
403   P p1(0, 0);
404   P p2(1, 0);
405   P p3(2, 0);
406   P p4(3, 0);
407   P p5(3, 1);
408   P p6(3, 2);
409   P p7(3, 3);
410   P p8(2, 3);
411   P p9(1, 3);
412   P p10(0, 3);
413   using S = std::vector<P>;
414   S input;
415   input.push_back(p1);
416   input.push_back(p2);
417   input.push_back(p3);
418   input.push_back(p4);
419   input.push_back(p5);
420   input.push_back(p6);
421   input.push_back(p7);
422   input.push_back(p8);
423   input.push_back(p9);
424   input.push_back(p10);
425   assert(f(input.begin(), input.end()));
426 }
427
428 int main() {
429   using P = point<int>;
430   using S = std::vector<P>;
431   using I = typename S::iterator;
432   using R = checker<I>;
433   using T = testcase<R>;
434
435   T suite[] = {
436     empty_set,
437     one_point,
438     two_points,
439     three_points_on_a_vertical_line,
440     three_points_on_a_horizontal_line,
```

```
441       ten_equal_points,
442       four_poles_and_many_duplicates_inside,
443       four_poles_with_duplicates,
444       quadrilateral_with_duplicates,
445       many_east_pole_candidates,
446       line_quadrilateral_line,
447       many_big_numbers,
448       noisy_box,
449       random_points_on_a_parabola,
450       random_points_in_a_square,
451       positive_overflow,
452       negative_overflow,
453       integer_overflow,
454       negative_coordinates,
455       degenerate_coordinates
456     };
457
458     R program = NAME::check;
459     for (T test: suite) {
460       try {
461         test(program);
462       }
463       catch(std::exception& e) {
464         std::cerr << e.what() << std::endl;
465       }
466     }
467     return 0;
468   }
```

E.3 `driver.c++`

```
1   /*
2     Performance Engineering Laboratory © 2017–2018
3   */
4
5   //#define PRUNING
6
7   unsigned long maxsize = 128 * 1024 * 1024; // 512 * ...works slowly
8   // 1 << 30 = 1073741824 there is space, but things become very slow
9
10  #if defined(MEASURE_TURNS)
11
12  long long turns = 0;
13
14  #endif
15
16  #if defined(MEASURE_COMPS) or defined(MEASURE_MOVES)
17
18  long long comps = 0;
19  long long moves = 0;
20
```

```
21 #endif
22
23 #include <cassert> // assert macro
24 #include <cstdlib> // std::atoi std::size_t
25 #include <cmath> // std::sqrt
26 #include <ctime> // std::clock_t std::clock CLOCKS_PER_SEC
27 #include <iostream> // std::cout std::cerr
28 #include <iterator> // std::iterator_traits std::distance
29 #include <limits> // std::numeric_limits
30 #include <random> // std::linear_congruential_engine
31
32 #include "algorithm.h++" // NAME::solve
33 #include "point.h++" // point
34
35 using linear_congruential_engine = std::linear_congruential_engine<
36   unsigned long long, 6364136223846793005U, 1442695040888963407U,
      ↪ 0U>;
37 constexpr unsigned long long seed = 10164167376618180197U;
38 linear_congruential_engine random_number_generator{seed};
39
40 #if defined(DISC)
41
42 template<typename I>
43 void generate(I p, I r) {
44   using P = typename std::iterator_traits<I>::value_type;
45   using T = typename P::coordinate;
46   T t = std::numeric_limits<T>::max();
47   std::uniform_int_distribution<T> distribution(-t, t);
48
49   for (I q = p; q != r; ++q) {
50     T x = 0;
51     T y = 0;
52     using R = double;
53     R radius = R(t);
54     R root = 0.0;
55     do {
56       x = distribution(random_number_generator);
57       y = distribution(random_number_generator);
58       R dx = R(x) * R(x);
59       R dy = R(y) * R(y);
60       root = std::sqrt(dx + dy);
61     } while (root > radius);
62     *q = P(x, y);
63   }
64 }
65
66 #elif defined(UNIVERSE)
67
68 template<typename I>
69 void generate(I p, I r) {
70   std::size_t n = r - p;
71   using P = typename std::iterator_traits<I>::value_type;
```

```
72    using T = typename P::coordinate;
73    T t = std::sqrt(n);
74    std::uniform_int_distribution<T> distribution(-t, +t);
75
76    for (I q = p; q != r; ++q) {
77      T x = distribution(random_number_generator);
78      T y = distribution(random_number_generator);
79      *q = P(x, y);
80    }
81  }
82
83  #elif defined(SPECIAL)
84
85  template<typename I>
86  void generate(I p, I r) {
87    assert(std::distance(p, r) >= 4);
88    using P = typename std::iterator_traits<I>::value_type;
89    P origin(0, 0);
90    P west(-1, 0);
91    P north(0, 1);
92    P east(1, 0);
93    P south(0, -1);
94    *p = west;
95    ++p;
96    *p = north;
97    ++p;
98    *p = east;
99    ++p;
100   *p = south;
101   ++p;
102   for (I q = p; q != r; ++q) {
103     *q = origin;
104   }
105 }
106
107 #elif defined(PARABOLA)
108
109 template<typename I>
110 void generate(I p, I r) {
111   using P = typename std::iterator_traits<I>::value_type;
112   using T = typename P::coordinate;
113   T j = 0;
114   T modulus = 1 << 15;
115   for (I q = p; q != r; ++q) {
116     *q = P(j, j * j);
117     j = (j + 1) % modulus;
118   }
119   std::shuffle(p, r, random_number_generator);
120 }
121
122 #elif defined(SORTED)
123
```

```
124 template<typename I>
125 void generate(I p, I r) {
126   using P = typename std::iterator_traits<I>::value_type;
127   using T = int;
128   T min = std::numeric_limits<T>::min();
129   T max = std::numeric_limits<T>::max();
130   std::uniform_int_distribution<T> distribution(min, max);
131
132   for (I q = p; q != r; ++q) {
133     T x = distribution(random_number_generator);
134     T y = distribution(random_number_generator);
135     *q = P(x, y);
136   }
137   std::sort(p, r,
138     [](P const& p, P const& q) -> bool {
139       return p.x < q.x;
140     });
141 }
142
143 #elif defined(BELL)
144
145 template<typename I>
146 void generate(I p, I r) {
147   using P = typename std::iterator_traits<I>::value_type;
148   using T = typename P::coordinate;
149   using D = typename std::iterator_traits<I>::difference_type;
150   D n = std::distance(p, r);
151   using R = double;
152   R radius = R(std::numeric_limits<T>::max());
153   R mean = 0.0;
154   R stddev = radius / (2 + std::log(n));
155   std::normal_distribution<R> distribution(mean, stddev);
156
157   for (I q = p; q != r; ++q) {
158     T x = std::round(distribution(random_number_generator));
159     T y = std::round(distribution(random_number_generator));
160     *q = P(x, y);
161   }
162 }
163
164 #else // default SQUARE
165
166 template<typename I>
167 void generate(I p, I r) {
168   using P = typename std::iterator_traits<I>::value_type;
169   using T = int; // typename P::coordinate;
170   T min = std::numeric_limits<T>::min();
171   T max = std::numeric_limits<T>::max();
172   std::uniform_int_distribution<T> distribution(min, max);
173
174   for (I q = p; q != r; ++q) {
175     T x = distribution(random_number_generator);
```

```
176       T y = distribution(random_number_generator);
177       *q = P(x, y);
178     }
179   }
180
181   #endif
182
183   void usage(char const* program) {
184     std::cerr << "Usage: " << program << " <n>\n";
185     exit(1);
186   }
187
188   #if defined(MEASURE_COMPS) or defined(MEASURE_MOVES)
189
190   template<typename T>
191   class counter {
192   private:
193
194     T datum;
195
196   public:
197
198     static constexpr bool is_signed = true;
199     static constexpr bool is_integer = true;
200     static constexpr bool is_exact = true;
201     static constexpr bool is_bounded = true;
202     static constexpr bool is_modulo = not is_signed;
203     static constexpr std::size_t radix = 2;
204     static constexpr std::size_t length = 1;
205     static constexpr std::size_t width = 8 * sizeof(T);
206     static constexpr T min = std::numeric_limits<T>::min();
207     static constexpr T max = std::numeric_limits<T>::max();
208
209     explicit counter()
210       : datum(0) {
211       moves += 1;
212     }
213
214     counter(int x)
215       : datum(x) {
216       moves += 1;
217     }
218
219     counter(counter const& other) :
220       datum(other.datum) {
221       moves += 1;
222     }
223
224     template<typename number>
225     explicit counter(number x = 0)
226       : datum(x) {
227       moves += 1;
```

```
228     }
229
230     counter(counter&& other) {
231       datum = std::move(other.datum);
232       moves += 1;
233     }
234
235     counter& operator=(counter const& other) {
236       datum = other.datum;
237       moves += 1;
238       return *this;
239     }
240
241     counter& operator=(counter&& other) {
242       datum = std::move(other.datum);
243       moves += 1;
244       return *this;
245     }
246
247     operator T() const {
248       return datum;
249     }
250
251     template<typename U>
252     friend bool operator==(counter<U> const&, counter<U>
          ↪ const&);
253
254     template<typename U>
255     friend bool operator!=(counter<U> const&, counter<U>
          ↪ const&);
256
257     template<typename U>
258     friend bool operator<(counter<U> const&, counter<U> const&);
259
260     template<typename U>
261     friend bool operator>(counter<U> const&, counter<U> const&);
262
263     template<typename U>
264     friend bool operator<=(counter<U> const&, counter<U> const&);
265
266     template<typename U>
267     friend bool operator>=(counter<U> const&, counter<U> const&);
268   };
269
270   template<typename T>
271   bool operator==(counter<T> const& x, counter<T> const& y) {
272     ++comps;
273     return x.datum == y.datum;
274   }
275
276   template<typename T>
277   bool operator!=(counter<T> const& x, counter<T> const& y) {
```

```
278   ++comps;
279   return x.datum ≠ y.datum;
280 }
281
282 template<typename T>
283 bool operator<(counter<T> const& x, counter<T> const& y) {
284   ++comps;
285   return x.datum < y.datum;
286 }
287
288 template<typename T>
289 bool operator>(counter<T> const& x, counter<T> const& y) {
290   ++comps;
291   return x.datum > y.datum;
292 }
293
294 template<typename T>
295 bool operator≤(counter<T> const& x, counter<T> const& y) {
296   ++comps;
297   return x.datum ≤ y.datum;
298 }
299
300 template<typename T>
301 bool operator≥(counter<T> const& x, counter<T> const& y) {
302   ++comps;
303   return x.datum ≥ y.datum;
304 }
305
306 #endif
307
308 int main(int argc, char** argv) {
309   unsigned long n = 15;
310   if (argc == 2) {
311     n = std::atoi(argv[1]);
312   }
313   else {
314     usage(argv[0]);
315   }
316
317   unsigned long repetitions = maxsize / n;
318   if (n > maxsize) {
319     repetitions = 1;
320     maxsize = n;
321   }
322
323 #if defined(MEASURE_MOVES) or defined(MEASURE_COMPS)
324
325   using P = point<counter<int>>;
326
327 #else
328
329   using P = point<int>;
```

```
330
331 #endif
332
333   using I = P*;
334   I a = new P[maxsize];
335   I b = a;
336   for (volatile unsigned long t = 0; t != repetitions; ++t) {
337     generate(b, b + n);
338     b = b + n;
339   }
340
341   b = a;
342
343 #if defined(MEASURE_TURNS)
344
345   turns = 0;
346
347 #elif defined(MEASURE_MOVES)
348
349   moves = 0;
350
351 #elif defined(MEASURE_COMPS)
352
353   comps = 0;
354
355 #elif defined(PRUNING)
356
357   pruning_efficiency = 0;
358
359 #else
360
361   std::clock_t start = std::clock();
362
363 #endif
364
365   for (volatile unsigned long t = 0; t != repetitions; ++t)  {
366     (void) NAME::solve(&b[0], &b[n]);
367     b = b + n;
368   }
369
370 #if defined(MEASURE_TURNS)
371
372   double t = double(repetitions) * double(n);
373   std::cout.precision(3);
374   std::cout << n << "\t" << double(turns) / t << "\n";
375
376 #elif defined(MEASURE_MOVES)
377
378   double t = double(repetitions) * double(n);
379   std::cout.precision(3);
380   std::cout << n << "\t" << double(moves) / t << "\n";
381
```

```
382 #elif defined(MEASURE_COMPS)
383
384   double t = double(repetitions) * double(n);
385   std::cout.precision(3);
386   std::cout << n << "\t" << double(comps) / t << "\n";
387
388 #elif defined(PRUNING)
389
390   double t = double(repetitions) * double(n);
391   std::cout.precision(3);
392   std::cout << n << "\tpruning: " << double(pruning_efficiency) *
        ↪ 100.0 / t << "\n";
393
394 #else
395
396   std::clock_t stop = std::clock();
397
398   double t = double(repetitions) * double(n);
399   double ns = 1000000000.0 * double(stop - start) /
        ↪ double(CLOCKS_PER_SEC);
400   std::cout.precision(4);
401   std::cout << n << "\t" << ns / t << "\n";
402
403 #endif
404
405   delete[] a;
406   return 0;
407 }
```

F. Makefile

F.1 makefile

```
CXX=g++
CXXFLAGS=-O3 -std=c++17 -x c++ -Wall -Wextra -fconcepts -DNDEBUG
IFLAGS = -I..

header-files:= $(wildcard *.h++)
implementations:= $(basename $(header-files))
square-tests:= $(addsuffix .square, $(implementations))
disc-tests:= $(addsuffix .disc, $(implementations))
universe-tests:= $(addsuffix .universe, $(implementations))
bell-tests:= $(addsuffix .bell, $(implementations))
special-tests:= $(addsuffix .special, $(implementations))
sorted-tests:= $(addsuffix .sorted, $(implementations))
parabola-tests:= $(addsuffix .parabola, $(implementations))
turn-tests:= $(addsuffix .turn, $(implementations))
comp-tests:= $(addsuffix .comp, $(implementations))
move-tests:= $(addsuffix .move, $(implementations))
sanitychecks:= $(addsuffix .check, $(implementations))
unittests:= $(addsuffix .test, $(implementations))
benchmarks:= $(addsuffix .benchmark, $(implementations))

.PHONY: all clean find pilot veryclean

N = 1024 32768 1048576 33554432 1073741824

$(square-tests): %.square : %.h++
        @cp $*.h++ algorithm.h++
        $(CXX) $(CXXFLAGS) $(IFLAGS) -DNAME=$* driver.c++
        @for n in $(N) ; do \
          ./a.out $$n ; \
        done; \
        rm -f algorithm.h++ ./a.out

$(disc-tests): %.disc : %.h++
        @cp $*.h++ algorithm.h++
        $(CXX) $(CXXFLAGS) $(IFLAGS) -DDISC -DNAME=$* driver.c++
        @for n in $(N) ; do \
          ./a.out $$n ; \
        done; \
        rm -f algorithm.h++ ./a.out

$(bell-tests): %.bell : %.h++
        @cp $*.h++ algorithm.h++
        $(CXX) $(CXXFLAGS) $(IFLAGS) -DBELL -DNAME=$* driver.c++
        @for n in $(N) ; do \
          ./a.out $$n ; \
        done; \
        rm -f algorithm.h++ ./a.out

$(universe-tests): %.universe : %.h++
        @cp $*.h++ algorithm.h++
        $(CXX) $(CXXFLAGS) $(IFLAGS) -DUNIVERSE -DNAME=$* driver.c++
        @for n in $(N) ; do \
          ./a.out $$n ; \
        done; \
        rm -f algorithm.h++ ./a.out

$(special-tests): %.special : %.h++
        @cp $*.h++ algorithm.h++
        $(CXX) $(CXXFLAGS) $(IFLAGS) -DSPECIAL -DNAME=$* driver.c++
        @for n in $(N) ; do \
```

```
61            ./a.out $$n ; \
62          done; \
63          rm -f algorithm.h++ ./a.out
64
65  $(sorted-tests): %.sorted : %.h++
66          @cp $*.h++ algorithm.h++
67          $(CXX) $(CXXFLAGS) $(IFLAGS) -DSORTED -DNAME=$* driver.c++
68          @for n in $(N) ; do \
69            ./a.out $$n ; \
70          done; \
71          rm -f algorithm.h++ ./a.out
72
73  $(parabola-tests): %.parabola : %.h++
74          @cp $*.h++ algorithm.h++
75          $(CXX) $(CXXFLAGS) $(IFLAGS) -DPARABOLA -DNAME=$* driver.c++
76          @for n in $(N) ; do \
77            ./a.out $$n ; \
78          done; \
79          rm -f algorithm.h++ ./a.out
80
81  $(turn-tests): %.turn : %.h++
82          @cp $*.h++ algorithm.h++
83          $(CXX) $(CXXFLAGS) $(IFLAGS) -DNAME=$* -DMEASURE_TURNS driver.c++
84          @for n in $(N) ; do \
85            ./a.out $$n ; \
86          done; \
87          rm -f algorithm.h++ ./a.out
88
89  $(comp-tests): %.comp : %.h++
90          @cp $*.h++ algorithm.h++
91          $(CXX) $(CXXFLAGS) $(IFLAGS) -DNAME=$* -DMEASURE_COMPS driver.c++
92          @for n in $(N) ; do \
93            ./a.out $$n ; \
94          done; \
95          rm -f algorithm.h++ ./a.out
96
97  $(move-tests): %.move : %.h++
98          @cp $*.h++ algorithm.h++
99          $(CXX) $(CXXFLAGS) $(IFLAGS) -DNAME=$* -DMEASURE_MOVES driver.c++
100         @for n in $(N) ; do \
101           ./a.out $$n ; \
102         done; \
103         rm -f algorithm.h++ ./a.out
104
105 $(benchmarks): %.benchmark : %.h++
106         @date 2>&1 | tee -a $*.log
107         @make -i --no-print-directory $*.turn 2>&1 | tee -a $*.log
108         @make -i --no-print-directory $*.comp 2>&1 | tee -a $*.log
109         @make -i --no-print-directory $*.move 2>&1 | tee -a $*.log
110         @make -i --no-print-directory $*.square 2>&1 | tee -a $*.log
111         @make -i --no-print-directory $*.disc 2>&1 | tee -a $*.log
112         @make -i --no-print-directory $*.bell 2>&1 | tee -a $*.log
113         @date 2>&1 | tee -a $*.log
114
115 sea:
116         @make -i --no-print-directory plane_sweep.benchmark
117         @make -i --no-print-directory divide_and_conquer.benchmark
118         @make -i --no-print-directory quickhull.benchmark
119         @make -i --no-print-directory bucketing.benchmark
120         @make -i --no-print-directory throw_away.benchmark
121
122 cad:
123         @make -i --no-print-directory sort.benchmark
124         @make -i --no-print-directory plane_sweep.benchmark
125         @make -i --no-print-directory torch.benchmark
```

```
126         @make -i --no-print-directory quickhull.benchmark
127         @make -i --no-print-directory throw_away.benchmark
128
129 TESTFLAGS=-O3 -std=c++17 -Wall -Wextra -x c++ -fconcepts -g -DDEBUG
130
131 $(sanitychecks): %.check : %.h++
132         @cp $*.h++ algorithm.h++
133         $(CXX) $(TESTFLAGS) $(IFLAGS) -DNAME=$* check-driver.c++
134         ./a.out
135         rm -f ./a.out
136
137 $(unittests): %.test : %.h++
138         @cp $*.h++ algorithm.h++
139         $(CXX) $(TESTFLAGS) $(IFLAGS) -DNAME=$* test-driver.c++
140         ./a.out
141         rm -f algorithm.h++ ./a.out
142
143 # Other tools
144
145 clean:
146         - rm  -f a.out temp algorithm.h++ 2>/dev/null
147
148 veryclean: clean
149         - rm -f *~ .*~ */*~ 2>/dev/null
150
151 find:
152         find . -type f -print -exec grep $(word) {} \; | less
```

References

[1] H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. Toussaint, Space-efficient planar convex hull algorithms, *Theoret. Comput. Sci.* **321**, 1 (2004), 25–40.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd edition, The MIT Press (2009).

[3] A. N. Gamby, Expected linear-time convex-hull algorithms: Investigation into possible improvements of the running times in the planar case, M. Sc. thesis, Dept. Comput. Sci., Univ. Copenhagen (2017).

[4] A. N. Gamby and J. Katajainen, A note on the implementation quality of a convex-hull algorithm, CPH STL report **2017-3**, Dept. Comput. Sci., Univ. Copenhagen (2017).

[5] A. N. Gamby and J. Katajainen, *Convex-hull algorithms: Implementation, testing, and experimentation*, Submitted (2018).

[6] A. N. Gamby and J. Katajainen, *Evaluating the heuristics for finding convex hulls in the plane*, Submitted (2018).

[7] J. Katajainen, Pure compile-time functions and classes in the CPH MPL, CPH STL report **2017-2**, Dept. Comput. Sci., Univ. Copenhagen (2017).

[8] J. Katajainen, Width-specific templates $\mathbb{N}$<b> and $\mathbb{Z}$<b> for integer types in C++: Going beyond C, CPH STL report **2017-1**, Department of Computer Science, University of Copenhagen (2017).