Supplement: Relevant source code

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 to 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 report, together with an accompanying `zip` file, contains the source code of the programs described and benchmarked in the paper "Convex-hull algorithms: Implementation, testing, and experimentation". The implementations of the following convex-hull algorithms are included: PLANE-SWEEP, TORCH, QUICKHULL, POLES-FIRST, THROW-AWAY, and INTROHULL. These programs describe some of the state of the art in convex-hull finding in 2018.

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

Released August 2018; Revised October 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 tried to implement the included algorithms in the best possible way, but 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. Still, there are certainly 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 report, 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 considered algorithms 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 [3] and the CPH MPL [4]. 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 paper "Convex-hull algorithms: Implementation, testing, and experimentation". 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.line`. Run the CPU benchmark for a multiset of random points on a line.

`quickhull.parabola`. Run the CPU benchmark for a multiset of random points on a parabola.

`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 the most relevant experiments for QUICKHULL in one go. For algorithm X, the results will be placed in 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     56.31
32768    73.05
1048576  90.77
33554432         109.1
```

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
```

A. Copyright notice

B. Use case

B.1 use-case.c++

```
#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";
}
```

C. Convex-hull algorithms

C.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 <utility> // std::pair std::get ...
#include "validation.h++" // same_multiset convex_polygon all_inside

namespace plane_sweep {

  // 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;
  }

  template<typename I>
  I partition_upper_lower(I pole, I antipole, I first, I past) {
    using P = typename std::iterator_traits<I>::value_type;
    using T = typename P::coordinate;
    T upper_limit = std::max((*pole).y, (*antipole).y);
    T lower_limit = std::min((*pole).y, (*antipole).y);
```

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

```
101     *source = p;
102     return target;
103   }
104
105   template<typename I>
106   I scan(I first, I top, I next, I past) {
107     assert(first != past);
108     for (I i = next; i != past; ++i) {
109       while (top != first and not right_turn(*(top - 1), *top, *i))
      ↪ {
110         --top;
111       }
112       ++top;
113       std::iter_swap(i, top);
114     }
115     return ++top;
116   }
117
118   template<typename I>
119   I scan_one_more(I first, I top, I extra) {
120     while (top != first and not right_turn(*(top - 1), *top,
      ↪ *extra)) {
121       --top;
122     }
123     return ++top;
124   }
125
126   template<typename I, typename C>
127   I chain(I pole, I rest, I antipole, C compare) {
128     using P = typename std::iterator_traits<I>::value_type;
129     if (std::distance(pole, rest) == 1) {
130       return rest;
131     }
132     std::sort(pole + 1, rest,
133       [&](P const& p, P const& q) -> bool {
134         return compare(p.x, q.x);
135       });
136     std::pair<I, I> pair = clean(pole, rest, compare);
137     I top = std::get<0>(pair);
138     I next = std::get<1>(pair);
139     I interior = scan(pole, top, next, rest);
140     interior = scan_one_more(pole, interior - 1, antipole);
141     return interior;
142   }
143
144   template<typename I>
145   I solve(I first, I past) {
146     using P = typename std::iterator_traits<I>::value_type;
147     using T = typename P::coordinate;
148     std::size_t n = past - first;
149     if (n < 2) {
150       return past;
```

```
151       }
152       std::pair<I, I> pair = find_poles(first, past);
153       I west = first;
154       I east = past - 1;
155       parallel_iter_swap(west, east, std::get<0>(pair),
          ↪ std::get<1>(pair));
156       if (*east == *west) {
157         return first + 1;
158       }
159       I middle = partition_upper_lower(west, east, first + 1, past -
          ↪ 1);
160       I rest_1 = chain(west, middle, east, std::less<T>());
161       std::iter_swap(middle, east);
162       east = middle;
163       I rest_2 = chain(east, past, west, std::greater<T>());
164       I interior = swap_blocks(east, rest_2, rest_1);
165       return interior;
166     }
167
168     template<typename I>
169     bool check(I first, I past) {
170       using P = typename std::iterator_traits<I>::value_type;
171       using S = std::vector<P>;
172       using J = typename S::iterator;
173       S data;
174       std::size_t n = past - first;
175       data.resize(n);
176       std::copy(first, past, data.begin());
177       J rest = solve(data.begin(), data.end());
178       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);
179       return ok;
180     }
181   }
182
183   #endif
```

C.2 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
```

```cpp
#include <iterator> // std::iterator_traits
#include "point.h++" // right_turn
#include <utility> // std::pair std::get
#include "validation.h++" // same_multiset convex_polygon all_inside

namespace torch {

  // 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>
  I go_upstairs(I first, I past) {
    if (first == past) {
      return past;
    }
    using P = typename std::iterator_traits<I>::value_type;
    using T = typename P::coordinate;
    T max = (*first).y;
    I i = first + 1;
    for (I j = first + 1; j != past; ++j) {
      if ((*j).y >= max) {
        max = (*j).y;
        std::iter_swap(i, j);
        ++i;
      }
    }
    return i;
  }

  template<typename I>
  I go_downstairs(I first, I past) {
    if (first == past) {
      return past;
    }
    using P = typename std::iterator_traits<I>::value_type;
    using T = typename P::coordinate;
    T min = (*first).y;
    I i = first + 1;
    for (I j = first + 1; j != past; ++j) {
      if ((*j).y <= min) {
        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
140     bool east_north_same = (*east == *north);
141     bool west_south_same = (*west == *south);
142     bool west_north_same = (*west == *north);
143     bool east_south_same = (*east == *south);
144
145     // recall the quadrant borders
146
147     T west_y = (*west).y;
148     T east_y = (*east).y;
149     T north_x = (*north).x;
150     T south_x = (*south).x;
151
152     // north–west corner
153
154     I done = west + 1;
155     if (not west_north_same) {
156       I rest = std::partition(done, last,
157         [&](P const& a) -> bool {
158           return (a.x ≤ north_x) and (a.y ≥ west_y);
159         });
160       if (rest ≠ done) {
161         std::sort(done, rest,
162           [](P const& a, P const& b) -> bool {
163             return (a.x < b.x);
164           });
```

```
165         rest = go_upstairs(west, rest);
166         rest = scan(west, rest);
167         done = scan_one_more(west, rest - 1, east);
168       }
169     }
170
171     // move east to Q2
172
173     std::iter_swap(done, east);
174     east = done;
175     done = done + 1;
176
177     // north–east corner
178
179     if (not east_north_same) {
180       I top = east - 1;
181       I rest = std::partition(done, past,
182         [&](P const& a) → bool {
183           return (a.x ≥ north_x) and (a.y ≥ east_y);
184         });
185       if (rest ≠ done) {
186         std::iter_swap(top, east);
187         std::swap(top, east);
188         std::iter_swap(top, rest - 1);
189         top = rest - 1;
190         std::sort(east + 1, top,
191           [](P const& a, P const& b) → bool {
192             return (a.x > b.x);
193           });
194         rest = go_upstairs(east, rest);
195         std::reverse(east, rest);
196         top = east;
197         done = scan(top, rest);
198         east = done - 1;
199       }
200     }
201
202     // south–east corner
203
204     if (not east_south_same) {
205       I rest = std::partition(done, past,
206         [&](P const& a) → bool {
207           return (a.x ≥ south_x) and (a.y ≤ east_y);
208         });
209       if (rest ≠ done) {
210         std::sort(done, rest,
211           [](P const& a, P const& b) → bool {
212             return (a.x > b.x);
213           });
214         rest = go_downstairs(east, rest);
215         rest = scan(east, rest);
216         done = scan_one_more(east, rest - 1, west);
```

```
217         }
218       }
219
220       // south–west corner
221
222       if (not west_south_same) {
223         I bottom = done - 1;
224         I rest = std::partition(done, past,
225           [&](P const& a) -> bool {
226             return (a.x <= south_x) and (a.y <= west_y);
227           });
228         if (rest != done) {
229           std::iter_swap(bottom, rest - 1);
230           std::sort(bottom, rest - 1,
231             [](P const& a, P const& b) -> bool {
232               return (a.x < b.x);
233             });
234           rest = go_downstairs(bottom, rest);
235           std::reverse(bottom, rest);
236           rest = scan(bottom, rest);
237           done = scan_one_more(bottom, rest - 1, west);
238         }
239       }
240       return done;
241     }
242
243     template<typename I>
244     bool check(I first, I past) {
245       using P = typename std::iterator_traits<I>::value_type;
246       using S = std::vector<P>;
247       using J = typename S::iterator;
248       S data;
249       std::size_t n = past - first;
250       data.resize(n);
251       std::copy(first, past, data.begin());
252       J rest = solve(data.begin(), data.end());
253       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);
254       return ok;
255     }
256   }
257
258   #endif
```

C.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   // move *st <- *rd and *nd <- *th by swaps in parallel
21
22   template<typename I>
23   void parallel_iter_swap(I st, I nd, I rd, I th) {
24     assert(st != nd and rd != th);
25     std::iter_swap(st, rd);
26     if (th == st) {
27       std::iter_swap(nd, rd);
28     }
29     else {
30       std::iter_swap(nd, th);
31     }
32   }
33
34   template<typename I>
35   std::pair<I, I> find_poles(I first, I past) {
36     using P = typename std::iterator_traits<I>::value_type;
37     auto pair = std::minmax_element(first, past,
38       [](P const& a, P const& b) -> bool {
39         return (a.x < b.x) or (a.x == b.x and a.y < b.y);
40       });
41     return pair;
42   }
43
44   template<typename I>
45   I find_furthest(I first, I past, I antipole) {
46     assert(first != past);
47     I pole = first;
48     I answer = pole;
49     using U = decltype(signed_area(*pole, *pole, *pole));
50     U best = 0;
51     if ((*antipole).x == (*pole).x) { // vertical
52       for (I i = first + 1; i != past; ++i) {
53         U Φ = signed_area(*pole, *antipole, *i);
54         if (Φ > best or (Φ == best and (*i).y < (*answer).y)) {
55           answer = i;
```

```
56            best = Φ;
57          }
58        }
59      }
60      else {
61        for (I i = first + 1; i ≠ past; ++i) {
62          U Φ = signed_area(*pole, *antipole, *i);
63          if (Φ > best or (Φ == best and (*i).x < (*answer).x)) {
64            answer = i;
65            best = Φ;
66          }
67        }
68      }
69      return answer;
70    }
71
72    template<typename I>
73    I partition_left_right(I first, I past, I antipole) {
74      assert(first ≠ past);
75      using P = typename std::iterator_traits<I>::value_type;
76      I pole = first;
77      I middle = std::partition(pole + 1, past,
78        [&](P const& q) → bool {
79          return not left_turn(*pole, q, *antipole);
80        });
81      return middle;
82    }
83
84    template <typename I>
85    void swap_blocks(I source, I past_the_end, I target) {
86      if (source == target or source == past_the_end) {
87        return;
88      }
89      using P = typename std::iterator_traits<I>::value_type;
90      I hole = target;
91      P p = *target;
92      I const last = past_the_end - 1;
93      while (true) {
94        *hole = *source;
95        ++hole;
96        if (source == last) {
97          break;
98        }
99        *source = *hole;
100       ++source;
101     }
102     *source = p;
103   }
104
105   template <typename I>
106   void move_away(I here, I rest, I past) {
107     if (here == rest or rest == past) {
```

```
108       return;
109     }
110     if (rest - here < past - rest) {
111       swap_blocks(here, rest, past - (rest - here));
112     }
113     else {
114       swap_blocks(rest, past, here);
115     }
116   }
117
118   template<typename I>
119   I recurse(I pole, I past, I antipole) {
120     std::size_t n = std::distance(pole, past);
121     if (n == 1) {
122       return past;
123     }
124     if (n == 2) {
125       if (no_turn(*(pole + 1), *pole, *antipole)) {
126         return pole + 1;
127       }
128       else {
129         return past;
130       }
131     }
132     I pivot = find_furthest(pole, past, antipole);
133     if (no_turn(*pivot, *pole, *antipole)) {
134       return pole + 1;
135     }
136     I last = past - 1;
137     std::iter_swap(pivot, last); // pivot at the end
138     I mid = partition_left_right(pole, last, last);
139     I eliminated = recurse(pole, mid, last);
140     std::iter_swap(mid, last);
141     std::iter_swap(eliminated, mid); // pivot at its final place
142     pivot = eliminated;
143     std::size_t m = past - mid;
144     ++mid;
145     ++eliminated;
146     move_away(eliminated, mid, past);
147     I interior = partition_left_right(pivot, pivot + m, antipole);
148     eliminated = recurse(pivot, interior, antipole);
149     return eliminated;
150   }
151
152   template<typename I>
153   I solve(I first, I past) {
154     std::size_t n = past - first;
155     if (n < 2 or (n == 2 and *first ≠ *(first + 1))) {
156       return past;
157     }
158     std::pair<I, I> pair = find_poles(first, past);
159     I west = first;
```

```
160       I east = past - 1;
161       parallel_iter_swap(west, east, std::get<0>(pair),
          ↪ std::get<1>(pair));
162       if (*west == *east) {
163         return first + 1;
164       }
165       I middle = partition_left_right(west, east, east);
166       std::size_t m = past - middle;
167       I eliminated = recurse(first, middle, east);
168       std::iter_swap(middle, east);
169       std::iter_swap(eliminated, middle); // east at its final place
170       east = eliminated;
171       ++middle;
172       ++eliminated;
173       move_away(eliminated, middle, past);
174       eliminated = recurse(east, east + m, west); // downunder
175       return eliminated;
176     }
177
178     template<typename I>
179     bool check(I first, I past) {
180       using P = typename std::iterator_traits<I>::value_type;
181       using S = std::vector<P>;
182       using J = typename S::iterator;
183       S data;
184       std::size_t n = past - first;
185       data.resize(n);
186       std::copy(first, past, data.begin());
187       J rest = solve(data.begin(), data.end());
188       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);
189       return ok;
190     }
191   }
192
193   #endif
```

C.4 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.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) {
```

```
#if defined(MEASURE_MOVES)

      ++moves;

#endif
        nw = dot;
        max_position[7] = i;
      }
    }
    return max_position;
  }

  template<typename S>
  void remove_duplicates(S& sequence) {
    assert(sequence.size() != 0);
    assert(std::is_sorted(sequence.begin(), sequence.end()));
    std::size_t i = 0;
    std::size_t j = 1;
    auto v = sequence[0];
    while (j != sequence.size()) {
      if (not (sequence[j] == v)) {
        std::swap(sequence[i], v);
        ++i;
        v = sequence[j];
      }
      ++j;
    }
    std::swap(sequence[i], v);
    ++i;
    std::size_t leftovers = sequence.size() - i;
    while (leftovers != 0) {
      sequence.pop_back();
      --leftovers;
    }
  }

  template<typename P, typename I>
  bool inside_convex_polygon(P const& p, I upper, I lower, I past) {
    assert(past - upper > 2);
    assert(*upper == *(past - 1));
    assert(*upper < *(upper + 1));
    assert(*upper < *(past - 2));
    I i = upper;
    do {
      ++i;
    } while (i != lower and *i < p);
    if (left_turn(*(i - 1), *i, p)) {
      return false;
    }
    I last = past - 1;
    I j = lower;
    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
```

C.6 introhull.h++

```
1  /*
2    Performance Engineering Laboratory © 2017–2018
```

```
3
4     Execute the some levels of quickhull and process the remaining
5     regions using the plane-sweep algorithm
6   */
7
8   #ifndef __INTROHULL__
9   #define __INTROHULL__
10
11  #include <algorithm> // std::sort std::partition ...
12  #include <cassert> // assert macro
13  #include <cstdlib> // std::size_t
14  #include <functional> // std::less
15  #include <iterator> // std::iterator_traits
16  #include "plane_sweep.h++" // plane_sweep::solve
17  #include "point.h++" // left_turn
18  #include <utility> // std::pair std::get ...
19  #include "validation.h++" // same_multiset convex_polygon all_inside
20  #include <vector> // std::vector
21
22  namespace introhull {
23
24    template<typename N>
25    inline N lg(N n) {
26      N k;
27      for (k = 0; n != 0; n >>= 1) {
28        ++k;
29      }
30      return k - 1;
31    }
32
33    inline int lg(int n) {
34      using N = std::size_t;
35      constexpr N  char_bit = std::numeric_limits<unsigned
        ↪ char>::digits;
36      return sizeof(int) * char_bit  - 1 - __builtin_clz(n);
37    }
38
39    inline long lg(long n) {
40      using N = std::size_t;
41      constexpr N char_bit = std::numeric_limits<unsigned
        ↪ char>::digits;
42      return sizeof(long) * char_bit - 1 - __builtin_clzl(n);
43    }
44
45    inline long long lg(long long n) {
46      using N = std::size_t;
47      constexpr N char_bit = std::numeric_limits<unsigned
        ↪ char>::digits;
48      return sizeof(long long) * char_bit - 1 - __builtin_clzll(n);
49    }
50
51    // move *st <- *rd and *nd <- *th by swaps in parallel
```

```
52
53  template<typename I>
54  void parallel_iter_swap(I st, I nd, I rd, I th) {
55    assert(st ≠ nd and rd ≠ th);
56    std::iter_swap(st, rd);
57    if (th == st) {
58      std::iter_swap(nd, rd);
59    }
60    else {
61      std::iter_swap(nd, th);
62    }
63  }
64
65  template<typename I>
66  std::pair<I, I> find_poles(I first, I past) {
67    using P = typename std::iterator_traits<I>::value_type;
68    auto pair = std::minmax_element(first, past,
69      [](P const& a, P const& b) → bool {
70        return (a.x < b.x) or (a.x == b.x and a.y < b.y);
71      });
72    return pair;
73  }
74
75  template<typename I>
76  I find_furthest(I first, I past, I east) {
77    assert(first ≠ past);
78    I west = first;
79    I answer = west;
80    using U = decltype(signed_area(*west, *west, *west));
81    U best = 0;
82    if ((*east).x == (*west).x) { // vertical
83      for (I i = first + 1; i ≠ past; ++i) {
84        U Φ = signed_area(*west, *east, *i);
85        if (Φ > best or (Φ == best and (*i).y < (*answer).y)) {
86          answer = i;
87          best = Φ;
88        }
89      }
90    }
91    else {
92      for (I i = first + 1; i ≠ past; ++i) {
93        U Φ = signed_area(*west, *east, *i);
94        if (Φ > best or (Φ == best and (*i).x < (*answer).x)) {
95          answer = i;
96          best = Φ;
97        }
98      }
99    }
100   return answer;
101 }
102
103 template<typename I>
```

```
104  I partition_left_right(I first, I past, I east) {
105    assert(first ≠ past);
106    using P = typename std::iterator_traits<I>::value_type;
107    I west = first;
108    I middle = std::partition(west + 1, past,
109      [&](P const& q) → bool {
110        return not left_turn(*west, q, *east);
111      });
112    return middle;
113  }
114
115  template <typename I>
116  void swap_blocks(I source, I past_source, I target) {
117    using P = typename std::iterator_traits<I>::value_type;
118    I hole = target;
119    P p = *target;
120    I const last = past_source − 1;
121    while (true) {
122      *hole = *source;
123      ++hole;
124      if (source == last) {
125        break;
126      }
127      *source = *hole;
128      ++source;
129    }
130    *source = p;
131  }
132
133  template <typename I>
134  void move_away(I here, I rest, I past) {
135    if (here == rest or rest == past) {
136      return;
137    }
138    if (rest − here < past − rest) {
139      swap_blocks(here, rest, past − (rest − here));
140    }
141    else {
142      swap_blocks(rest, past, here);
143    }
144  }
145
146  template<typename I, typename C>
147  I recurse(I pole, I past, I antipole, std::size_t depth, C
       ↪ compare) {
148    std::size_t n = std::distance(pole, past);
149    if (n == 1) {
150      return past;
151    }
152    if (n == 2) {
153      if (no_turn(*(pole + 1), *pole, *antipole)) {
154        return pole + 1;
```

```
155       }
156       else {
157         return past;
158       }
159     }
160     if (depth == 0) {
161       return plane_sweep::chain(pole, past, antipole, compare);
162     }
163     I pivot = find_furthest(pole, past, antipole);
164     if (no_turn(*pivot, *pole, *antipole)) {
165       return pole + 1;
166     }
167     I last = past - 1;
168     std::iter_swap(pivot, last); // pivot at the end
169     I mid = partition_left_right(pole, last, last);
170     I eliminated = recurse(pole, mid, last, depth - 1, compare);
171     std::iter_swap(mid, last);
172     std::iter_swap(eliminated, mid); // pivot at its final place
173     pivot = eliminated;
174     std::size_t m = past - mid;
175     ++mid;
176     ++eliminated;
177     move_away(eliminated, mid, past);
178     I interior = partition_left_right(pivot, pivot + m, antipole);
179     eliminated = recurse(pivot, interior, antipole, depth - 1,
      ↪ compare);
180     return eliminated;
181   }
182
183   template<typename I>
184   I solve(I first, I past) {
185     using P = typename std::iterator_traits<I>::value_type;
186     using T = typename P::coordinate;
187     std::size_t n = past - first;
188     if (n < 2 or (n == 2 and *first != *(first + 1))) {
189       return past;
190     }
191     std::size_t stop = lg(n);
192     std::pair<I, I> pair = find_poles(first, past);
193     I west = first;
194     I east = past - 1;
195     parallel_iter_swap(west, east, std::get<0>(pair),
      ↪ std::get<1>(pair));
196     if (*west == *east) {
197       return first + 1;
198     }
199     I middle = partition_left_right(west, east, east);
200     std::size_t m = past - middle;
201     I eliminated = recurse(first, middle, east, stop,
      ↪ std::less<T>());
202     std::iter_swap(middle, east);
203     std::iter_swap(eliminated, middle); // east at its final place
```

```
204       east = eliminated;
205       ++middle;
206       ++eliminated;
207       move_away(eliminated, middle, past);
208       I rest = recurse(east, east + m, west, stop,
          ↪ std::greater<T>());
209       return rest;
210     }
211
212     template<typename I>
213     bool check(I first, I past) {
214       using P = typename std::iterator_traits<I>::value_type;
215       using S = std::vector<P>;
216       using J = typename S::iterator;
217       S data;
218       std::size_t n = past - first;
219       data.resize(n);
220       std::copy(first, past, data.begin());
221       J rest = solve(data.begin(), data.end());
222       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);
223       return ok;
224     }
225   }
226
227   #endif
```

D. Micro-benchmarks

D.1 sort.h++

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

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

namespace sort {

  template<typename I>
  I solve(I first, I past) {
    using P = typename std::iterator_traits<I>::value_type;
    std::sort(first, past,
      [](P const& p, P const& q) {
        return p.x < q.x;
      });
    return past;
  }
}
```

D.2 lexicographic_sort.h++

```
/*
  This dummy sorts the points in ascending lexicographic order
*/

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

namespace lexicographic_sort {

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

D.3 angular_sort.h++

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

```
3     distance from o
4   */
5
6   #include <algorithm> // std::sort
7   #include <cstddef> // std::size_t
8   #include <iterator> // std::iterator_traits
9
10  #define MULTI_PRECISION
11
12  #include "point.h++"
13
14  namespace angular_sort {
15
16    template<typename I>
17    I solve(I first, I past) {
18      using P = typename std::iterator_traits<I>::value_type;
19      I west = std::min_element(first, past,
20        [](P const& p, P const& q) -> bool {
21          return (p.x < q.x) or (p.x == q.x and p.y < q.y);
22        });
23      P o = *west;
24      std::sort(first, past,
25        [&](P const& p, P const& q) -> bool {
26          int turn = orientation(o, p, q);
27          if (turn == 0) {
28            return (L_∞distance(o, p) < L_∞distance(o, q));
29          }
30          return (turn == +1);
31        });
32      return past;
33    }
34  }
```

E. Helpers

E.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 + 4>;
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      using N = std::size_t;
84      using T = typename P::coordinate;
85      constexpr N w = cphmpl::width<T>;
86      using Z = cphstl::ℤ<2 * w + 4>;
87      Z p_x = Z(p.x);
88      Z p_y = Z(p.y);
89      Z lhs = (Z(q.x) - p_x) * (Z(r.y) - p_y);
90      Z rhs = (Z(r.x) - p_x) * (Z(q.y) - p_y);
91      return lhs == rhs;
92    }
93
94    // compute the signed area of the parallelogram spanned by
95    // \vec{pq} and \vec{pr}.
96
97    template<typename P>
98    auto signed_area(P const& p, P const& q, P const& r) {
```

```
99      using N = std::size_t;
100     using T = typename P::coordinate;
101     constexpr N w = cphmpl::width<T>;
102     using Z = cphstl::ℤ<2 * w + 5>;
103     Z p_x = Z(p.x);
104     Z p_y = Z(p.y);
105     Z dx_1 = Z(q.x) - p_x;
106     Z dx_2 = Z(r.x) - p_x;
107     Z dy_1 = Z(q.y) - p_y;
108     Z dy_2 = Z(r.y) - p_y;
109     Z result = dx_1 * dy_2 - dx_2 * dy_1;
110     return result;
111   }
112
113   // return the square of the distance between p and q
114
115   template<typename P>
116   auto distance_squared(P const& p, P const& q) {
117     using N = std::size_t;
118     using T = typename P::coordinate;
119     constexpr N w = cphmpl::width<T>;
120     using Z = cphstl::ℤ<2 * w + 5>;
121     Z p_x = Z(p.x);
122     Z p_y = Z(p.y);
123     Z q_x = Z(q.x);
124     Z q_y = Z(q.y);
125     return (p_x - q_x) * (p_x - q_x) + (p_y - q_y) * (p_y - q_y);
126   }
127
128   template<typename P>
129   auto L_∞distance(P const& p, P const& q) {
130     using N = std::size_t;
131     using T = typename P::coordinate;
132     constexpr N w = cphmpl::width<T>;
133     using Z = cphstl::ℤ<w + 2>;
134     Z dx = cphstl::abs(Z(q.x) - Z(p.x));
135     Z dy = cphstl::abs(Z(q.y) - Z(p.y));
136     return std::max(dx, dy);
137   }
138
139   template<typename P>
140   auto L_1_distance(P const& p, P const& q) {
141     using N = std::size_t;
142     using T = typename P::coordinate;
143     constexpr N w = cphmpl::width<T>;
144     using Z = cphstl::ℤ<w + 3>;
145     Z dx = cphstl::abs(Z(q.x) - Z(p.x));
146     Z dy = cphstl::abs(Z(q.y) - Z(p.y));
147     return dx + dy;
148   }
149
150   // return the orientation when moving from p to r via q
```

```
151     // +1: left turn
152     //  0: p, q, and r are collinear
153     // -1: right turn
154
155     template<typename P>
156     int orientation(P const& p, P const& q, P const& r) {
157       using N = std::size_t;
158       using T = typename P::coordinate;
159       constexpr N w = cphmpl::width<T>;
160       using Z = cphstl::ℤ<2 * w + 4>;
161       Z p_x = Z(p.x);
162       Z p_y = Z(p.y);
163       Z lhs = (Z(q.x) - p_x) * (Z(r.y) - p_y);
164       Z rhs = (Z(r.x) - p_x) * (Z(q.y) - p_y);
165       if (lhs == rhs) {
166         return 0;
167       }
168       return (lhs > rhs) ? +1 : -1;
169     }
170   }
171
172   #endif
173
174   #if defined(MULTI_PRECISION)
175
176   using namespace multi_precision;
177
178   #endif
179
180   #if defined(DOUBLE_PRECISION)
181
182   namespace double_precision {
183
184     template<typename P>
185     bool left_turn(P const& p, P const& q, P const& r) {
186
187   #ifdef MEASURE_TURNS
188
189       ++turns;
190
191   #endif
192
193       using T = typename P::coordinate;
194       using Z = long long;
195       Z mid_x = Z(q.x);
196       Z mid_y = Z(q.y);
197       Z dx_1 = mid_x - Z(p.x);
198       Z dx_2 = Z(r.x) - mid_x;
199       Z dy_1 = mid_y - Z(p.y);
200       Z dy_2 = Z(r.y) - mid_y;
201
202       /* We have 31 bits + 1 for the sign in int, and
```

```
203         we may need (31 (+1 from minus)) * 2 (from mult) = 64 bits
204         in total, but we only have 63 bits + 1 for the sign.
205         If we overflow the same direction, the order is preserved. */
206       bool check1 = Z(T(dx1)) != dx1 and Z(T(dy2)) != dy2;
207       bool check2 = Z(T(dx2)) != dx2 and Z(T(dy1)) != dy1;
208       if (check1 or check2) {
209         // overflow happening
210         if (check1) {
211           if ((dx1 > 0) == (dy2 > 0)) {
212             // first part overflows above
213             if (check2 and ((dx2 > 0) == (dy1 > 0))) {
214               // both overflow above
215               return dx1 * dy2 > dx2 * dy1;
216             }
217             else {
218               return true; // only first term overflows above
219             }
220           }
221           else {
222             // first term overflows below
223             if (check2 and ((dx2 > 0) != (dy1 > 0))) {
224               // second term also overflows below
225               return dx1 * dy2 > dx2 * dy1;
226             }
227             else {
228               // only first term overflows below
229               return false;
230             }
231           }
232         }
233         else {
234           // second term overflows; result is inverse of the direction
235           return (dx2 > 0) != (dy1 > 0);
236         }
237         return dx1 * dy2 > dx2 * dy1;
238       }
239       else {
240         // standard setting
241         return dx1 * dy2 > dx2 * dy1;
242       }
243     }
244
245     template<typename P>
246     bool no_turn(P const& p, P const& q, P const& r) {
247
248   #ifdef MEASURE_TURNS
249
250       ++turns;
251
252   #endif
253
254       using T = typename P::coordinate;
```

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

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

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

E.2 validation.h++

```
#ifndef __VALIDATION__
#define __VALIDATION__

#include <algorithm> // std::copy std::sort std::equal std::rotate
#include <cassert> // assert macro
#include <cstdlib> // std::size_t
#include <iostream> // std streams
#include <iterator> // std::iterator_traits
#include "point.h++" // left_turn right_turn
#include <vector> // std::vector

namespace validation {

  template<typename I, typename J>
  bool same_multiset(I p, I q, J r, J s) {
    using P = typename std::iterator_traits<I>::value_type;
    using S = std::vector<P>;
    std::size_t n = q - p;
    std::size_t m = s - r;
    if (m != n) {
      return false;
    }
    S backup;
    backup.resize(n);
    std::copy(p, q, backup.begin());
    std::sort(backup.begin(), backup.end(),
      [](P const& a, P const& b) -> bool {
        return (a.x < b.x) or (a.x == b.x and a.y < b.y);
      });
    S other;
    other.resize(n);
    std::copy(r, s, other.begin());
    std::sort(other.begin(), other.end(),
      [](P const& a, P const& b) -> bool {
        return (a.x < b.x) or (a.x == b.x and a.y < b.y);
      });
    return std::equal(backup.begin(), backup.end(), other.begin(),
      [](P const& a, P const& b) -> bool {
        return (a.x == b.x) and (a.y == b.y);
      });
  }

  // [p, r) circular chain of vertices

  template<typename I>
  bool left_turns_only(I p, I r) {
    std::size_t h = r - p;
    if (h < 3) {
      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
```

F. Drivers

F.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()));
}
```

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

```
233   input.push_back(P(0, 0));
234   input.push_back(P(1, 0));
235   input.push_back(P(2, 0));
236   input.push_back(P(2, 0));
237   input.push_back(P(3, 0));
238
239   input.push_back(P(4, 1));
240   input.push_back(P(5, 2));
241   input.push_back(P(6, 3));
242   input.push_back(P(6, 3));
243   input.push_back(P(7, 2));
244   input.push_back(P(8, 1));
245
246   input.push_back(P(4, -1));
247   input.push_back(P(5, -2));
248   input.push_back(P(6, -3));
249   input.push_back(P(6, -3));
250   input.push_back(P(7, -2));
251   input.push_back(P(8, -1));
252
253   input.push_back(P(9, 0));
254   input.push_back(P(10, 0));
255   input.push_back(P(11, 0));
256   input.push_back(P(12, 0));
257   assert(f(input.begin(), input.end()));
258 }
259
260 template<typename checker>
261 void many_big_numbers(checker f) {
262   std::cout << "many big numbers\n";
263   using T = int;
264   using P = point<int>;
265   T max = std::numeric_limits<T>::max();
266   P origo(0, 0);
267   P left_bottom(-max, -max);
268   P just_above(-max, -max + 1);
269   P right_top(max, max);
270   P left_top(-max, max);
271   P neighbour_below(-max, max - 1);
272   P neighbour_beside(-max - 1, max);
273   using S = std::vector<P>;
274   S input;
275   input.push_back(left_bottom);
276   input.push_back(just_above);
277   input.push_back(origo);
278   input.push_back(left_top);
279   input.push_back(right_top);
280   input.push_back(neighbour_below);
281   input.push_back(neighbour_beside);
282   assert(f(input.begin(), input.end()));
283 }
284
```

```
285 template<typename checker>
286 void noisy_box(checker f) {
287   std::cout << "noisy box\n";
288   using T = int;
289   using P = point<T>;
290   T max = std::numeric_limits<T>::max();
291   using S = std::vector<P>;
292   using I = typename S::iterator;
293   constexpr std::size_t n = 10;
294   S input(n);
295   std::uniform_int_distribution<T> distribution(-max + 1, max - 1);
296   for (I q = input.begin(); q != input.end(); ++q) {
297     T x = distribution(random_number_generator);
298     T y = distribution(random_number_generator);
299     *q = P(x, y);
300   }
301   input.push_back(P(max, max));
302   input.push_back(P(max, -max));
303   input.push_back(P(-max, -max));
304   input.push_back(P(-max, max));
305   assert(f(input.begin(), input.end()));
306 }
307
308 template<typename checker>
309 void random_points_on_a_parabola(checker f) {
310   std::cout << "random points on a parabola\n";
311   using P = point<int>;
312   using S = std::vector<P>;
313   using I = typename S::iterator;
314   std::size_t const n = 4;
315   S input(n);
316   int j = 0;
317   for (I q = input.begin(); q != input.end(); ++q) {
318     *q = P(j, j * j);
319     ++j;
320   }
321   std::shuffle(input.begin(), input.end(), random_number_generator);
322   assert(f(input.begin(), input.end()));
323 }
324
325 template<typename checker>
326 void random_points_in_a_square(checker f) {
327   std::cout << "random points in a square\n";
328   std::size_t const bign = 10000;
329   for (std::size_t n = 0; n <= bign; ++n) {
330     if (n % 100 == 0) {
331       std::cout << "." << std::flush;
332     }
333     using P = point<int>;
334     using S = std::vector<P>;
335     S input(n);
336     generate(input.begin(), input.end());
```

```
337     assert(f(input.begin(), input.end()));
338   }
339   std::cout << "\n" << std::flush;
340 }
341
342 template<typename checker>
343 void positive_overflow(checker f) {
344   std::cout << "positive overflow\n";
345   using T = int;
346   using P = point<T>;
347   T max = std::numeric_limits<T>::max();
348   P origo(100, 100);
349   P left_top(100, max);
350   P right_top(max, max);
351   P middle(100000, 100000);
352   P right_bottom(max, 100);
353   using S = std::vector<P>;
354   S input;
355   input.push_back(left_top);
356   input.push_back(right_bottom);
357   input.push_back(origo);
358   input.push_back(middle);
359   input.push_back(right_top);
360   assert(f(input.begin(), input.end()));
361 }
362
363 template<typename checker>
364 void negative_overflow(checker f) {
365   std::cout << "negative overflow\n";
366   using T = int;
367   using P = point<T>;
368   T min = std::numeric_limits<T>::min();
369   P origo;
370   P left_top(min, 0);
371   P left_bottom(min, min);
372   P right_bottom(0, min);
373   P middle(-10000, -10000);
374   using S = std::vector<P>;
375   S input;
376   input.push_back(left_top);
377   input.push_back(right_bottom);
378   input.push_back(origo);
379   input.push_back(middle);
380   input.push_back(left_bottom);
381   assert(f(input.begin(), input.end()));
382 }
383
384 template<typename checker>
385 void corners_of_the_universe(checker f) {
386   std::cout << "corners of the universe\n";
387   using T = int;
388   using P = point<T>;
```

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

```
441   input.push_back(p3);
442   input.push_back(p4);
443   input.push_back(p5);
444   input.push_back(p6);
445   input.push_back(p7);
446   input.push_back(p8);
447   input.push_back(p9);
448   input.push_back(p10);
449   assert(f(input.begin(), input.end()));
450 }
451
452 int main() {
453   using P = point<int>;
454   using S = std::vector<P>;
455   using I = typename S::iterator;
456   using R = checker<I>;
457   using T = testcase<R>;
458
459   T suite[] = {
460     empty_set,
461     one_point,
462     two_points,
463     three_points_on_a_vertical_line,
464     three_points_on_a_horizontal_line,
465     ten_equal_points,
466     four_poles_and_many_duplicates_inside,
467     four_poles_with_duplicates,
468     quadrilateral_with_duplicates,
469     duplicates_on_the_periphery,
470     many_east_pole_candidates,
471     line_quadrilateral_line,
472     many_big_numbers,
473     noisy_box,
474     random_points_on_a_parabola,
475     random_points_in_a_square,
476     positive_overflow,
477     negative_overflow,
478     corners_of_the_universe,
479     negative_coordinates,
480     degenerate_coordinates
481   };
482
483   R program = NAME::check;
484   for (T test: suite) {
485     try {
486       test(program);
487     }
488     catch(std::exception& e) {
489       std::cerr << e.what() << std::endl;
490     }
491   }
492   return 0;
```

```
493 }
```

F.3 driver.c++

```
1  /*
2    Performance Engineering Laboratory © 2017–2018
3  */
4
5  unsigned long maxsize = 128 * 1024 * 1024; // 512 * ...works slowly
6  // 1 << 30 = 1073741824 there is space, but things become very slow
7
8  #if defined(MEASURE_TURNS)
9
10 long long turns = 0;
11
12 #endif
13
14 #if defined(MEASURE_COMPS) or defined(MEASURE_MOVES)
15
16 long long comps = 0;
17 long long moves = 0;
18
19 #endif
20
21 #include <cassert> // assert macro
22 #include <cstdlib> // std::atoi std::size_t
23 #include <cmath> // std::sqrt
24 #include <ctime> // std::clock_t std::clock CLOCKS_PER_SEC
25 #include <iostream> // std::cout std::cerr
26 #include <iterator> // std::iterator_traits std::distance
27 #include <limits> // std::numeric_limits
28 #include <random> // std::linear_congruential_engine
29
30 #include "algorithm.h++" // NAME::solve
31 #include "point.h++" // point
32
33 using linear_congruential_engine = std::linear_congruential_engine<
34   unsigned long long, 6364136223846793005U, 1442695040888963407U,
      ↪ 0U>;
35 constexpr unsigned long long seed = 10164167376618180197U;
36 linear_congruential_engine random_number_generator{seed};
37
38 #if defined(DISC)
39
40 template<typename I>
41 void generate(I p, I r) {
42   using P = typename std::iterator_traits<I>::value_type;
43   using T = typename P::coordinate;
44   T t = std::numeric_limits<T>::max();
45   std::uniform_int_distribution<T> distribution(-t, t);
46   for (I q = p; q != r; ++q) {
```

```
47      T x = 0;
48      T y = 0;
49      using R = double;
50      R radius = R(t);
51      R root = 0.0;
52      do {
53        x = distribution(random_number_generator);
54        y = distribution(random_number_generator);
55        R dx = R(x) * R(x);
56        R dy = R(y) * R(y);
57        root = std::sqrt(dx + dy);
58      } while (root > radius);
59      *q = P(x, y);
60    }
61  }
62
63  #elif defined(UNIVERSE)
64
65  template<typename I>
66  void generate(I p, I r) {
67    std::size_t n = r - p;
68    using P = typename std::iterator_traits<I>::value_type;
69    using T = typename P::coordinate;
70    T t = std::sqrt(n);
71    std::uniform_int_distribution<T> distribution(-t, +t);
72    for (I q = p; q != r; ++q) {
73      T x = distribution(random_number_generator);
74      T y = distribution(random_number_generator);
75      *q = P(x, y);
76    }
77  }
78
79  #elif defined(SPECIAL)
80
81  template<typename I>
82  void generate(I p, I r) {
83    assert(std::distance(p, r) >= 4);
84    using P = typename std::iterator_traits<I>::value_type;
85    P origin(0, 0);
86    P west(-1, 0);
87    P north(0, 1);
88    P east(1, 0);
89    P south(0, -1);
90    *p = west;
91    ++p;
92    *p = north;
93    ++p;
94    *p = east;
95    ++p;
96    *p = south;
97    ++p;
98    for (I q = p; q != r; ++q) {
```

```
99      *q = origin;
100   }
101 }
102
103 #elif defined(LINE)
104
105 template<typename I>
106 void generate(I p, I r) {
107   using P = typename std::iterator_traits<I>::value_type;
108   using T = typename P::coordinate;
109   T t = std::numeric_limits<T>::max();
110   std::uniform_int_distribution<T> distribution(-t, t);
111   for (I q = p; q != r; ++q) {
112     T x = distribution(random_number_generator);
113     *q = P(x, x);
114   }
115 }
116
117 #elif defined(PARABOLA)
118
119 template<typename I>
120 void generate(I p, I r) {
121   using P = typename std::iterator_traits<I>::value_type;
122   using T = typename P::coordinate;
123   T j = 0;
124   T modulus = 1 << 15;
125   for (I q = p; q != r; ++q) {
126     *q = P(j, j * j);
127     j = (j + 1) % modulus;
128   }
129   std::shuffle(p, r, random_number_generator);
130 }
131
132 #elif defined(SORTED)
133
134 template<typename I>
135 void generate(I p, I r) {
136   using P = typename std::iterator_traits<I>::value_type;
137   using T = int;
138   T min = std::numeric_limits<T>::min();
139   T max = std::numeric_limits<T>::max();
140   std::uniform_int_distribution<T> distribution(min, max);
141   for (I q = p; q != r; ++q) {
142     T x = distribution(random_number_generator);
143     T y = distribution(random_number_generator);
144     *q = P(x, y);
145   }
146   std::sort(p, r,
147     [](P const& p, P const& q) -> bool {
148       return p.x < q.x;
149     });
150 }
```

```
151
152 #elif defined(BELL)
153
154 template<typename I>
155 void generate(I p, I r) {
156   using P = typename std::iterator_traits<I>::value_type;
157   using T = typename P::coordinate;
158   using D = typename std::iterator_traits<I>::difference_type;
159   D n = std::distance(p, r);
160   using R = double;
161   R radius = R(std::numeric_limits<T>::max());
162   R mean = 0.0;
163   R stddev = radius / (2 + std::log(n));
164   std::normal_distribution<R> distribution(mean, stddev);
165   for (I q = p; q != r; ++q) {
166     T x = std::round(distribution(random_number_generator));
167     T y = std::round(distribution(random_number_generator));
168     *q = P(x, y);
169   }
170 }
171
172 #else // default SQUARE
173
174 template<typename I>
175 void generate(I p, I r) {
176   using P = typename std::iterator_traits<I>::value_type;
177   using T = int; // typename P::coordinate;
178   T min = std::numeric_limits<T>::min();
179   T max = std::numeric_limits<T>::max();
180   std::uniform_int_distribution<T> distribution(min, max);
181   for (I q = p; q != r; ++q) {
182     T x = distribution(random_number_generator);
183     T y = distribution(random_number_generator);
184     *q = P(x, y);
185   }
186 }
187
188 #endif
189
190 void usage(char const* program) {
191   std::cerr << "Usage: " << program << " <n>\n";
192   exit(1);
193 }
194
195 #if defined(MEASURE_COMPS) or defined(MEASURE_MOVES)
196
197 template<typename T>
198 class counter {
199 private:
200
201   T datum;
202
```

```
203 public:
204
205   static constexpr bool is_signed = true;
206   static constexpr bool is_integer = true;
207   static constexpr bool is_exact = true;
208   static constexpr bool is_bounded = true;
209   static constexpr bool is_modulo = not is_signed;
210   static constexpr std::size_t radix = 2;
211   static constexpr std::size_t length = 1;
212   static constexpr std::size_t width = 8 * sizeof(T);
213   static constexpr T min = std::numeric_limits<T>::min();
214   static constexpr T max = std::numeric_limits<T>::max();
215
216   explicit counter()
217     : datum(0) {
218     moves += 1;
219   }
220
221   counter(int x)
222     : datum(x) {
223     moves += 1;
224   }
225
226   counter(counter const& other) :
227     datum(other.datum) {
228     moves += 1;
229   }
230
231   template<typename number>
232   explicit counter(number x = 0)
233     : datum(x) {
234     moves += 1;
235   }
236
237   counter(counter&& other) {
238     datum = std::move(other.datum);
239     moves += 1;
240   }
241
242   counter& operator=(counter const& other) {
243     datum = other.datum;
244     moves += 1;
245     return *this;
246   }
247
248   counter& operator=(counter&& other) {
249     datum = std::move(other.datum);
250     moves += 1;
251     return *this;
252   }
253
254   operator T() const {
```

```
255       return datum;
256     }
257
258     template<typename U>
259     friend bool operator==(counter<U> const&, counter<U>
          ↪ const&);
260
261     template<typename U>
262     friend bool operator!=(counter<U> const&, counter<U>
          ↪ const&);
263
264     template<typename U>
265     friend bool operator<(counter<U> const&, counter<U> const&);
266
267     template<typename U>
268     friend bool operator>(counter<U> const&, counter<U> const&);
269
270     template<typename U>
271     friend bool operator<=(counter<U> const&, counter<U> const&);
272
273     template<typename U>
274     friend bool operator>=(counter<U> const&, counter<U> const&);
275   };
276
277   template<typename T>
278   bool operator==(counter<T> const& x, counter<T> const& y) {
279     ++comps;
280     return x.datum == y.datum;
281   }
282
283   template<typename T>
284   bool operator!=(counter<T> const& x, counter<T> const& y) {
285     ++comps;
286     return x.datum != y.datum;
287   }
288
289   template<typename T>
290   bool operator<(counter<T> const& x, counter<T> const& y) {
291     ++comps;
292     return x.datum < y.datum;
293   }
294
295   template<typename T>
296   bool operator>(counter<T> const& x, counter<T> const& y) {
297     ++comps;
298     return x.datum > y.datum;
299   }
300
301   template<typename T>
302   bool operator<=(counter<T> const& x, counter<T> const& y) {
303     ++comps;
304     return x.datum <= y.datum;
```

```
305 }
306
307 template<typename T>
308 bool operator≥(counter<T> const& x, counter<T> const& y) {
309   ++comps;
310   return x.datum ≥ y.datum;
311 }
312
313 #endif
314
315 int main(int argc, char** argv) {
316   unsigned long n = 15;
317   if (argc == 2) {
318     n = std::atoi(argv[1]);
319   }
320   else {
321     usage(argv[0]);
322   }
323
324   unsigned long repetitions = maxsize / n;
325   if (n > maxsize) {
326     repetitions = 1;
327     maxsize = n;
328   }
329
330 #if defined(MEASURE_MOVES) or defined(MEASURE_COMPS)
331
332   using P = point<counter<int>>;
333
334 #else
335
336   using P = point<int>;
337
338 #endif
339
340   using I = P*;
341   I a = new P[maxsize];
342   I b = a;
343   for (volatile unsigned long t = 0; t ≠ repetitions; ++t) {
344     generate(b, b + n);
345     b = b + n;
346   }
347
348   b = a;
349
350 #if defined(MEASURE_TURNS)
351
352   turns = 0;
353
354 #elif defined(MEASURE_MOVES)
355
356   moves = 0;
```

```
357
358 #elif defined(MEASURE_COMPS)
359
360   comps = 0;
361
362 #elif defined(PRUNING)
363
364   pruning_efficiency = 0;
365
366 #else
367
368   std::clock_t start = std::clock();
369
370 #endif
371
372   for (volatile unsigned long t = 0; t != repetitions; ++t) {
373     (void) NAME::solve(&b[0], &b[n]);
374     b = b + n;
375   }
376
377 #if defined(MEASURE_TURNS)
378
379   double t = double(repetitions) * double(n);
380   std::cout.precision(3);
381   std::cout << n << "\t" << double(turns) / t << "\n";
382
383 #elif defined(MEASURE_MOVES)
384
385   double t = double(repetitions) * double(n);
386   std::cout.precision(3);
387   std::cout << n << "\t" << double(moves) / t << "\n";
388
389 #elif defined(MEASURE_COMPS)
390
391   double t = double(repetitions) * double(n);
392   std::cout.precision(3);
393   std::cout << n << "\t" << double(comps) / t << "\n";
394
395 #elif defined(PRUNING)
396
397   double t = double(repetitions) * double(n);
398   std::cout.precision(3);
399   std::cout << n << "\tpruning: " << double(pruning_efficiency) *
      ↪ 100.0 / t << "\n";
400
401 #else
402
403   std::clock_t stop = std::clock();
404
405   double t = double(repetitions) * double(n);
406   double ns = 1000000000.0 * double(stop - start) /
      ↪ double(CLOCKS_PER_SEC);
```

```
407     std::cout.precision(4);
408     std::cout << n << "\t" << ns / t << "\n";
409
410 #endif
411
412     delete[] a;
413     return 0;
414 }
```

G. Makefile

G.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))
elimsquare-tests:= $(addsuffix .elimsquare, $(implementations))
elimdisc-tests:= $(addsuffix .elimdisc, $(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++
```

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

```
126         @make -i --no-print-directory $*.comp 2>&1 | tee -a $*.log
127         @make -i --no-print-directory $*.move 2>&1 | tee -a $*.log
128         @make -i --no-print-directory $*.square 2>&1 | tee -a $*.log
129         @make -i --no-print-directory $*.disc 2>&1 | tee -a $*.log
130         @make -i --no-print-directory $*.bell 2>&1 | tee -a $*.log
131         @make -i --no-print-directory $*.elimsquare 2>&1 | tee -a $*.log
132         @make -i --no-print-directory $*.elimdisc 2>&1 | tee -a $*.log
133         @date 2>&1 | tee -a $*.log
134
135 all:
136         @make -i --no-print-directory sort.benchmark
137         @make -i --no-print-directory plane_sweep.benchmark
138         @make -i --no-print-directory torch.benchmark
139         @make -i --no-print-directory quickhull.benchmark
140         @make -i --no-print-directory throw_away.benchmark
141
142 TESTFLAGS=-O3 -std=c++17 -Wall -Wextra -x c++ -fconcepts -g -DDEBUG
143
144 $(sanitychecks): %.check : %.h++
145         @cp $*.h++ algorithm.h++
146         $(CXX) $(TESTFLAGS) $(IFLAGS) -DNAME=$* check-driver.c++
147         ./a.out
148         rm -f ./a.out
149
150 $(unittests): %.test : %.h++
151         @cp $*.h++ algorithm.h++
152         $(CXX) $(TESTFLAGS) $(IFLAGS) -DNAME=$* test-driver.c++
153         ./a.out
154         rm -f algorithm.h++ ./a.out
155
156 # Other tools
157
158 clean:
159         - rm  -f a.out temp algorithm.h++ 2>/dev/null
160
161 veryclean: clean
162         - rm -f *~ .*~ */*~ 2>/dev/null
163
164 find:
165         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] J. Katajainen, Pure compile-time functions and classes in the CPH MPL, CPH STL report **2017-2**, Dept. Comput. Sci., Univ. Copenhagen (2017).

[4] J. Katajainen, *Width-specific templates* $\mathbb{N}$<b> *and* $\mathbb{Z}$<b> *for integer types in C++: Going beyond C*, Work in progress (2018).