

Convex-hull algorithms in C++

Ask Neve Gamby¹ 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 smallest convex polygon enclosing all the points of S . In the output the vertices must be reported in the order in which they appear along the boundary of the polygon—either in clockwise or counterclockwise order. Further, when the coordinates of the points are integers, the computed output should always be correct.

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

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

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

Contents

Abstract	1
1 Introduction	4
2 Simple use case	5
3 Use of the <code>makefile</code>	6
References	7
A Source code	8
A.1 Copyright notice	8
A.2 Release date	8
B Convex-hull algorithms	9
B.1 <code>plane_sweep.h++</code>	9
B.2 <code>torch.h++</code>	13
B.3 <code>divide_and_conquer.h++</code>	18
B.4 <code>quickhull.h++</code>	23
B.5 <code>poles_first.h++</code>	27
B.6 <code>throw_away.h++</code>	30
B.7 <code>introhull.h++</code>	36
B.8 <code>bucketing.h++</code>	41
C Micro-benchmarks	46
C.1 <code>multi_precision.h++</code>	46
C.2 <code>double_precision.h++</code>	46
C.3 <code>floating_point_filter.h++</code>	47
C.4 <code>minmax_element.h++</code>	47
C.5 <code>partition.h++</code>	48
C.6 <code>copy.h++</code>	48
C.7 <code>sort.h++</code>	49
C.8 <code>lexicographic_sort.h++</code>	49
C.9 <code>angular_sort.h++</code>	50
C.10 <code>bsort.h++</code>	50
C.11 <code>tsort.h++</code>	51
C.12 <code>shuffle.h++</code>	51
D Helpers	53
D.1 <code>point.h++</code>	53
D.2 <code>validation.h++</code>	61
D.3 <code>bucketsort.h++</code>	67
D.4 <code>two_phase_bucketsort.h++</code>	70
E Drivers	73
E.1 <code>check-driver.c++</code>	73
E.2 <code>test-driver.c++</code>	73
E.3 <code>driver.c++</code>	83

F Makefile	92
F.1 makefile	92

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. [4, Section 33.3]. This book makes good job in explaining Graham's ROTATIONAL-SWEEP algorithm [11], but this algorithm is not very efficient in practice. The convex-hull algorithms (PLANE-SWEEP [2], TORCH [10], DIVIDE & CONQUER [14], QUICKHULL [6], POLES-FIRST [1], THROW-AWAY [5], INTROHULL [8], and BUCKETING [9]), the implementations of which are included in this collection, are known to be faster. We have tried to implement these algorithms in the best possible way, but we have to admit that we have made many embarrassing errors on the journey. Some of the algorithms 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 collection, we can just congratulate you. In this case, it is quite probable that you have something that could be published.

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

Let us assume that the input points are given in a sequence, i.e. in a `std::vector`. An *in-place* convex-hull algorithm (see, for example, [3]) partitions this sequence into two parts: (1) the first part contains all the extreme points in clockwise or counterclockwise order of their appearance on P and (2) the second part contains all the remaining points that are inside P or on the periphery of P .

All the algorithms in this collection rearrange the points in the input sequence in place. In addition to this side effect, the algorithms return an iterator to the first interior point. Thus, for two iterators `i` and `k`, if the range `[[i..k)` specifies the input and `j` is the returned output, the range `[[i..j)` contains the extreme points in circular order and the range `[[j..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 [13] and the CPH MPL [12]. Since both of these library components rely on concepts, you should compile the code with the option `-fconcepts`. We had these tools in the parent directory, so we also had to use the compiler option `-I...`

3. Use of the makefile

You can use `make` to redo the experiments done in the papers [7, 8, 9]. 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. In this case the driver `driver.c++` will report the size of the input (n) and the running time in nanoseconds divided by n .

```

shell> make plane_sweep.square
g++ -O3 -std=c++17 -x c++ -Wall -Wextra -fconcepts -DNDEBUG -I.. -
    ↪ DNAME=plane_sweep driver.c++
1024    55.1
32768   52.58
1048576 57.52
33554432    80.37

```

The same `makefile` can also handle the micro-benchmarks since these are packaged in the same way as the convex-hull algorithms. Each micro-benchmark is defined its own namespace—the name of this namespace is the same as the name of the file without the trailing suffix `h++`—and the driver will run the function `solve` in this namespace. To give an example, let us see how fast `INTROSORT` can sort a sorted sequence of points according to their x -coordinates:

```

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

```

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

```

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

```

References

- [1] S. G. Akl and G. T. Toussaint, A fast convex hull algorithm, *Inform. Process. Lett.* **7**, 5 (1978), 219–222.
- [2] A. M. Andrew, Another efficient algorithm for convex hulls in two dimensions, *Inform. Process. Lett.* **9**, 5 (1979), 216–219.
- [3] 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.
- [4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd edition, The MIT Press (2009).
- [5] L. Devroye and G. T. Toussaint, A note on linear expected time algorithms for finding convex hulls, *Computing* **26**, 4 (1981), 361–366.
- [6] W. F. Eddy, A new convex hull algorithm for planar sets, *ACM Trans. Math. Software* **3**, 4 (1977), 398–403.

- [7] A. N. Gamby and J. Katajainen, A note on the implementation quality of a convex-hull algorithm, CPH STL report **2017-3**, Dept. Comput. Sci., Univ. Copenhagen (2017).
- [8] A. N. Gamby and J. Katajainen, *Convex-hull algorithms: Implementation, testing, and experimentation*, Submitted (2018).
- [9] Ask Neve Gamby and Jyrki Katajainen, *A faster convex-hull algorithm via bucketing*, Work in progress (2018).
- [10] A. J. P. Gomes, A total order heuristic-based convex hull algorithm for points in the plane, *Comput.-Aided Design* **70** (2016), 153–160.
- [11] R. L. Graham, An efficient algorithm for determining the convex hull of a finite planar set, *Inform. Process. Lett.* **1**, 4 (1972), 132–133.
- [12] J. Katajainen, Pure compile-time functions and classes in the CPH MPL, CPH STL report **2017-2**, Dept. Comput. Sci., Univ. Copenhagen (2017).
- [13] J. Katajainen, *Width-specific templates \mathbb{N} and \mathbb{Z} for integer types in C++: Going beyond C*, Work in progress (2018).
- [14] F. P. Preparata and S. J. Hong, Convex hulls of finite sets of points in two and three dimensions, *Commun. ACM* **20**, 2 (1977), 87–93.

A. Source code

A.1 Copyright notice

Copyright © 2000–2018 by The authors and Performance Engineering Laboratory (University of Copenhagen)

The programs included in the CPH STL may be freely copied and distributed, provided that no changes whatsoever are made. Changes are permissible only if the modified files are given new names, different from the names of existing files in the CPH STL, and only if the modified files are clearly identified as not being part of the library. The programs may also be used in part, as long as they are attributed to the original source. Usage of the source code in derived works is otherwise unrestricted.

The authors have tried to produce correct and useful programs, but no warranty of any kind should be assumed.

A.2 Release date

2018-10-26

B. Convex-hull algorithms

B.1 *plane_sweep.h++*

```

1  /*
2   Performance Engineering Laboratory © 2017–2018
3  */
4
5  #ifndef __PLANE_SWEEP__
6  #define __PLANE_SWEEP__
7
8  #include <algorithm> // std::sort std::partition ...
9  #include <cassert> // assert macro
10 #include <cstdlib> // std::size_t
11 #include <functional> // std::less std::greater
12 #include <iterator> // std::iterator_traits
13 #include "point.h++" // left_turn right_turn
14 #include "two_phase_bucketsort.h++" // two_phase_bucketsort::sort
15 #include <utility> // std::pair std::get ...
16 #include "validation.h++" // same_multiset convex_polygon all_inside
17
18 // #define TWOPHASEBUCKETSORT
19
20 namespace plane_sweep {
21
22     template<typename variable>
23     void ignore_warning(variable) {
24     }
25
26     // move *st <- *rd and *nd <- *th by swaps in parallel
27
28     template<typename I>
29     void parallel_iter_swap(I st, I nd, I rd, I th) {
30         assert(st ≠ nd and rd ≠ th);
31         std::iter_swap(st, rd);
32         if (th == st) {
33             std::iter_swap(nd, rd);
34         }
35         else {
36             std::iter_swap(nd, th);
37         }
38     }
39
40     template<typename I>
41     std::pair<I, I> find_poles(I first, I past) {
42         using P = typename std::iterator_traits<I>::value_type;
43         auto pair = std::minmax_element(first, past,
44             [](P const& a, P const& b) → bool {
45                 return (a.x < b.x) or (a.x == b.x and a.y < b.y);
46             });
47         return pair;
48     }

```

```

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

```

```

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

```

```

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

```

```

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

```

B.2 torch.h++

```

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

```

```

36     }
37     using P = typename std::iterator_traits<I>::value_type;
38     using T = typename P::coordinate;
39     T max = (*first).y;
40     I i = first + 1;
41     for (I j = first + 1; j  $\neq$  past; ++j) {
42         if ((*j).y  $\geq$  max) {
43             max = (*j).y;
44             std::iter_swap(i, j);
45             ++i;
46         }
47     }
48     return i;
49 }
50
51 template<typename I>
52 I go_downstairs(I first, I past) {
53     if (first == past) {
54         return past;
55     }
56     using P = typename std::iterator_traits<I>::value_type;
57     using T = typename P::coordinate;
58     T min = (*first).y;
59     I i = first + 1;
60     for (I j = first + 1; j  $\neq$  past; ++j) {
61         if ((*j).y  $\leq$  min) {
62             min = (*j).y;
63             std::iter_swap(i, j);
64             ++i;
65         }
66     }
67     return i;
68 }
69
70 template<typename I>
71 I scan(I first, I past) {
72     assert(first  $\neq$  past);
73     I top = first;
74     I next = first + 1;
75     for (I i = next; i  $\neq$  past; ++i) {
76         while (top  $\neq$  first and not right_turn>(*top, *i))
77              $\hookrightarrow$  {
78                 --top;
79             }
80         ++top;
81         std::iter_swap(i, top);
82     }
83     return ++top;
84 }
85
86 template<typename I>
87 I scan_one_more(I first, I top, I extra) {

```

```

87     while (top  $\neq$  first and not right_turn(*(top - 1), *top,
88          $\hookrightarrow$  *extra)) {
89         --top;
90     }
91     return ++top;
92 }
93
94 template<typename I>
95 I solve(I first, I past) {
96     using P = typename std::iterator_traits<I>::value_type;
97     using T = typename P::coordinate;
98     std::size_t n = past - first;
99     if (n < 2) {
100         return past;
101     }
102
103     // find the west and east poles
104
105     std::pair<I, I> pair = std::minmax_element(first, past,
106         [](P const& a, P const& b)  $\rightarrow$  bool {
107             return (a.x < b.x) or (a.x == b.x and a.y < b.y);
108         });
109     I west = std::get<0>(pair);
110     I east = std::get<1>(pair);
111
112     // all points equal?
113
114     if (*west == *east) {
115         return first + 1;
116     }
117
118     // all points on a vertical line?
119
120     if ((*west).x == (*east).x) {
121         parallel_iter_swap(first, first + 1, west, east);
122         return first + 2;
123     }
124
125     // move west to its final place and east temporarily to the end
126
127     I last = past - 1;
128     parallel_iter_swap(first, last, west, east);
129     west = first;
130     east = last;
131
132     // find the south and north poles
133
134     pair = std::minmax_element(first, past,
135         [](P const& a, P const& b)  $\rightarrow$  bool {
136             return (a.y < b.y) or (a.y == b.y and a.x < b.x);
137         });
138     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

```

B.3 *divide_and_conquer.h++*

```

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

```

```

29     }
30 }
31
32 template<typename I>
33 std::pair<I, I> find_poles(I first, I past) {
34     using P = typename std::iterator_traits<I>::value_type;
35     auto pair = std::minmax_element(first, past,
36         [](P const& a, P const& b) → bool {
37             return (a.x < b.x) or (a.x == b.x and a.y < b.y);
38         });
39     return pair;
40 }
41
42 template<typename I>
43 I partition_upper_lower(I first, I past) {
44     assert(first ≠ past);
45     using P = typename std::iterator_traits<I>::value_type;
46     using T = typename P::coordinate;
47     std::pair<I, I> pair = find_poles(first, past);
48     I west = first;
49     I east = past - 1;
50     parallel_iter_swap(west, east, std::get<0>(pair),
51         ↪ std::get<1>(pair));
52
53     T upper_limit = std::max((*west).y, (*east).y);
54     T lower_limit = std::min((*west).y, (*east).y);
55     I i = std::partition(west + 1, east,
56         [&](P const& q) → bool {
57             if (q.y ≥ upper_limit) {
58                 return true;
59             }
60             else if (q.y ≤ lower_limit) {
61                 return false;
62             }
63             return not left_turn(*west, q, *east);
64         });
65     std::iter_swap(i, east);
66     return i + 1;
67 }
68
69 template<typename I, typename C>
70 std::pair<I, I> clean(I pole, I rest, C compare) {
71     assert(pole ≠ rest);
72     I k = pole + 1;
73     while (k ≠ rest and (*k).x == (*pole).x) {
74         ++k;
75     }
76     if (k == pole + 1) {
77         return std::make_pair(pole, pole + 1); // (top, next)
78     }
79     I bottom = std::min_element(pole, k, compare);
80     std::iter_swap(pole, bottom);

```

```

80     I top = std::max_element(pole + 1, k, compare);
81     if ((*top).y == (*pole).y) {
82         return std::make_pair(pole, k);
83     }
84     std::iter_swap(pole + 1, top);
85     return std::make_pair(pole + 1, k);
86 }
87
88 template<typename I>
89 void swap_blocks(I source, I past_the_end, I target) {
90     // retains the order in [source, past_the_end)
91     if (source == target or source == past_the_end) {
92         return;
93     }
94     using P = typename std::iterator_traits<I>::value_type;
95     I hole = target;
96     P p = *target;
97     I const last = past_the_end - 1;
98     while (true) {
99         *hole = *source;
100        ++hole;
101        if (source == last) {
102            break;
103        }
104        *source = *hole;
105        ++source;
106    }
107    *source = p;
108 }
109
110 template<typename I>
111 I scan(I first, I top, I next, I past) {
112     assert(first != past);
113     for (I i = next; i != past; ++i) {
114         while (top != first and not right_turn(*(top - 1), *top, *i))
115             ↪ {
116             --top;
117             }
118         ++top;
119         std::iter_swap(i, top);
120     }
121     return ++top;
122 }
123
124 template<typename I>
125 I scan_one_more(I first, I top, I extra) {
126     while (top != first and not right_turn(*(top - 1), *top,
127         ↪ *extra)) {
128         --top;
129     }
130     return ++top;
131 }

```

```

130
131 template<typename I, typename C>
132 I brute_force(I first, I past, C compare) {
133     std::size_t n = past - first;
134     assert(n ≤ 4);
135     if (n ≤ 1) {
136         return past;
137     }
138     else if (n == 2) {
139         if (*first == *(first + 1)) {
140             return first + 1;
141         }
142         if (not compare(*first, *(first + 1))) {
143             std::iter_swap(first, first + 1);
144         }
145         return past;
146     }
147     if (n == 3) {
148         I west = std::min_element(first, past, compare);
149         std::iter_swap(first, west);
150         I east = std::max_element(first + 1, past, compare);
151         if (*first == *east) {
152             return first + 1;
153         }
154         std::iter_swap(first + 2, east);
155         if (right_turn(*first, *(first + 1), *(first + 2))) {
156             return first + 3;
157         }
158         std::iter_swap(first + 1, first + 2);
159         return first + 2;
160     }
161     // n == 4
162     I west = std::min_element(first, past, compare);
163     std::iter_swap(first, west);
164     I east = std::max_element(first + 1, past, compare);
165     if (*first == *east) {
166         return first + 1;
167     }
168     std::iter_swap(first + 3, east);
169     if (not compare(*(first + 1), *(first + 2))) {
170         std::iter_swap(first + 1, first + 2);
171     }
172     I rest = scan(first, first, first + 1, past);
173     return rest;
174 }
175
176 template<typename I, typename C>
177 I recurse(I first, I past, C compare) {
178     std::size_t n = past - first;
179     if (n ≤ 4) {
180         return brute_force(first, past, compare);
181     }

```

```

182     I middle = first + n / 2;
183     I rest1 = recurse(first, middle, compare);
184     I rest2 = recurse(middle, past, compare);
185     std::size_t h2 = rest2 - middle;
186     I rest = rest1 + h2;
187     swap_blocks(middle, rest2, rest1);
188     std::inplace_merge(first, rest1, rest, compare);
189     std::pair<I, I> pair = clean(first, rest, compare);
190     I top = std::get<0>(pair);
191     I next = std::get<1>(pair);
192     rest = scan(first, top, next, rest);
193     return rest;
194 }
195
196 template<typename I>
197 I solve(I first, I past) {
198     using P = typename std::iterator_traits<I>::value_type;
199     std::size_t n = past - first;
200     if (n < 2) {
201         return past;
202     }
203     I middle = partition_upper_lower(first, past);
204     I rest1 = recurse(first, middle,
205         [](P const& a, P const& b) {
206             return a.x < b.x or (a.x == b.x and a.y < b.y);
207         });
208     if (middle == past) {
209         return rest1;
210     }
211     I west = first;
212     std::iter_swap(rest1 - 1, middle - 1);
213     I east = middle - 1;
214     I rest2 = recurse(east, past,
215         [](P const& a, P const& b) {
216             return a.x > b.x or (a.x == b.x and a.y > b.y);
217         });
218     if (rest2 ≠ east) {
219         rest2 = scan_one_more(east, rest2 - 1, west);
220     }
221     std::size_t h2 = rest2 - east;
222     I rest = rest1 + (h2 - 1);
223     swap_blocks(east, rest2, rest1 - 1);
224     return rest;
225 }
226
227 template<typename I>
228 bool check(I first, I past) {
229     using P = typename std::iterator_traits<I>::value_type;
230     using S = std::vector<P>;
231     using J = typename S::iterator;
232     S data;
233     std::size_t n = past - first;

```

```

234     data.resize(n);
235     std::copy(first, past, data.begin());
236     J rest = solve(data.begin(), data.end());
237     bool ok = validation::same_multiset(data.begin(), data.end(),
    ↪ first, past) and validation::convex_polygon(data.begin(),
    ↪ rest) and validation::all_inside(rest, data.end(),
    ↪ data.begin(), rest);
238     return ok;
239 }
240 }
241
242 #endif

```

B.4 quickhull.h++

```

1  /*
2   Performance Engineering Laboratory © 2017–2018
3  */
4
5  #ifndef __QUICKHULL__
6  #define __QUICKHULL__
7
8  #include "point.h++" // signed_area left_turn
9
10 #include <algorithm> // std::minmax_element std::partitioning ...
11 #include <cassert> // assert macro
12 #include <cstdlib> // std::size_t
13 #include <iterator> // std::iterator_traits
14 #include <utility> // std::pair std::make_pair ...
15 #include "validation.h++" // same_multiset convex_polygon all_inside
16 #include <vector> // std::vector
17
18 namespace quickhull {
19
20     template <typename variable>
21     void ignore_warning(variable) {
22     }
23
24     // move *st <- *rd and *nd <- *th by swaps in parallel
25
26     template<typename I>
27     void parallel_iter_swap(I st, I nd, I rd, I th) {
28         assert(st ≠ nd and rd ≠ th);
29         std::iter_swap(st, rd);
30         if (th == st) {
31             std::iter_swap(nd, rd);
32         }
33         else {
34             std::iter_swap(nd, th);
35         }
36     }

```

```

37
38 template<typename I>
39 std::pair<I, I> find_poles(I first, I past) {
40     using P = typename std::iterator_traits<I>::value_type;
41     auto pair = std::minmax_element(first, past,
42         [](P const& a, P const& b) → bool {
43             return (a.x < b.x) or (a.x == b.x and a.y < b.y);
44         });
45     return pair;
46 }
47
48 template<typename I>
49 I find_furthest(I first, I past, I antipole) {
50     assert(first  $\neq$  past);
51     I pole = first;
52     I answer = pole;
53     using U = decltype(signed_area(*pole, *pole, *pole));
54     U best = 0;
55     if ((*antipole).x == (*pole).x) { // vertical
56         for (I i = first + 1; i  $\neq$  past; ++i) {
57             U  $\Phi$  = signed_area(*pole, *antipole, *i);
58             if ( $\Phi$  > best or ( $\Phi$  == best and (*i).y < (*answer).y)) {
59                 answer = i;
60                 best =  $\Phi$ ;
61             }
62         }
63     }
64     else {
65         for (I i = first + 1; i  $\neq$  past; ++i) {
66             U  $\Phi$  = signed_area(*pole, *antipole, *i);
67             if ( $\Phi$  > best or ( $\Phi$  == best and (*i).x < (*answer).x)) {
68                 answer = i;
69                 best =  $\Phi$ ;
70             }
71         }
72     }
73     return answer;
74 }
75
76 template<typename I>
77 I partition_left_right(I first, I past, I antipole) {
78     assert(first  $\neq$  past);
79     using P = typename std::iterator_traits<I>::value_type;
80     I pole = first;
81     I middle = std::partition(pole + 1, past,
82         [&](P const& q) → bool {
83             return not left_turn(*pole, q, *antipole);
84         });
85     return middle;
86 }
87
88 template <typename I>

```



```

89 void swap_blocks(I source, I past_the_end, I target) {
90     if (source == target or source == past_the_end) {
91         return;
92     }
93     using P = typename std::iterator_traits<I>::value_type;
94     I hole = target;
95     P p = *target;
96     I const last = past_the_end - 1;
97     while (true) {
98         *hole = *source;
99         ++hole;
100        if (source == last) {
101            break;
102        }
103        *source = *hole;
104        ++source;
105    }
106    *source = p;
107 }
108
109 template <typename I>
110 void move_away(I here, I rest, I past) {
111     if (here == rest or rest == past) {
112         return;
113     }
114     if (rest - here < past - rest) {
115         swap_blocks(here, rest, past - (rest - here));
116     }
117     else {
118         swap_blocks(rest, past, here);
119     }
120 }
121
122 template<typename I>
123 I recurse(I pole, I past, I antipole) {
124     std::size_t n = std::distance(pole, past);
125     if (n == 1) {
126         return past;
127     }
128     if (n == 2) {
129         if (no_turn(*(pole + 1), *pole, *antipole)) {
130             return pole + 1;
131         }
132         else {
133             return past;
134         }
135     }
136     I pivot = find_furthest(pole, past, antipole);
137     if (no_turn(*pivot, *pole, *antipole)) {
138         return pole + 1;
139     }
140     I last = past - 1;

```

```

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

```

```

192     bool ok = validation::same_multiset(data.begin(), data.end(),
    ↪ first, past) and validation::convex_polygon(data.begin(),
    ↪ rest) and validation::all_inside(rest, data.end(),
    ↪ data.begin(), rest);
193     return ok;
194 }
195 }
196
197 #endif

```

B.5 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  $\neq$  rest);
96     I i = first;
97     if (rest - polygon == 1) {
98         for (I j = first; j  $\neq$  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  $\neq$  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  $\neq$  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  $\neq$  end - 1; ++k) {
126         if (*lower < *k) {
127             lower = k;
128         }
129     }
130     for (I j = first; j  $\neq$  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(),
173     ↪ first, past) and validation::convex_polygon(data.begin(),
174     ↪ rest) and validation::all_inside(rest, data.end(),
175     ↪ data.begin(), rest);
176     return ok;
177 }
178 }
179 }
180 #endif

```

B.6 *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  $\neq$  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  $\neq$  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     ++moves; ++moves; ++moves;
69
70 #endif
71     Z dot = Z(CONVERSION x) + Z(CONVERSION y);
72 #if defined(MEASURE_COMPS)
73     ++comps;
74
75 #endif
76     if (dot > ne) {
77 #if defined(MEASURE_MOVES)
78     ++moves;
79
80 #endif
81     ne = dot;
82     max_position[4] = i;
83     }
84 #if defined(MEASURE_MOVES)
85     ++moves; ++moves; ++moves;
86
87 #endif
88     dot = Z(CONVERSION x) - Z(CONVERSION y);
89 #if defined(MEASURE_COMPS)
90     ++comps;
91
92 #endif
93     if (dot > se) {
94 #if defined(MEASURE_MOVES)
95     ++moves;
96
97 #endif
98     se = dot;
99     max_position[5] = i;
100     }
101 #if defined(MEASURE_MOVES)
102     ++moves; ++moves; ++moves;
103
104 #endif
105     dot = -(Z(CONVERSION x) + Z(CONVERSION y));
106 #if defined(MEASURE_COMPS)
107     ++comps;
108
109 #endif
```



```

116
117 #endif
118     if (dot > sw) {
119 #if defined(MEASURE_MOVES)
120
121         ++moves;
122
123 #endif
124         sw = dot;
125         max_position[6] = i;
126     }
127 #if defined(MEASURE_MOVES)
128
129     ++moves; ++moves; ++moves;
130
131 #endif
132     dot = Z(CONVERSION y) - Z(CONVERSION x);
133 #if defined(MEASURE_COMPS)
134
135     ++comps;
136
137 #endif
138     if (dot > nw) {
139 #if defined(MEASURE_MOVES)
140
141         ++moves;
142
143 #endif
144         nw = dot;
145         max_position[7] = i;
146     }
147 }
148 return max_position;
149 }
150
151 template<typename S>
152 void remove_duplicates(S& sequence) {
153     assert(sequence.size() ≠ 0);
154     assert(std::is_sorted(sequence.begin(), sequence.end()));
155     std::size_t i = 0;
156     std::size_t j = 1;
157     auto v = sequence[0];
158     while (j ≠ sequence.size()) {
159         if (not (sequence[j] == v)) {
160             std::swap(sequence[i], v);
161             ++i;
162             v = sequence[j];
163         }
164         ++j;
165     }
166     std::swap(sequence[i], v);
167     ++i;

```

```

168     std::size_t leftovers = sequence.size() - i;
169     while (leftovers  $\neq$  0) {
170         sequence.pop_back();
171         --leftovers;
172     }
173 }
174
175 template<typename P, typename I>
176 bool inside_convex_polygon(P const& p, I upper, I lower, I past) {
177     assert(past - upper > 2);
178     assert(*upper == *(past - 1));
179     assert(*upper < *(upper + 1));
180     assert(*upper < *(past - 2));
181     I i = upper;
182     do {
183         ++i;
184     } while (i  $\neq$  lower and *i < p);
185     if (left_turn(*(i - 1), *i, p)) {
186         return false;
187     }
188     I last = past - 1;
189     I j = lower;
190     do {
191         ++j;
192     } while (j  $\neq$  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  $\neq$  rest);
202     I i = first;
203     if (rest - polygon == 1) {
204         for (I j = first; j  $\neq$  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  $\neq$  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  $\neq$  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  $\neq$  end - 1; ++k) {
232         if (*lower < *k) {
233             lower = k;
234         }
235     }
236     for (I j = first; j  $\neq$  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  $\neq$  extrema.size(); ++i) {
255         std::iter_swap(first + i, extrema[i]);
256     }
257     // Step 2
258     I rest = plane_sweep::solve(first, first + extrema.size());
259     // Step 3
260     I interior = eliminate_inner_points(rest, past, first, rest);
261     return interior;
262 }
263
264 template<typename I>
265 I solve(I first, I past) {
266     I rest = prune(first, past);
267     return plane_sweep::solve(first, rest);
268 }
269
270 template<typename I>
271 bool check(I first, I past) {

```

```

272     using P = typename std::iterator_traits<I>::value_type;
273     using S = std::vector<P>;
274     using J = typename S::iterator;
275     S data;
276     std::size_t n = past - first;
277     data.resize(n);
278     std::copy(first, past, data.begin());
279     J rest = solve(data.begin(), data.end());
280     bool ok = validation::same_multiset(data.begin(), data.end(),
    ↪ first, past) and validation::convex_polygon(data.begin(),
    ↪ rest) and validation::all_inside(rest, data.end(),
    ↪ data.begin(), rest);
281     return ok;
282 }
283 }
284
285 #endif

```

B.7 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
148     ↪ compare) {
149     std::size_t n = std::distance(pole, past);
150     if (n == 1) {
151         return past;
152     }
153     if (n == 2) {
154         if (no_turn(*(pole + 1), *pole, *antipole)) {
155             return pole + 1;
156         }
157         else {
158             return past;
159         }
160     }
161     if (depth == 0) {
162         return plane_sweep::chain(pole, past, antipole, compare);
163     }
164     I pivot = find_furthest(pole, past, antipole);
165     if (no_turn(*pivot, *pole, *antipole)) {
166         return pole + 1;
167     }
168     I last = past - 1;
169     std::iter_swap(pivot, last); // pivot at the end
170     I mid = partition_left_right(pole, last, last);
171     I eliminated = recurse(pole, mid, last, depth - 1, compare);
172     std::iter_swap(mid, last);
173     std::iter_swap(eliminated, mid); // pivot at its final place
174     pivot = eliminated;
175     std::size_t m = past - mid;
176     ++mid;
177     ++eliminated;
178     move_away(eliminated, mid, past);
179     I interior = partition_left_right(pivot, pivot + m, antipole);
180     eliminated = recurse(pivot, interior, antipole, depth - 1,
181     ↪ compare);
182     return eliminated;

```

```

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),
196 ↪ std::get<1>(pair));
197     if (*west == *east) {
198         return first + 1;
199     }
200     I middle = partition_left_right(west, east, east);
201     std::size_t m = past - middle;
202     I eliminated = recurse(first, middle, east, stop,
203 ↪ std::less<T>());
204     std::iter_swap(middle, east);
205     std::iter_swap(eliminated, middle); // east at its final place
206     east = eliminated;
207     ++middle;
208     ++eliminated;
209     move_away(eliminated, middle, past);
210     I rest = recurse(east, east + m, west, stop,
211 ↪ std::greater<T>());
212     return rest;
213 }
214
215 template<typename I>
216 bool check(I first, I past) {
217     using P = typename std::iterator_traits<I>::value_type;
218     using S = std::vector<P>;
219     using J = typename S::iterator;
220     S data;
221     std::size_t n = past - first;
222     data.resize(n);
223     std::copy(first, past, data.begin());
224     J rest = solve(data.begin(), data.end());
225     bool ok = validation::same_multiset(data.begin(), data.end(),
226 ↪ first, past) and validation::convex_polygon(data.begin(),
227 ↪ rest) and validation::all_inside(rest, data.end(),
228 ↪ data.begin(), rest);
229     return ok;
230 }
231 }
232 #endif

```


B.8 bucketing.h++

```

1  /*
2   Performance Engineering Laboratory © 2017–2018
3  */
4
5  #ifndef __BUCKETING__
6  #define __BUCKETING__
7
8  #include <cmath> // std::sqrt std::ceil
9  #include <cstdlib> // std::size_t
10 #include <iterator> // std::iterator_traits
11 #include <limits> // std::numeric_limits
12 #include "plane_sweep.h++" // plane_sweep::solve
13 #include "point.h++"
14 #include <utility> // std::pair
15 #include <vector> // std::vector
16
17 namespace bucketing {
18
19     template<typename V>
20     class identity {
21     public:
22
23         V operator()(V const& e) const {
24             return e;
25         }
26
27         V& operator()(V& e) const {
28             return e;
29         }
30     };
31
32     template<typename D, typename V, typename K = identity<V>>
33     class formula {
34     public:
35
36         formula(D m, V const& min, V const& max, K key = K())
37             : a(double(m - 1) / (double(key(max)) - double(key(min)))),
38               b(double(m - 1) * double(key(min)) / (double(key(max)) -
39 ↪ double(key(min)))),
39               key(key) {
40         }
41
42         D operator()(V const& x) {
43             return D(a * double(key(x)) - b);
44         }
45
46     private:
47
48         double a;

```

```

49     double b;
50     K key;
51 };
52
53 // move *st <- *rd and *nd <- *th by swaps in parallel
54
55 template<typename I>
56 void parallel_iter_swap(I st, I nd, I rd, I th) {
57     assert(st ≠ nd and rd ≠ th);
58     std::iter_swap(st, rd);
59     if (th == st) {
60         std::iter_swap(nd, rd);
61     }
62     else {
63         std::iter_swap(nd, th);
64     }
65 }
66
67 // all points on the same vertical line
68
69 template<typename I>
70 I vertical_case(I first, I past) {
71     assert(std::distance(first, past) ≥ 2);
72     using P = typename std::iterator_traits<I>::value_type;
73     std::pair<I, I> pair = std::minmax_element(first, past,
74         [](P const& p, P const& q) → bool {
75             return (p.y < q.y);
76         });
77     I south = std::get<0>(pair);
78     I north = std::get<1>(pair);
79     if ((*south).y == (*north).y) {
80         std::iter_swap(first, south);
81         return first + 1;
82     }
83     parallel_iter_swap(first, first + 1, south, north);
84     return first + 2;
85 }
86
87 template<typename I>
88 I prune(I first, I past) {
89     using P = typename std::iterator_traits<I>::value_type;
90     using T = typename P::coordinate;
91     using D = typename std::iterator_traits<I>::difference_type;
92     D n = std::distance(first, past);
93     if (n < 2) {
94         return past;
95     }
96
97     // find the extreme values in the ±x directions
98
99     std::pair<I, I> pair = std::minmax_element(first, past,
100         [](P const& p, P const& q) → bool {

```

```

101     return p.x < q.x;
102   });
103   I west = std::get<0>(pair);
104   I east = std::get<1>(pair);
105   if ((*west).x == (*east).x) {
106     return vertical_case(first, past);
107   }
108   D m = std::ceil(std::sqrt(n)); // divided by 2 made the program
    ↪ faster
109   formula slab(m, (*west).x, (*east).x);
110
111   T* slab_min = new T[m];
112   T* slab_max = new T[m];
113   for (D j = 0; j ≠ m; ++j) {
114     slab_max[j] = std::numeric_limits<T>::min();
115     slab_min[j] = std::numeric_limits<T>::max();
116   }
117
118   // calculate the slab extrema
119
120   for (I j = first; j ≠ past; ++j) {
121     D i = slab((*j).x);
122     if ((*j).y < slab_min[i]) {
123       slab_min[i] = (*j).y;
124     }
125     if ((*j).y > slab_max[i]) {
126       slab_max[i] = (*j).y;
127     }
128   }
129
130   // find the extreme values in the ±y directions
131
132   D bottom = 0;
133   D top = 0;
134   T ymin = std::numeric_limits<T>::max();
135   T ymax = std::numeric_limits<T>::min();
136   for (D i = 0; i ≠ m; ++i) {
137     if (slab_max[i] > ymax) {
138       ymax = slab_max[i];
139       top = i;
140     }
141     if (slab_min[i] < ymin) {
142       ymin = slab_min[i];
143       bottom = i;
144     }
145   }
146
147   // determine the boundaries for the extreme values
148
149   T roof = (*east).y; // Q1 go backwards upstairs
150   T previous = roof;
151   for (D j = m - 1; j ≥ top; --j) {

```

```

152     previous = roof;
153     roof = std::max(roof, slab_max[j]);
154     slab_max[j] = previous;
155 }
156
157 T floor = (*east).y; // Q2 go backwards downstairs
158 previous = floor;
159 for (D j = m - 1; j ≥ bottom; --j) {
160     previous = floor;
161     floor = std::min(floor, slab_min[j]);
162     slab_min[j] = previous;
163 }
164
165 floor = (*west).y; // Q3 go downstairs
166 previous = floor;
167 for (D j = 0; j < bottom; ++j) {
168     previous = floor;
169     floor = std::min(floor, slab_min[j]);
170     slab_min[j] = previous;
171 }
172 slab_min[bottom] = std::max(previous, slab_min[bottom]);
173
174 roof = (*west).y; // Q4 go upstairs
175 previous = roof;
176 for (D j = 0; j < top; ++j) {
177     previous = roof;
178     roof = std::max(roof, slab_max[j]);
179     slab_max[j] = previous;
180 }
181 slab_max[top] = std::min(previous, slab_max[top]);
182
183 // partition the input
184
185 I o = first;
186 for (I j = first; j ≠ past; ++j) {
187     D column = slab((*j).x);
188     if ((*j).y ≥ slab_max[column] or (*j).y ≤ slab_min[column]) {
189         std::iter_swap(j, o);
190         ++o;
191     }
192 }
193 delete[] slab_min;
194 delete[] slab_max;
195 return o;
196 }
197
198 template<typename I>
199 I solve(I first, I past) {
200     I rest = prune(first, past);
201     I interior = plane_sweep::solve(first, rest);
202     return interior;
203 }

```

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

C. Micro-benchmarks

C.1 *multi_precision.h++*

```

1  /*
2   This dummy performs Graham's scan
3  */
4
5  #define MULTI_PRECISION
6
7  #include <functional>
8  #include <iterator>
9  #include "plane_sweep.h++"
10 #include "point.h++"
11 #include <utility>
12
13 namespace multi_precision {
14
15     template<typename I>
16     I solve(I first, I past) {
17         using P = typename std::iterator_traits<I>::value_type;
18         using T = typename P::coordinate;
19         std::pair<I, I> pair = plane_sweep::clean(first, past,
20             ↪ std::less<T>());
21         I top = std::get<0>(pair);
22         I next = std::get<1>(pair);
23         I interior = plane_sweep::scan(first, top, next, past);
24         return interior;
25     }
26 }

```

C.2 *double_precision.h++*

```

1  /*
2   This dummy performs Graham's scan
3  */
4
5  #define DOUBLE_PRECISION
6
7  #include <functional>
8  #include <iterator>
9  #include "plane_sweep.h++"
10 #include <utility>
11
12 namespace double_precision {
13
14     template<typename I>
15     I solve(I first, I past) {
16         using P = typename std::iterator_traits<I>::value_type;
17         using T = typename P::coordinate;

```

```

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

```

C.3 *floating_point_filter.h++*

```

1  /*
2   This dummy performs Graham's scan
3  */
4
5  #define FLOATING_POINT_FILTER
6
7  #include <functional>
8  #include <iterator>
9  #include "plane_sweep.h++"
10 #include <utility>
11
12 namespace floating_point_filter {
13
14     template<typename I>
15     I solve(I first, I past) {
16         using P = typename std::iterator_traits<I>::value_type;
17         using T = typename P::coordinate;
18         std::pair<I, I> pair = plane_sweep::clean(first, past,
19         ↪ std::less<T>());
20         I top = std::get<0>(pair);
21         I next = std::get<1>(pair);
22         I interior = plane_sweep::scan(first, top, next, past);
23         return interior;
24     }
25 }

```

C.4 *minmax_element.h++*

```

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

```

```

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

```

C.5 *partition.h++*

```

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

```

C.6 *copy.h++*

```

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

```



```

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

```

C.7 *sort.h++*

```

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

```

C.8 *lexicographic_sort.h++*

```

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

```

C.9 *angular_sort.h++*

```

1  /*
2   Sort all the points around o lexicographically by polar angle and
3   distance from o
4  */
5
6  #include <algorithm> // std::sort
7  #include <cstdint> // 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 }

```

C.10 *bsort.h++*

```

1  /*
2   This dummy sorts the points according to their x-coordinates
3  */
4
5  #include "bucketsort.h++"
6  #include <iterator> // std::iterator_traits
7
8  namespace bsort {
9
10     template<typename I>
11     I solve(I first, I past) {
12         using P = typename std::iterator_traits<I>::value_type;

```

```

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

```

C.11 *tsort.h++*

```

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

```

C.12 *shuffle.h++*

```

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

```

```
    ↪ std::linear_congruential_engine<N, 6364136223846793005U,  
    ↪ 1442695040888963407U, 0U>;  
10 constexpr N init = 10164167376618180197U;  
11 linear_congruential_engine generator{init};  
12  
13 namespace shuffle {  
14  
15     template<typename I>  
16     void solve(I first, I past) {  
17         std::shuffle(first, past, generator);  
18     }  
19 }
```

D. Helpers

D.1 *point.h++*

```

1  #ifndef __POINT__
2  #define __POINT__
3
4  #include <algorithm> // std::min std::max
5  #include <cassert> // assert macro
6  #include <cstddef> // std::size_t
7  #include <limits> // std::numeric_limits
8  #include <string> // std::string std::to_string
9
10 #if not defined(MULTI_PRECISION) and not
    ↪ defined(FLOATING_POINT_FILTER)
11 #define DOUBLE_PRECISION
12 #endif
13
14 #if defined(MULTI_PRECISION) or defined(FLOATING_POINT_FILTER)
15
16 #include "cphmpl/functions.h++" // cphmpl::width
17 #include "cphstl/integers.h++" // cphstl::Z
18
19 #endif
20
21 template<typename T>
22 class point {
23 public:
24
25     using self = point<T>;
26     using coordinate = T;
27
28     T x;
29     T y;
30
31     explicit point()
32         : x(0), y(0) {
33     }
34
35     point(T x_coordinate, T y_coordinate)
36         : x(x_coordinate), y(y_coordinate) {
37     }
38
39     std::string to_string() const {
40         return "(" + std::to_string(x) + ", " + std::to_string(y) + ")";
41     }
42
43     bool operator==(point<T> const& other) const {
44         return x == other.x and y == other.y;
45     }
46
47     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>
60     ↪ const& p) {
61     return os << p.to_string();
62 }
63 };
64 #if defined(MULTI_PRECISION) or defined(FLOATING_POINT_FILTER)
65 namespace multi_precision {
66
67     template<typename P>
68     bool left_turn(P const& p, P const& q, P const& r) {
69         using N = std::size_t;
70         using T = typename P::coordinate;
71         constexpr N w = cphmpl::width<T>;
72         using Z = cphstl::Z<2 * w + 4>;
73         Z p_x = Z(p.x);
74         Z p_y = Z(p.y);
75         Z lhs = (Z(q.x) - p_x) * (Z(r.y) - p_y);
76         Z rhs = (Z(r.x) - p_x) * (Z(q.y) - p_y);
77         return lhs > rhs;
78     }
79
80     template<typename P>
81     bool no_turn(P const& p, P const& q, P const& r) {
82         using N = std::size_t;
83         using T = typename P::coordinate;
84         constexpr N w = cphmpl::width<T>;
85         using Z = cphstl::Z<2 * w + 4>;
86         Z p_x = Z(p.x);
87         Z p_y = Z(p.y);
88         Z lhs = (Z(q.x) - p_x) * (Z(r.y) - p_y);
89         Z rhs = (Z(r.x) - p_x) * (Z(q.y) - p_y);
90         return lhs == rhs;
91     }
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 px = Z(p.x);
104    Z py = Z(p.y);
105    Z dx1 = Z(q.x) - px;
106    Z dx2 = Z(r.x) - px;
107    Z dy1 = Z(q.y) - py;
108    Z dy2 = Z(r.y) - py;
109    Z result = dx1 * dy2 - dx2 * dy1;
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 px = Z(p.x);
122     Z py = Z(p.y);
123     Z qx = Z(q.x);
124     Z qy = Z(q.y);
125     return (px - qx) * (px - qx) + (py - qy) * (py - qy);
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 L1distance(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 px = Z(p.x);
162     Z py = Z(p.y);
163     Z lhs = (Z(q.x) - px) * (Z(r.y) - py);
164     Z rhs = (Z(r.x) - px) * (Z(q.y) - py);
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 dx1 = mid_x - Z(p.x);
198         Z dx2 = Z(r.x) - mid_x;
199         Z dy1 = mid_y - Z(p.y);
200         Z dy2 = 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))  $\neq$  dx1 and Z(T(dy2))  $\neq$  dy2;
207     bool check2 = Z(T(dx2))  $\neq$  dx2 and Z(T(dy1))  $\neq$  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)  $\neq$  (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)  $\neq$  (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 dx1 = mid_x - Z(p.x);
259     Z dx2 = Z(r.x) - mid_x;
260     Z dy1 = mid_y - Z(p.y);
261     Z dy2 = Z(r.y) - mid_y;
262
263     bool check1 = Z(T(dx1))  $\neq$  dx1 and Z(T(dy2))  $\neq$  dy2;
264     bool check2 = Z(T(dx2))  $\neq$  dx2 and Z(T(dy1))  $\neq$  dy1;
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 (((dx1 > 0) == (dy2 > 0)) == ((dx2 > 0) == (dy1 >
270  $\hookrightarrow$  0))) {
271                 // the non-overflow part must also be the same
272                 return dx1 * dy2 == dx2 * dy1;
273             }
274         }
275         return false;
276     }
277     else {
278         return dx1 * dy2 == dx2 * dy1;
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             ++turns;
289         #endif
290
291         using Z = long long;
292         Z mid_x = Z(q.x);
293         Z mid_y = Z(q.y);
294         Z dx1 = mid_x - Z(p.x);
295         Z dx2 = Z(r.x) - mid_x;
296         Z dy1 = mid_y - Z(p.y);
297         Z dy2 = Z(r.y) - mid_y;
298         return (unsigned long long) (dx1 * dy2 - dx2 * dy1);
299     }
300 }
301 }
302
303 using namespace double_precision;
304
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

```

D.2 validation.h++

```

1  #ifndef __VALIDATION__
2  #define __VALIDATION__
3
4  #include <algorithm> // std::copy std::sort std::equal std::rotate
5  #include <cassert> // assert macro
6  #include <cstdlib> // std::size_t
7  #include <iostream> // std streams
8  #include <iterator> // std::iterator_traits
9  #include "point.h++" // left_turn right_turn
10 #include <vector> // std::vector
11
12 namespace validation {
13
14     template<typename I, typename J>
15     bool same_multiset(I p, I q, J r, J s) {
16         using P = typename std::iterator_traits<I>::value_type;
17         using S = std::vector<P>;
18         std::size_t n = q - p;
19         std::size_t m = s - r;
20         if (m  $\neq$  n) {
21             return false;
22         }
23         S backup;
24         backup.resize(n);
25         std::copy(p, q, backup.begin());
26         std::sort(backup.begin(), backup.end(),
27             [](P const& a, P const& b)  $\rightarrow$  bool {
28             return (a.x < b.x) or (a.x == b.x and a.y < b.y);
29             });
30         S other;
31         other.resize(n);
32         std::copy(r, s, other.begin());
33         std::sort(other.begin(), other.end(),
34             [](P const& a, P const& b)  $\rightarrow$  bool {
35             return (a.x < b.x) or (a.x == b.x and a.y < b.y);
36             });
37         return std::equal(backup.begin(), backup.end(), other.begin(),
38             [](P const& a, P const& b)  $\rightarrow$  bool {
39             return (a.x == b.x) and (a.y == b.y);
40             });
41     }
42
43     // [p, r) circular chain of vertices
44
45     template<typename I>
46     bool left_turns_only(I p, I r) {
47         std::size_t h = r - p;
48         if (h < 3) {
49             return true;

```

```

50     }
51     for (I q = p; q  $\neq$  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  $\neq$  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  $\neq$  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  $\neq$  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  $\neq$  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  $\neq$  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  $\neq$  s; ++i) {
337             if (above(east, hull.end(), *i)) {
338                 std::cerr << "Inside: " << *i
339                     << " above the upper hull (ccw)\n";
340                 return false;
341             }
342         }
343     }
344     return true;
345 }
346 }
347
348 #endif

```

D.3 bucketsort.h++

```

1  /*
2   This program implements bucketsort
3
4   Source: O. Nevalainen and T. Raita, An internal hybrid sort
5   algorithm revisited, The Computer Journal 35,2 (1992), 177–183
6
7   Author: Jyrki Katajainen © 2018
8
9   Worst-case running time:  $O(n \lg n)$ 

```

```

10 Parameter:  $m = \min(n/5, 2^{16})$ 
11 Average-case running time:  $O(n \lg(n/m))$ 
12 Extra space:  $n$  elements,  $m + O(\lg n)$  words
13
14 Observed sorting time per  $n \lg n$  [ns]; input: random permutation
15     1024     1.07
16     32768    0.82
17    1048576   1.17
18    33554432  2.04
19 */
20
21 #include <algorithm> // std::sort std::copy
22 #include <cassert> // assert macro
23 #include <cstdint> // std::ptrdiff_t
24 #include <iterator> // std::distance std::iterator_traits
25 #include <utility> // std::pair
26
27 namespace bucketsort {
28
29     constexpr std::ptrdiff_t threshold = 100;
30     constexpr std::ptrdiff_t max_m = (1 << 16);
31
32     template<typename V>
33     class identity {
34     public:
35
36         V operator()(V const& e) const {
37             return e;
38         }
39
40         V& operator()(V& e) const {
41             return e;
42         }
43     };
44
45     template<typename D, typename V, typename K = identity<V>>
46     class bucket {
47     public:
48
49         bucket(D m, V const& min, V const& max, K key = K())
50             : a(double(m - 1) / (double(key(max)) - double(key(min)))),
51               b(double(m - 1) * double(key(min)) / (double(key(max)) -
52 ↪ double(key(min)))),
53               key(key) {
54
55         D operator()(V const& x) {
56             return D(a * double(key(x)) - b);
57         }
58
59     private:
60

```

```

61     double a;
62     double b;
63     K key;
64 };
65
66 template<typename R, typename C, typename K = identity<typename
    ↪ std::iterator_traits<R>::value_type>>
67 void sort(R data, R past, C compare, K key = K()) {
68     using V = typename std::iterator_traits<R>::value_type;
69     using D = typename std::iterator_traits<R>::difference_type;
70     D const n = std::distance(data, past);
71
72     // small input
73
74     if (n < threshold) {
75         std::sort(data, past, compare);
76         return;
77     }
78     D m = std::min(n / 5, max_m);
79     V* copy = new V[n];
80     D* counter = new D[m + 1];
81
82     // initialization
83
84     std::pair<R, R> pair = std::minmax_element(data, past,
    ↪ compare);
85     R leftmost = std::get<0>(pair);
86     R rightmost = std::get<1>(pair);
87     if (key(*leftmost) == key(*rightmost)) {
88         return;
89     }
90     bucket formula(m, *leftmost, *rightmost, key);
91     for (D j = 0; j ≤ m; ++j) {
92         counter[j] = 0;
93     }
94
95     // determination of the number of elements in the buckets
96
97     for (D i = 0; i ≠ n; ++i) {
98         D j = formula(*(data + i));
99         counter[j] = counter[j] + 1;
100        copy[i] = *(data + i);
101    }
102    assert(counter[m] == 0);
103
104    // initialization of the cursors to the buckets
105
106    for (D j = 1; j ≤ m; ++j) {
107        counter[j] = counter[j - 1] + counter[j];
108    }
109
110    // construction of the buckets

```

```

111
112     for (D i = n - 1; i ≥ 0; --i) {
113         D j = formula(copy[i]);
114         counter[j] = counter[j] - 1;
115         data[counter[j]] = copy[i];
116     }
117
118     // sorting the buckets
119
120     for (D j = 0; j ≠ m; ++j) {
121         std::sort(data + counter[j], data + counter[j + 1], compare);
122     }
123     delete[] copy;
124     delete[] counter;
125 }
126 }

```

D.4 two_phase_bucketsort.h++

```

1  /*
2   This program implements two-phase bucketsort
3
4   Author: Jyrki Katajainen © 2018
5
6   Worst-case running time:  $O(n \lg n)$ 
7   Parameter:  $m$ 
8   Average-case running time:  $O(n)$ 
9   Extra space:  $O(\sqrt{n} + m)$  words
10
11  Observed sorting time per  $n \lg n$  [ns]; input: random permutation
12      1024      3.33
13      32768     1.92
14      1048576   2.09
15      33554432  2.41
16  */
17
18  #include <algorithm> // std::sort std::copy
19  #include "bucketsort.h++"
20  #include <cassert> // assert macro
21  #include <cmath> // std::sqrt
22  #include <cstddef> // std::ptrdiff_t
23  #include <iterator> // std::distance std::iterator_traits
24  #include <utility> // std::pair
25
26  namespace two_phase_bucketsort {
27
28      constexpr std::ptrdiff_t threshold = 100;
29
30      template<typename R, typename C, typename K =
31          ↪ bucketsort::identity<typename
32          ↪ std::iterator_traits<R>::value_type>>

```

```

31 void sort(R data, R past, C compare, K key = K()) {
32     using D = typename std::iterator_traits<R>::difference_type;
33     D const n = std::distance(data, past);
34
35     // small input
36
37     if (n < threshold) {
38         std::sort(data, past, compare);
39         return;
40     }
41     D m = D(std::sqrt(n));
42     D* counter = new D[m];
43     D* cursor = new D[m + 1];
44
45     // initialize the bucket formula
46
47     std::pair<R, R> pair = std::minmax_element(data, past,
48     ↪ compare);
49     R leftmost = std::get<0>(pair);
50     R rightmost = std::get<1>(pair);
51     if (key(*leftmost) == key(*rightmost)) {
52         return;
53     }
54     bucketsort::bucket formula(m, *leftmost, *rightmost, key);
55     for (D j = 0; j ≠ m; ++j) {
56         counter[j] = 0;
57     }
58
59     // determine the number of elements in the buckets
60
61     for (D i = 0; i ≠ n; ++i) {
62         D j = formula(*(data + i));
63         counter[j] = counter[j] + 1;
64     }
65
66     // initialize the cursors to the buckets
67
68     cursor[0] = counter[0];
69     for (D j = 1; j ≠ m; ++j) {
70         cursor[j] = cursor[j - 1] + counter[j];
71     }
72     cursor[m] = n;
73
74     // move the elements into the buckets
75
76     D home = m - 1;
77     for (D k = 0; k ≠ n; ++k) {
78         while (counter[home] == 0) {
79             --home;
80         }
81         D j = formula(data[cursor[home] - 1]);
82         if (home ≠ j) {

```

```
82         std::swap(data[cursor[home] - 1], data[cursor[j] - 1]);
83     }
84     counter[j] = counter[j] - 1;
85     cursor[j] = cursor[j] - 1;
86 }
87
88 // sort the buckets
89
90 for (D j = 0; j  $\neq$  m; ++j) {
91     bucketsort::sort(data + cursor[j], data + cursor[j + 1],
92     ↪ compare, key);
93 }
94 delete[] counter;
95 delete[] cursor;
96 }
```


E. Drivers

E.1 *check-driver.cpp*

```

1  #include "algorithm.h++" // NAME::solve NAME::check
2  #include <cassert> // assert macro
3  #include <iterator> // std::distance
4  #include "point.h++" // point
5  #include <vector> // std::vector
6
7  int main() {
8      using P = point<int>;
9      using S = std::vector<P>;
10     using I = typename S::iterator;
11
12     S bag{P(0, 0), P(0, 1), P(0, 2), P(0, 1)};
13     I rest = NAME::solve(bag.begin(), bag.end());
14     auto h = std::distance(bag.begin(), rest);
15     assert(h == 2);
16     assert(NAME::check(bag.begin(), bag.end()));
17 }

```

E.2 *test-driver.cpp*

```

1  /*
2   Performance Engineering Laboratory © 2017–2018
3
4   Brownie points:
5   11 Sep 2017: The test cases for overflow detection by
6   ↪ Marius–Florin Cristian
7  */
8  #include <algorithm> // std::copy std::equal
9  #include <cassert> // assert macro
10 #include <cstdlib> // std::rand std::size_t
11 #include <exception> // std::exception
12 #include <iostream> // std::cout std::cerr
13 #include <iterator> // std::iterator_traits
14 #include <limits> // std::numeric_limits
15 #include <random> // std::linear_congruential_engine
16 #include <vector> // std::vector
17
18 #include "algorithm.h++" // NAME::check
19 #include "point.h++" // point
20
21 using linear_congruential_engine =
22     ↪ std::linear_congruential_engine<unsigned long long,
23     ↪ 6364136223846793005U, 1442695040888963407U, 0U>;
24 unsigned long long const seed = 10164167376618180197U;
25 linear_congruential_engine random_number_generator{seed};
26

```

```

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  $\neq$  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  $\neq$  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 }

E.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,
35     ↪ 0U>;
36 constexpr unsigned long long seed = 10164167376618180197U;
37 linear_congruential_engine random_number_generator{seed};
38
39 #if defined(DISC)
40
41 template<typename I>
42 void generate(I p, I r) {
43     using P = typename std::iterator_traits<I>::value_type;
44     using T = typename P::coordinate;
45     T t = std::numeric_limits<T>::max();
46     std::uniform_int_distribution<T> distribution(-t, t);
47     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  $\neq$  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)  $\geq$  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  $\neq$  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  $\neq$  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  $\neq$  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  $\neq$  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)  $\rightarrow$  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  $\neq$  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  $\neq$  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 }
```

F. Makefile

F.1 makefile

```

1  CXX=g++
2  CXXFLAGS=-O3 -std=c++17 -x c++ -Wall -Wextra -fconcepts -DNDEBUG
3  IFLAGS = -I..
4
5  header-files:= $(wildcard *.h++)
6  implementations:= $(basename $(header-files))
7  square-tests:= $(addsuffix .square, $(implementations))
8  disc-tests:= $(addsuffix .disc, $(implementations))
9  universe-tests:= $(addsuffix .universe, $(implementations))
10 bell-tests:= $(addsuffix .bell, $(implementations))
11 special-tests:= $(addsuffix .special, $(implementations))
12 line-tests:= $(addsuffix .line, $(implementations))
13 sorted-tests:= $(addsuffix .sorted, $(implementations))
14 parabola-tests:= $(addsuffix .parabola, $(implementations))
15 turn-tests:= $(addsuffix .turn, $(implementations))
16 comp-tests:= $(addsuffix .comp, $(implementations))
17 move-tests:= $(addsuffix .move, $(implementations))
18 elimsquare-tests:= $(addsuffix .elimsquare, $(implementations))
19 elimdisc-tests:= $(addsuffix .elimdisc, $(implementations))
20 sanitychecks:= $(addsuffix .check, $(implementations))
21 unittests:= $(addsuffix .test, $(implementations))
22 benchmarks:= $(addsuffix .benchmark, $(implementations))
23
24 .PHONY: all clean find pilot veryclean
25
26 N = 1024 32768 1048576 33554432 #1073741824
27
28 $(square-tests): %.square : %.h++
29     @cp *.h++ algorithm.h++
30     $(CXX) $(CXXFLAGS) $(IFLAGS) -DNAME=$* driver.c++
31     @for n in $(N) ; do \
32         ./a.out $$n ; \
33     done ; \
34     rm -f algorithm.h++ ./a.out
35
36 $(disc-tests): %.disc : %.h++
37     @cp *.h++ algorithm.h++
38     $(CXX) $(CXXFLAGS) $(IFLAGS) -DDISC -DNAME=$* driver.c++
39     @for n in $(N) ; do \
40         ./a.out $$n ; \
41     done ; \
42     rm -f algorithm.h++ ./a.out
43
44 $(bell-tests): %.bell : %.h++
45     @cp *.h++ algorithm.h++
46     $(CXX) $(CXXFLAGS) $(IFLAGS) -DBELL -DNAME=$* driver.c++
47     @for n in $(N) ; do \
48         ./a.out $$n ; \
49     done ; \
50     rm -f algorithm.h++ ./a.out
51
52 $(universe-tests): %.universe : %.h++
53     @cp *.h++ algorithm.h++
54     $(CXX) $(CXXFLAGS) $(IFLAGS) -DUNIVERSE -DNAME=$* driver.c++
55     @for n in $(N) ; do \
56         ./a.out $$n ; \
57     done ; \
58     rm -f algorithm.h++ ./a.out
59
60 $(special-tests): %.special : %.h++

```

```

61     @cp *.h++ algorithm.h++
62     $(CXX) $(CXXFLAGS) $(IFLAGS) -DSPECIAL -DNAME=$* driver.c++
63     @for n in $(N) ; do \
64     ./a.out $$n ; \
65     done; \
66     rm -f algorithm.h++ ./a.out
67
68 $(line-tests): %.line : %.h++
69     @cp *.h++ algorithm.h++
70     $(CXX) $(CXXFLAGS) $(IFLAGS) -DLINE -DNAME=$* driver.c++
71     @for n in $(N) ; do \
72     ./a.out $$n ; \
73     done; \
74     rm -f algorithm.h++ ./a.out
75
76 $(sorted-tests): %.sorted : %.h++
77     @cp *.h++ algorithm.h++
78     $(CXX) $(CXXFLAGS) $(IFLAGS) -DSORTED -DNAME=$* driver.c++
79     @for n in $(N) ; do \
80     ./a.out $$n ; \
81     done; \
82     rm -f algorithm.h++ ./a.out
83
84 $(parabola-tests): %.parabola : %.h++
85     @cp *.h++ algorithm.h++
86     $(CXX) $(CXXFLAGS) $(IFLAGS) -DPARABOLA -DNAME=$* driver.c++
87     @for n in $(N) ; do \
88     ./a.out $$n ; \
89     done; \
90     rm -f algorithm.h++ ./a.out
91
92 $(turn-tests): %.turn : %.h++
93     @cp *.h++ algorithm.h++
94     $(CXX) $(CXXFLAGS) $(IFLAGS) -DNAME=$* -DMEASURE_TURNS driver.c++
95     @for n in $(N) ; do \
96     ./a.out $$n ; \
97     done; \
98     rm -f algorithm.h++ ./a.out
99
100 $(comp-tests): %.comp : %.h++
101     @cp *.h++ algorithm.h++
102     $(CXX) $(CXXFLAGS) $(IFLAGS) -DNAME=$* -DMEASURE_COMPS driver.c++
103     @for n in $(N) ; do \
104     ./a.out $$n ; \
105     done; \
106     rm -f algorithm.h++ ./a.out
107
108 $(move-tests): %.move : %.h++
109     @cp *.h++ algorithm.h++
110     $(CXX) $(CXXFLAGS) $(IFLAGS) -DNAME=$* -DMEASURE_MOVES driver.c++
111     @for n in $(N) ; do \
112     ./a.out $$n ; \
113     done; \
114     rm -f algorithm.h++ ./a.out
115
116 $(elimsquare-tests): %.elimsquare : %.h++
117     @cp *.h++ algorithm.h++
118     $(CXX) $(CXXFLAGS) $(IFLAGS) -DPRUNING -DNAME=$* driver.c++
119     @for n in $(N) ; do \
120     ./a.out $$n ; \
121     done; \
122     rm -f algorithm.h++ ./a.out
123
124 $(elimdisc-tests): %.elimdisc : %.h++
125     @cp *.h++ algorithm.h++

```

```

126     $(CXX) $(CXXFLAGS) $(IFLAGS) -DPRUNING -DDISC -DNAME=$* driver.c++
127     @for n in $(N) ; do \
128     ./a.out $$n ; \
129     done; \
130     rm -f algorithm.h++ ./a.out
131
132 $(benchmarks): %.benchmark : %.h++
133     @date 2>&1 | tee -a $.log
134     @make -i --no-print-directory $.turn 2>&1 | tee -a $.log
135     @make -i --no-print-directory $.comp 2>&1 | tee -a $.log
136     @make -i --no-print-directory $.move 2>&1 | tee -a $.log
137     @make -i --no-print-directory $.square 2>&1 | tee -a $.log
138     @make -i --no-print-directory $.disc 2>&1 | tee -a $.log
139     @make -i --no-print-directory $.bell 2>&1 | tee -a $.log
140     @make -i --no-print-directory $.elimsquare 2>&1 | tee -a $.log
141     @make -i --no-print-directory $.elimdisc 2>&1 | tee -a $.log
142     @date 2>&1 | tee -a $.log
143
144 cad:
145     @make -i --no-print-directory sort.benchmark
146     @make -i --no-print-directory plane_sweep.benchmark
147     @make -i --no-print-directory torch.benchmark
148     @make -i --no-print-directory quickhull.benchmark
149     @make -i --no-print-directory throw_away.benchmark
150
151 TESTFLAGS=-O3 -std=c++17 -Wall -Wextra -x c++ -fconcepts -g -DDEBUG
152
153 $(sanitychecks): %.check : %.h++
154     @cp $.h++ algorithm.h++
155     $(CXX) $(TESTFLAGS) $(IFLAGS) -DNAME=$* check-driver.c++
156     ./a.out
157     rm -f ./a.out
158
159 $(unittests): %.test : %.h++
160     @cp $.h++ algorithm.h++
161     $(CXX) $(TESTFLAGS) $(IFLAGS) -DNAME=$* test-driver.c++
162     ./a.out
163     rm -f algorithm.h++ ./a.out
164
165 # Other tools
166
167 clean:
168     - rm -f a.out temp algorithm.h++ 2>/dev/null
169
170 veryclean: clean
171     - rm -f *~ .*~ */*~ 2>/dev/null
172
173 find:
174     find . -type f -print -exec grep $(word) {} \; | less

```