

# Solution of the test

*Grishchenko Dmitry*

## Problem 1

In this problem, we consider Delaunay triangulation algorithm described in [1][Chapter 9] for the specific set of points  $P = \{p_1, \dots, p_n\}$ , where each  $p_i$  is a 2-dimensional point with **Int32** coordinates.

To answer on the proposed question, we need to understand which lines of the algorithm requires for “longer” datatype.

First, in the algorithm we find two additional points  $p_{-1}$  and  $p_{-2}$  that could be larger than Int32; more precisely, if  $P$  consists of all 4 limit points of Int32 for  $p_{-1}$  we will need to work with  $3 \times \text{INT\_MIN}$  that requires for Int64 datatype.

Another 2 operations that are performed by the algorithm are :the procedure of legalizing the edge and the test whether the given point is inside or on the border of the triangle.

The only hard part of the legalizing procedure is to check if the point is inside of the circumcircle of the Int32 triangle. Let us first provide an algorithm for this check and then come up with a conclusion on the “size” of the long int required for it.

Since we are working with integer types only, the circumcircle could not be interpreted as a pair of its center and radius for this validation test. However, there is an integer way to check if the point inside of the circle. Let us consider triangle  $(p_1, p_2, p_3)$  and  $x_i$  is a point to check (see Figure 1). (As we could see from the figure, points  $y_i$  are outside of the circle.)

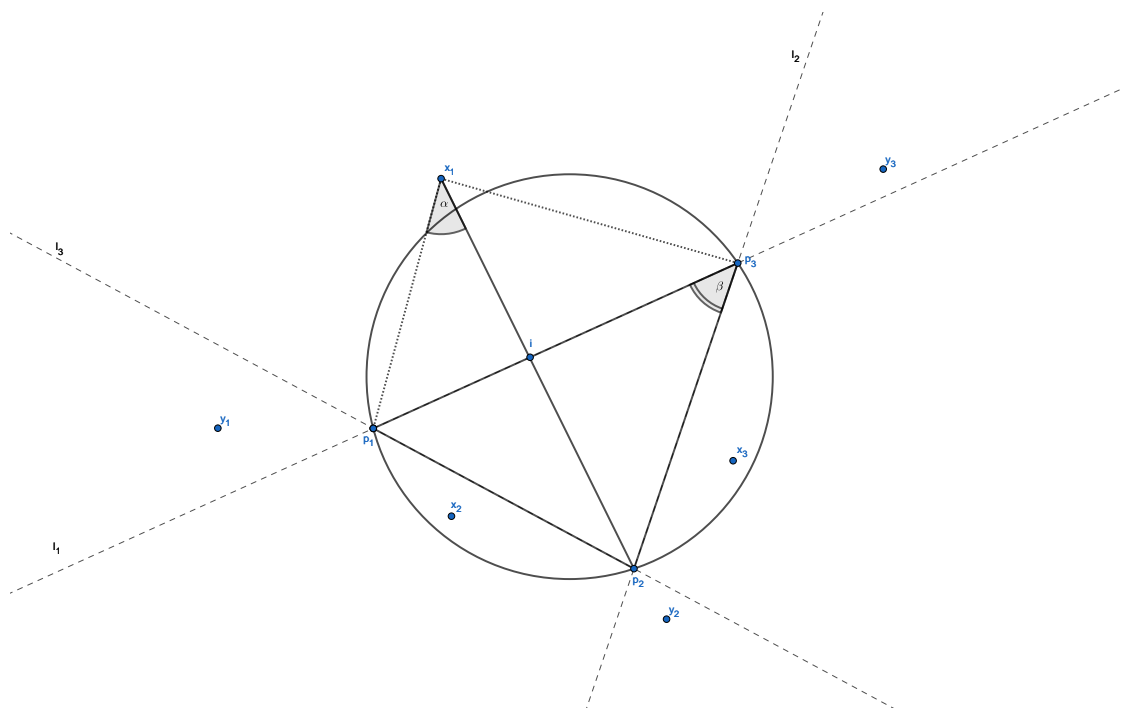


Figure 1

First, let us check if the point is inside of the triangle. If point  $x$  is in the triangle  $(p_1, p_2, p_3)$  then the

sum of areas of  $(p_1, p_2, x)$ ,  $(p_1, p_3, x)$ , and  $(p_2, p_3, x)$  is equal to the area of  $(p_1, p_2, p_3)$ ; moreover, if one of the areas is equal to 0 the point is on the corresponding edge of the triangle. So this test requires Int96 if the outer product is used to compute the area (an absolute value of (1)).

If the point is outside of the triangle let us check in which “sector” of the plane it is; in other words we need to understand if the point of the type  $y$  or of the type  $x$  and furthermore to specify the subscript. To do it, for every line  $l_i$  let us understand if the point is “above” the line or not. For this, let us calculate the oriented area of triangle  $(p_1, x_2, p_3)$  and if it is positive, then the point  $x_2$  is above the line (if it is 0 it is on the line and if it is negative it is under the line). By the definition the doubled signed area is

$$2S = (x^2 - x^1)(y^3 - y^2) - (x^3 - x^2)(y^2 - y^1). \quad (1)$$

where  $p_1 = (x^1, y^1)$ ,  $p_2 = (x^2, y^2)$ , and  $p_3 = (x^3, y^3)$ . It is easy to see, that  $2S$  could be calculated using Int128. In the very special case when the point is on the line an easy check if the point is between the ends of the segment would answer to the question if the point is inside of the circle.

Without loss of generality let us consider that our point is  $x_1$ . From the school geometry classes it is known, that for point  $x_1$  the necessary and sufficient condition to be inside of the circle (or on the border) is : angle  $\alpha$  is greater or equal than  $\beta$ . This corresponds to  $\cos(\beta) \geq \cos(\alpha)$ .

Since the division procedure does not preserve integers to compare two cosines we should compare the following terms (the sign  $\leq$  implies that point  $x_1$  is inside of the circle.)

$$\langle \overline{x_1 p_1}, \overline{x_1 p_3} \rangle \|\overline{p_2 p_1}\|_2 \|\overline{p_2 p_3}\|_2 \quad \text{Vs} \quad \langle \overline{p_2 p_1}, \overline{p_2 p_3} \rangle \|\overline{x_1 p_1}\|_2 \|\overline{x_1 p_3}\|_2.$$

However, since the distance is not integer we should operate with a squares of both sides

$$\underbrace{(\langle \overline{x_1 p_1}, \overline{x_1 p_3} \rangle)^2}_{\text{Int128} + \text{small amount} \leq 6 \text{ Int64} + \text{small amount} \leq 3} \underbrace{\|\overline{p_2 p_1}\|_2^2}_{\text{Int128} + \text{small amount} \leq 6 \text{ Int64} + \text{small amount} \leq 3} \|\overline{p_2 p_3}\|_2^2 \quad \text{Vs} \quad (\langle \overline{p_2 p_1}, \overline{p_2 p_3} \rangle)^2 \|\overline{x_1 p_1}\|_2^2 \|\overline{x_1 p_3}\|_2^2.$$

that requires for **Int512** (or **Int268** if the specific types are allowed).<sup>1</sup>

### Better solution.

Let us recall the following theorem.

**Theorem 1.** *If  $a, b, c$ , and  $d$  form a convex polygon then  $d$  lies in the circle determined by  $a, b$ , and  $c$  iff :*

$$\begin{vmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{vmatrix} > 0 \quad (2)$$

Since we know the position of point we could work with convex polygon (e.g., for  $x_1$  the polygon would be  $a = p_1, b = p_2, c = p_3$ , and  $d = x_1$ ). Now, we could see that it is enough to calculate the sign of the determinant. Unfortunately for this we need to calculate determinant itself and for this we need to figure with objects of type  $a_x * b_y * (c_x^2 + c_y^2)$  that requires **Int192**.

### Another solution

Finally, we could calculate the intersection of diagonals  $i$  of the polygon  $x_1, p_1, p_2, p_3$  and check if

$$\|\overline{x_1 i}\|_2 \|\overline{p_2 i}\|_2 > \|\overline{p_1 i}\|_2 \|\overline{p_3 i}\|_2$$

that will correspond to the outer position of point  $x_1$ . However the intersection of two lines is rational with numerator and denominator both being bigger than Int64. That requires scaling of ALL points to have

---

<sup>1</sup>Coordinates of  $\overline{x_1 p_1}$  (the same is true for all the other vectors) could be twice bigger than the limits of Int32 so for the scalar product we should use Int64 + 3 extra bits.

integers that leads to the bigger than Int128 for the intersection and bigger than Int96 for the vertices. Finally, to operate with integers we should consider squares of the distances instead of the distances itself that leads to the **Int512** at least that is worse than in previous case so we won't provide a precise analysis here..

**Answer.**

In conclusion, the most “size-consuming” operation requires **Int192**.

## Problem 2

In this problem, we create class *cRegularPolygon* (class *tOctet* in the code) that corresponds to the convex polygon with regular 45 edges. The object of this class should support all the provided public methods. Moreover, we provide some unit-tests to show that all the methods works correct. (We consider that all the points are integers and all the intersections of edges are integers as well so we do not need to provide the tools for odd-point analysis)

**Structure of the class.** Every object of the class could be represented by its edges; however this is not useful for the required methods. To make it more practical, we propose representing the object by *limits* that are more operation-tolerant. More precisely, every regular direction line corresponds to the equation  $ax + by = c$ , where  $a$  and  $b$  depend only on the direction, however  $c$  corresponds to the *limit*. Furthermore, we consider the limit of *cRegularPolygon* in direction *eDir*  $d$  to be a limit of edge that has  $d + 2$  direction while moving in anticlockwise direction.

Using this structure any regular polygon could be represented (starting from point (0 edges) up to the full octagon (8 edges)).

**Constructors.** To make the class useful we should equip it with different constructors. The default one that creates an object with all limits equal to 0. Second type of constructor is by point (class *tPoint*) that is a pair of 2 Int32 coordinates  $(x, y)$ . This constructor should calculate the limits of the point and fill the attribute of the class. Third, we consider constructor by the pair of points (class *tSegment* that is an ordered pair of 2 *tPoints*.) We also provide a constructor by any list of points that creates octet by point and then uses *Combine* (to be described after) procedure to add all the points from the list. More precisely, this constructor creates the minimal octet that contains all the points. Finally, we provide the copy constructor. Since we are not allocating the memory by *new* we do not need different from default destructor.

**Auxiliary classes.** As we mentioned above, class *tOctet* considers two auxiliary classes *tPoint* and *tSegment*. As we already said, we would need to calculate the limits of these objects to construct *tOctet* so we provide this functionality to these classes. Moreover, we add basic operators to operate with points like with vectors ('+', '-' and scalar multiplication). Also we add functionality of the shift (add *tPoint*) to the *tSegment* object. Furthermore, we need to operate with *tSegments* objects in future so we provide *direction* attribute that is a single (the pair of them ) for the regular segment and a pair of neighbour directions for the odd-angle one.

**Realization of functions.** For all functions for the simplicity we assume that all the points has small coordinates so the arithmetical operations such as sum and difference of two points returns a point as well. It is quite practical assumption and it makes all algorithms easier. We understand, that some additional functionality should be added to work with real Int32 points; however for this test we assume it is unnecessary.

Moreover, we consider all the diagonals to intersect in integer points to simplify the work and get rid of oddpoint type.

**Cover point.** The action of this function is quite easy, we just make all the limits of octet that are smaller then the corresponding limits of the point to be the same as for the point.

To check that point **Is inside** of the octet we check if all it's limits are not greater than the corresponding limits for the octet. If one (or some) of them are equal to the octet's one than the point is on the edge (is vertex).

**Common point.** Function common point works for 2 neighbour directions for now. Using the switch-case construction it uses an explicit formula for the every pair of the neighbour directions.

**Has edge.** In this function we check that the limit is smaller than the sum(for odd direction) or average (for orthogonal direction) of two neighbour directions.

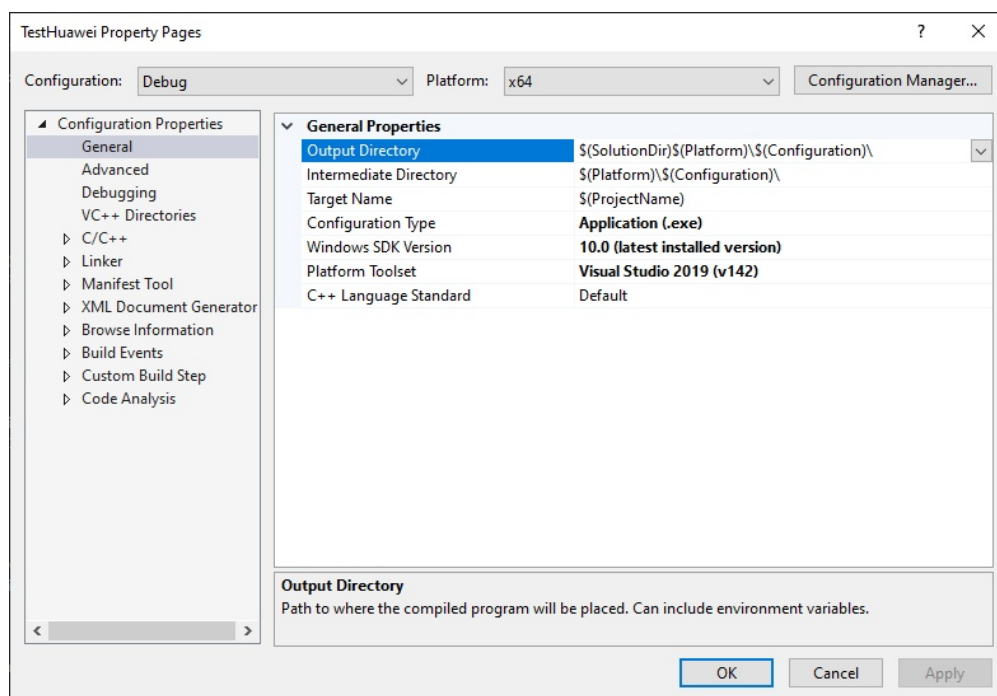
**Intersect.** This function returns `std::shared_ptr<Octet>` to avoid memory leaks. The intersection if it exists is an octet with every limit be the minimal of the corresponding limits of these two octets. However, the intersection is not empty if for all directions  $oct1.Limit(dir) + oct2.Limit(dir+4) \geq 0$ .

**Cure.** Is an auxiliary function that postproceed the intersection procedure to avoid stripes. It assumes that octet is non-empty and consists of 2 steps. First is to avoid big diagonal limits that is for all diagonal limit we make it equal to the sum of the neighbour limits if it is bigger. Second, for all orthogonal limits (assume E) we calculate the limit of orthogonal directions (N and S) in 2 different ways: using  $Limit(S)/Limit(N)$  for the octet and using  $(+)(Limit(NE)-Limit(SE))/2$ . In case when the second value is bigger we make the corresponding diagonal and basic orthogonal (E) limits smaller to fit.

**Inflate.** This easy procedure is just add the required amount for limits (inflate parameter for orthogonal) and the scaled by  $\sqrt{2}$  for diagonal. However, there are two points. First, if inflate parameter is negative we do inflation if and only if the octet is inflatable that is if we proceed the procedure the octet should not become empty. Second, if we perform inflate with parameter  $d$  and then with parameter  $-d$  the result should be the same as before inflation that calls for the correct rounding in case of negative inflation.

## General remarks.

The project is created in MS VS 2019 and contains boost library for unit-tests. The library is a part of the project and for the correctness of the compilation the following general properties should be used.



## References

- [1] De Berg, M., Cheong, O., Van Kreveld, M., Overmars, M.: Computational geometry: Algorithms and applications. santa clara (2008)