**Distance round #2 / Base/ November 2019**

**Problem 1. VICTORY-PRO**

The basic idea of solving this problem is based on the following fact: in order to draw a unit square, you need to have either one triangle of type 1 and 2, or one triangle of type 3 and 4. Thus, from *a*1 triangles of type 1 and *a*2 triangles of type 2 it is possible to make no more than min{*a*1, *a*2} squares, and from *a*3 triangles of type 3 and *a*4 triangles of type 4- min{*a*3, *a*4} squares.

Now, to solve the problem, you need to determine what the maximum square can be composed of *K* = min{*a*1, *a*2} + min{*a*3, *a*4} unit squares. To do this, you can either iterate over all the squares of natural numbers

(1, 4, 9, 16, ...) until you meet a number greater than *K,* or extract the square root of *K* and round the result to the nearest integer down. It is necessary to take into account the low accuracy of the operation of taking a square root from a large number in some programming languages, so you need to check the result by inverse squaring or use a type with extended accuracy.

In the first case, the complexity of the algorithm will be О(*К*1/2), and in the second-O(1).

**Problem 2. Online registration**

When solving this problem, you must first select the school numbers from each record with the name of the school and form an array containing these numbers. Then, using the resulting array, determine the number of schools and school numbers that occur in it no more than five times.

When you select a school number from its name s, you should move sequentially along the line s from left to right until you meet a digit. This can be done using the following program snippet:

i = 0

**while** s[i] **not in** '0123456789':

 i += 1

start = i

**while** i < len(s) **and** s[i] **in** '0123456789':

 i +=1

finish = i

After executing this fragment of the program, the substring of the string *s* from the character with the number start to the character with the number finish (not inclusive) is the desired school number. You should select this number and store it in the array *sch*.

Since the school number can be very large, you should not convert it to a numeric data type, but save it as a string. The resulting array of school numbers should be sorted in lexicographic (alphabetical) order. Now, to get the desired answer, it remains to count how many times each school meets in the array, and form a list of schools that meet no more than 5 times. This can be done, for example, using the following program snippet:

count = 1

result = 0

answer = []

**for** i **in** **range**(1, **len**(sch)):

 **if** sch[i] == sch[i – 1]:

 count += 1

 **else**:

 **if** count <= 5:

 result += 1

 answer.append(sch[i-1])

 count = 0

**if** count <= 5: # не забываем последнюю школу

 result += 1

 answer.append(sch[-1])

**Distance round #2 / Advanced / November 2019**

**Problem 1 Robot tournament**

Let us first analyze a partial solution based on the assumption that all problems are evaluated equally. Imagine each task as a segment on the time axis. Then the problem is reduced to the classical one: from a given set of segments on a straight line, choose the largest number of segments that do not have common points.

The following algorithm can be used to solve this problem. First, choose from all the segments the one that ends to the left of all. From the segments that start to the right of it, again choose the segment that ends to the left of all, etc.

It is not difficult to determine that the implementation of this algorithm has asymptotic complexity O(*N*2). A more efficient solution can be obtained by sorting all the beginnings and ends of the segments and going from left to right through the sorted array. Such a solution, taking into account the use of fast sorting, has already asymptotic complexity O(*N* log *N*).

To solve the initial problem in the General case, when each problem at the interregional Olympiad is estimated by its number of points, different approaches can be used. Some of them are discussed below in the presented solutions.

The first version of the solution. It is based on the use of dynamic programming method and is as follows. Let's sort all the problems proposed at the tournament by the time of completion of their solution. Let *a*[*i*] be the maximum number of points that can be scored by solving only problems from among the "first" *i* problems. Consider the *i*-th problem. Let *j* be the number of the last problem that ends no later than the *i*-th problem is given (that is, *sj* + *tj* ≤ *si, a sj+1* + *tj+1* > *sj*). Then

*a*[*i*] = max(*a*[*i* – 1], *a*[*j*] + *c*[*i*]).

It follows from this that one can either solve the problem with number *i* and some set of problems with numbers not greater than *j*, or not solve the problem with number *i*. The Answer to the question will be the number *a*[*n*].

It remains to explain how to find the number *j*. This can be done in several ways:

* iterating over all numbers from 1 to *i* (solution with asymptotic complexity O(*N*2));
* binary search on the interval from 1 to *i* (complexity O(*N*log *N*));
* • using the "two pointers" method, that is, remembering the previous *j* and searching for each next *j* on the segment from the previous j, since the new value will be greater than or equal to the previous one (thus, to calculate all *a*[*i*] of each candidate, we will check each *j* only once and the complexity of such a solution will be equal to O(*N* log *N*), taking into account sorting).

Restore response. In addition to finding the highest amount of points that John will score in the tournament, it is also required to display the number and list of tasks, the solution of which will allow him to score such a sum. This can be done by one of the standard ways to recover a response in dynamic programming. For example, in addition to the array a, we will store an additional array *prev*, in which we will specify what choice was made at each step. If *a*[*i*] = *c*[*i*] + *a*[*j*], then in *prev*[*j*] we write *j.* this will mean that the problem with the number *i* is solved, and the previous problem has a number no greater than *j*. Otherwise, the number -1 must be written to *prev[i]*. Now, going from the end of the array *prev*, you can restore the list of tasks to be solved.

The second solution. When implementing this option, first sort together all the beginnings and ends of the segments, keeping for each point its type (beginning or end) and the number of the segment to which it belongs. We will consider sorted events from left to right. In the variable *best*, we will store the best result that can be obtained using only tasks whose execution time has already ended by the current moment ("finished" tasks). If the next point is the beginning of the segment with the number *i*, then write in *b*[*i*] the current value of the variable *best*. Thus, *b*[*i*] is the best result that can be obtained before solving the *i*-th task. If the next point is the end of the *i*-th segment, it is necessary to update the value of the variable *best* as follows: if the *i*-th task is solved, then *best* = *b*[*i*] + *c*[*i*], otherwise the value of the variable *best* does not change. Of the two options, you need to choose the one in which the value of the variable *best* is the largest, that is, *best* = max{*best*, *b*[*i*] + *c*[*i*]}

**Problem 2 The way home**

If we consider small constraints, the solution of the problem can be as follows. Let's construct the graph corresponding to the city without building, thus intersections will correspond to vertices of the graph, and roads – to edges. Then we remove all edges in the graph that fell into built-up quarters (the complexity of this procedure is equal to O(*n*  \* square of the city)). In the resulting graph, let's start a search in width from point (0, 0) to determine the distances to all possible locations of the future Mayor's house, and choose the closest one from them. By restoring the path when implementing a width search in the standard way, you can find the path rotation points.

If the area of the city is large, and there are not many building blocks, you can use the *compression of coordinates.* To do this, when constructing the graph, we will use only those horizontal and vertical lines on which the city hall, future houses or the boundaries of the building blocks are located.

In the case of maximum constraints, the solution of the original problem can be as follows. We will solve the problem separately for each possible location of the future house of the Mayor. Let the house in question have coordinates (*xi*, *yi*). We assume that *xi* ≥ 0, *yi* ≥ 0. It is easy to understand what needs to be changed in the solution to consider and cases of negative coordinates.

Consider the ways out of city hall to the North. There are two possible ways:

1) we go North, then turn right (East), and then left (North again); in this case, any segment of the path can have a length equal to 0;

2) we go to the North, crossing the horizontal street on which the Mayor's house will be located, then turn right (to the East), and then right again (back to the South).Сравнивая эти два пути можно сказать, что первый путь всегда короче второго.

Next, find out how far we can travel from point (0, 0) to the North (see Fig. 1). To do this, let's go through all the building blocks and cross them with a beam coming out of the origin to the North. Let us be able to travel unhindered to the point (0, *R*). If the house in question is on this segment, then the problem is solved.

Similarly, consider a ray coming from point (*xi*, *yi*) to the South, and find on it a point (*xi*, *S*) with a minimum positive ordinate *S,* such that from this point we can easily reach the Mayor's house. If *S* > *R*, then there is no path of the first type. Otherwise, we try to find the minimum value of *t* on the segment [*S*, *R*], such that we can travel from point (0, *t*) to point (*xi*, *t*). To do this, consider all the quarters that intersect with the band 0 ≤ *x* ≤ *xi*. Consider their projections on the *Оу* axis, find the Union of these projections (open intervals), and find a point *Y* on The *Оу* axis with a minimum coordinate not less than *S*, not covered by merging of segments. To do this, you can sort together the beginning and end of the intervals and go through these points, counting the *balance*. If *Y* is not greater than *R*, then the desired path has a right turn at (0, *Y*) and a left turn at (*xi*, *Y*). Otherwise, there is no path of the first kind, and you need to go to find the path of the second kind.

If *R* ≤ yi, then there is no path of the second kind either. Otherwise, consider the ray coming out of the point (*xi*, *yi*) to the North (see Fig. 2), and find on it the farthest point *S*, which can be easily reached from the point (*xi*, *yi*). Consider the segment [*x*i, min{*R*, *S*}] and find on it the minimum point *Y*, uncovered by merging of the projection of segments, considered above. If such a point *Y* exists, then of all paths leaving point (0, 0) to the North, the shortest path has turns at points (0, *Y*) and (*xi*, *Y*). Otherwise, there are no such paths.

Note that if *Y* = *yi*, then the second turn is not needed.

This algorithm has asymptotic complexity O(*kn* log *n*)

0

0

*xi*

yi

*R*

Fig. 1.

0

0

*xi*

yi

*R*

*S*

Fig. 2.

*S*

*Y*

*Y*