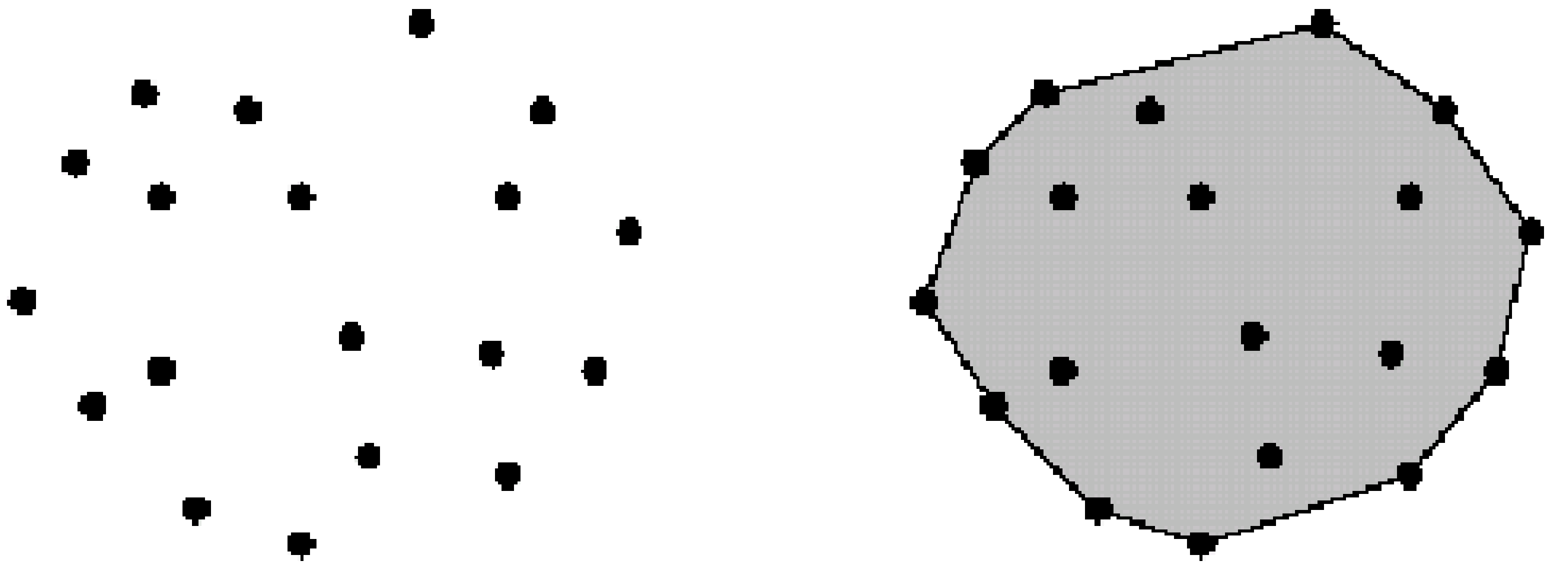
HW #4 Convex Hull殼

- **In this project (using C++, or Java or Python if you prefer), let us consider a fundamental structure in** computational geometry**, called the convex hull.**

- **Given a set of points, the convex hull is defined intuitively by surrounding a collection of points with a rubber band and letting the rubber band snap tightly around the points.**

HW #4 (2)



Convex hull

HW #4 (3)

- The (planar) convex hull problem is, given a set of *n* points *P* in the plane, output a representation of *P*'s convex hull.

- The convex hull is a closed convex polygon, the simplest representation is a counterclockwise enumeration of the vertices of the convex hull.

HW #4 (4)

- **A** clockwise **is also possible, We usually prefer** counterclockwise **enumerations, since they correspond to positive orientations, but obviously one representation is easily converted into the other.**
- **Ideally, the hull should consist only of extreme points, in the sense that if three points lie on an edge of the boundary of the convex hull, then the middle point should not be output as part of the hull.**

HW #4 (5)

- **In this project, the following classes and function headers are desired.**

```
#define PI          3.1415926535897931

//-- Class representing a point in 2D, (x, y)
class Point { public:
    Point() : x(0), y(0) {};
    Point (double ix, double iy) : x(ix), y(iy) {};
```

HW #4 (6)

```
    Point (const Point &p) : x(p.x), y(p.y) {};

    bool operator==(const Point & p) {
        return (x == p.x) && (y == p.y);
    }

    double x;
    double y;
};
```

HW #4 (7)

```
//-- Class representing a line in 2D,
//-- a*x + b*y + c = 0

class Line { public:
   Line() : a(0), b(0), c(0) {};
   Line(double ia, double ib, double ic) :
         a(ia), b(ib), c(ic) {};
   Line(const Line &p) : a(p.a), b(p.b), c(p.c) {};
```

HW #4 (8)

```
    bool operator==(const Line & p) {
        return (a == p.a) && (b == p.b) &&
               (c == p.c);
    }

    double a;
    double b;
    double c;
};
```

HW #4 (9)

//-- A class for finding convex hull of a set of
//-- points. Input is a const pass by reference
//-- vector<Point>, output is a pass by reference
//-- vector<Point>.

Class Convexhull { public:

//-- Find the convex hull of input points.
//-- This function will be implemented by you.

HW #4 (10)

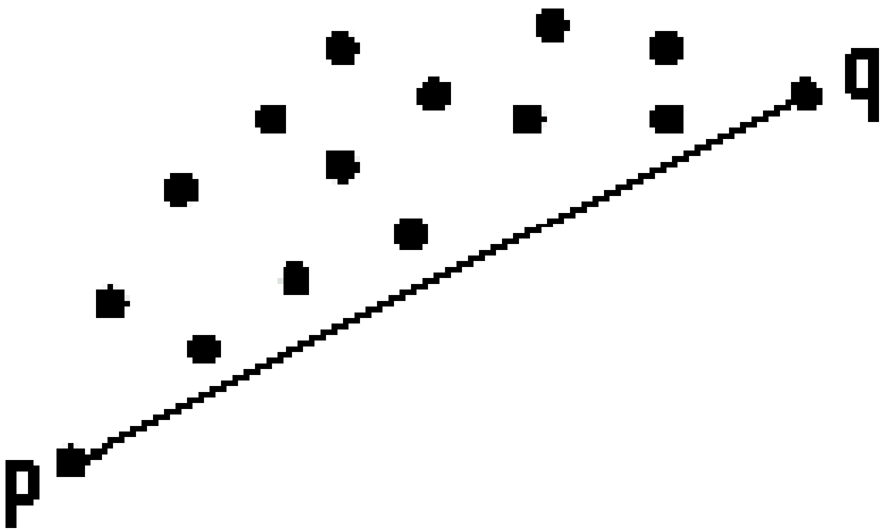
void FindConvexHull (const vector<Point>& input, vector<Point>& output);

private:

//-- A function returns a directed line *pq* from
//-- point *p* to *q*

Line FindLine(const Point& p,
const Point& q);

HW #4 (11)



All other points are on one side if *p* and *q* is on convex hull

HW #4 (12)

//-- A function returns true if *r* is on the right

//-- side of the line from *p* to *q*.

//-- If *pq* is horizontal, returns true only if *r* is

//-- above it.

```
bool isOnRight (const Point& p,
      const Point& q, const Point& r);
```

HW #4 (13)

//-- Find the lowest point (the point with

//-- smallest y-coordinate) among all input

//-- points in O(*n*). This function will be

// -- implemented by you.

Point FindLowestPoint (vector<Point>& input);

HW #4 (14)

```
//-- Find the angle pqr of three points.
//-- This function will be implemented by you.

double ComputeAngle (const Point& p,
        const Point& q, const Point& r);
};
```

HW #4 (15)

- **To find a convex hull, there is a simple O(n^3) brute force convex algorithm, which operates by considering each ordered pair of points (p, q), and determining whether all the remaining points of the set lie on the one side of the directed line pq from p to q.**

1. Implement the function FindConvexHull-BF using the brute force **algorithm stated above.**

HW #4 (16)

- **You should time your code, for example, calling *gettimeofday()*, and report timing information in** microseconds**.**

- **To simplify the problem, you do not need to sort the output in counterclockwise direction, and we assume that no 3 points are lying on the same straight line.**
- **Note that the output vector should not contain redundant point.**

HW #4 (17)

- **The pseudo code of the brute force algorithm is followed:**

for each point *p* in input

for each other point *q* not equal to *p* in input

find the line *pq*

for each other point *r* not equal to *p* and *q* in input

check if *r* is lying on the right side of *pq*

end

HW #4 (18)

```
        if all r are lying on right side of pq
                push p and q into output if they are not in
                output
        end
    end
end
```

HW #4 (19)

- **The previous algorithm is slow and the output is unsorted.**
- **Now we will consider is an O(n^2) sorting algorithm called SelectionSort.**

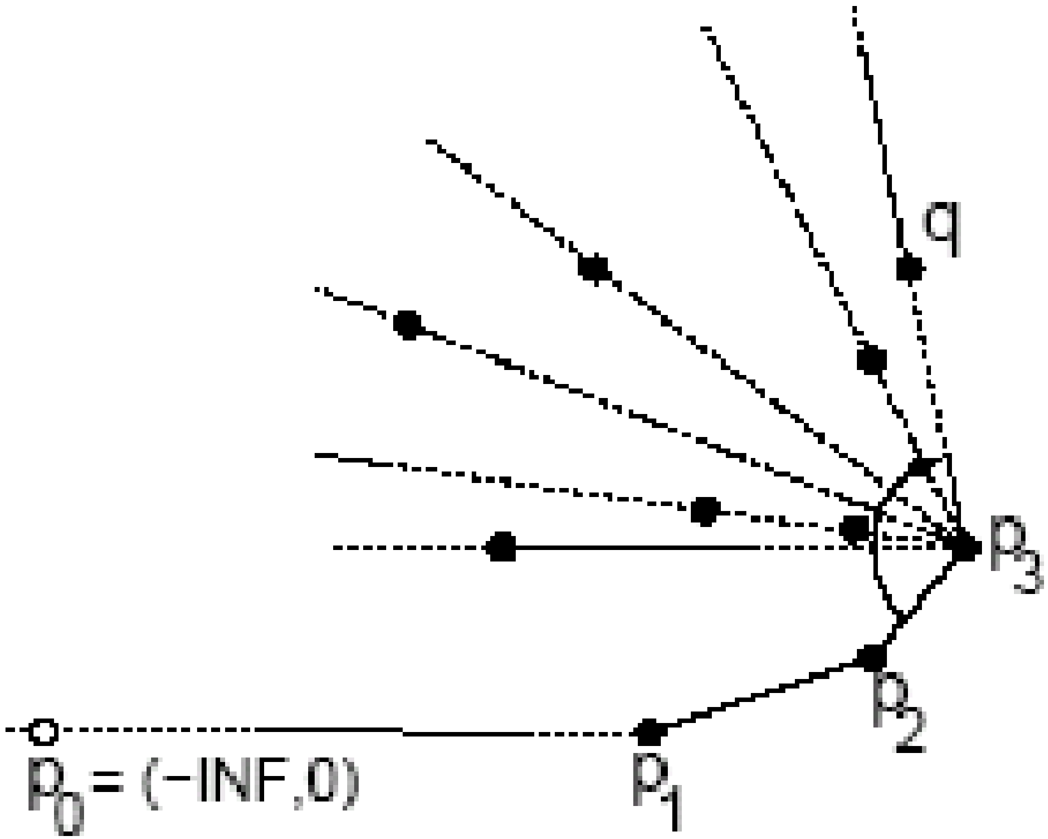
- **For sorting, this algorithm repeatedly finds the next element to add to the sorted order from the remaining items. The corresponding convex hull algorithm is called Javis's march, which builds the hull in O(nh) time by a process called gift-wrapping.**

HW #4 (20)

- **The algorithm operates by considering any one point that is on the hull, say, the lowest point. We then find the next edge on the hull in counterclockwise order.**

- **Assuming that p_{k-1} and p_k were the last two points added to the hull, compute the point r that maximizes the angle $p_{k-1}p_kr$.**
- **Thus, we can find the point r in O(n) time.**

HW #4 (21)



Javis's march algorithm, the next point on convex hull is the point has largest angle with the p_{k-1} and p_k

HW #4 (22)

- **After repeating this *h* times, where *h* is the number of output, we will return back to the** starting point **and we are done. Thus, the overall running time is O(*nh*).**

- **One technical detail is how we find an edge from which to start. One easy way to do this is to let p_1 be the point with the lowest y coordinate, and let p_0 be the point (-∞, lowest-y-coordinate), which is infinitely far to the right.**

HW #4 (23)

- The point p_0 is only used for computing the initial angles, after which it is discarded.
- The pseudo code of Javis's march algorithm is followed:

Find the lowest point of input to be the initial point

Initial

p = (-infinite, lowest-y-coordinate)

q = initial point

r = any input point not equal to *q*

HW #4 (24)

While *r* is not equal to initial point

Find *r* among all input points that maximizes the angle *pqr*

Push *r* onto output

Update

$p = q$

$q = r$

end

HW #4 (25)

- **We assume that no 3 points are lying on the same straight line.**
- **You can first find the line *pq* and *rq*, and then compute the angle by dot product of directional vector. Beware of the direction of the line.**

2. Implement the function FindConvexHull-JM, using the Javis's march algorithm.

HW #4 (26)

- **You should time your code, for example, calling *gettimeofday()*, and report timing information in** microseconds**.**

- **The final part is the display of convex hull on screen so that user may view it graphically.**

- We shall provide a program to plot graph of convex hull in OpenGL window.

HW #4 (27)

- Input.txt starts with an integer n telling the number of points in the set, and continues with n lines of (x, y) pairs representing coordinates in 2D.

11

100 100

0 450

350 0

0 320

350 -250

-200.5 250.5

-450.5 -380.5

500 0

-500.5 353.5

400 350

-380 -250

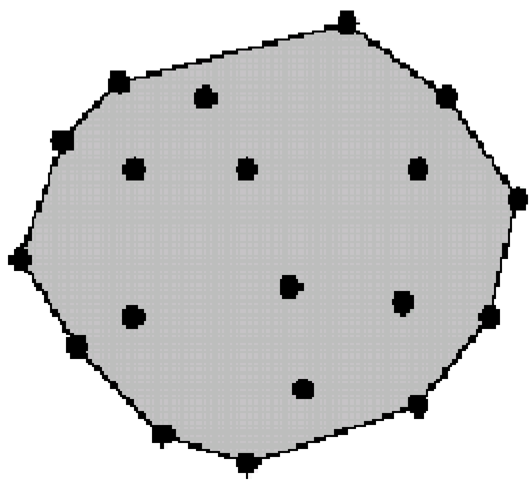
HW #4 (28)

- ***Draw-Points (const vector<Point>& input)***

//-- Each point is displayed at a size of at least

//-- one pixel.

//-- Provide data to the first part of Output.txt



HW #4 (29)

- ***Draw-Lines (const vector<Point>& input)***

  //-- Successive pairs of points are displayed as

  //-- endpoints of individual line segments.

  //-- Provide data to the second part of Output.txt

3. Implement the functions Draw-Points and Draw-Lines.

HW #4 (30)

- The first part of data in Output.txt is used to draw the subset of points lying inside the hull. It starts with an integer n_1 telling the number of points, and continues with n_1 lines of (x, y) pairs.
- The second part of data in Output.txt is used to draw a sequence of connected line segments joining the hull vertices (extreme points). It starts with an integer n_2 telling the number of vertices, and continues with n_2 lines of (x, y) pairs in counterclockwise order.

5

350 0

100 100

0 320

-200.5 250.5

-380 -250

6

500 0

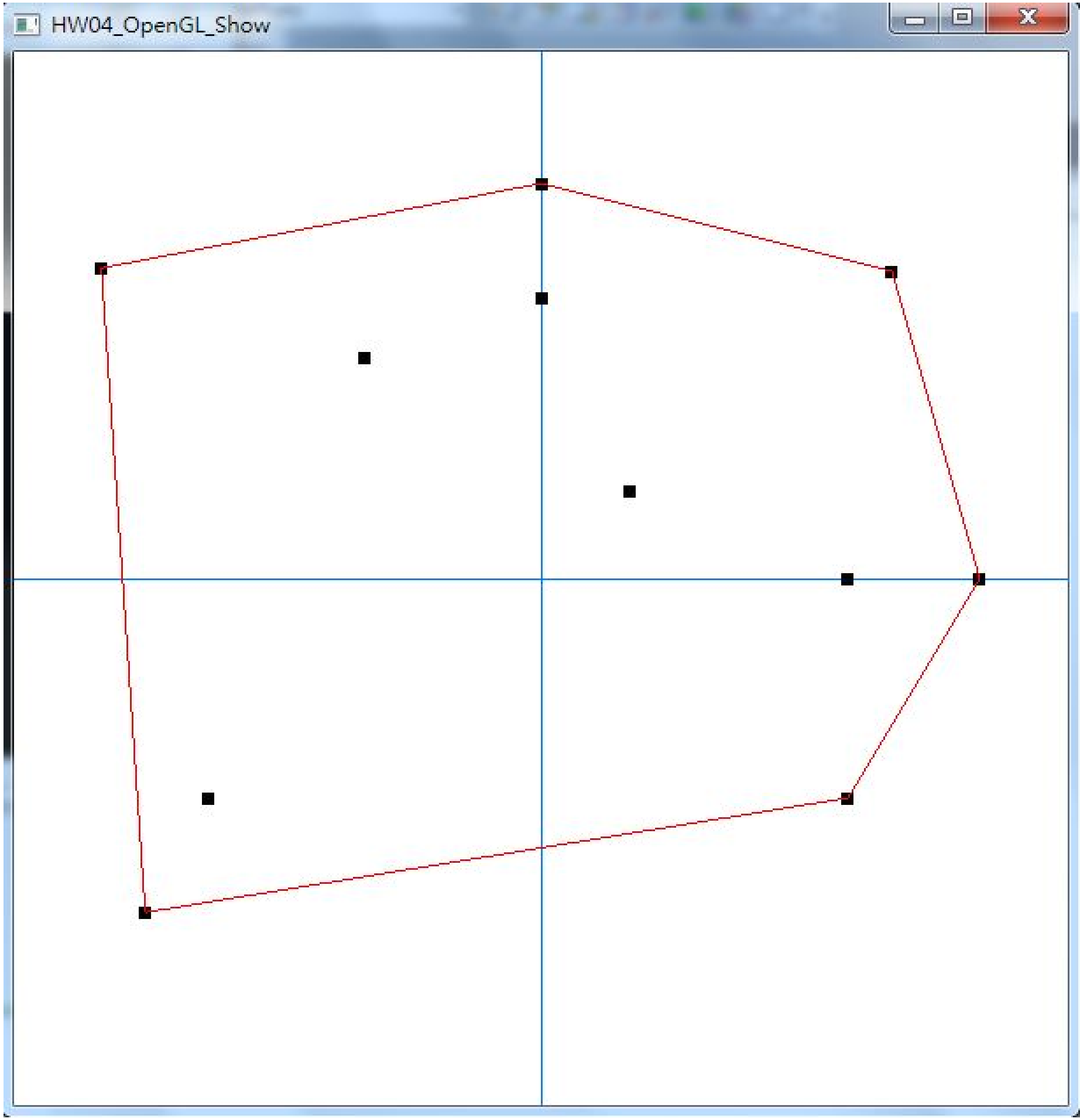
400 350

0 450

-500.5 353.5

-450.5 -380.5

350 -250



HW #4 (31)

- Note that:

1. $n = n_1 + n_2$.
2. Using square brackets ([ ]) to retrieve vector elements does **not** perform bounds checking; using member function at to retrieve vector elements does perform bounds checking.