1 The algorithm

1.1 Introduction to the solution — general thoughts

First let us consider, how can we describe the phenomenon of casting shadow. We assume that the sun is infinitely far away, and that it is a point. With such an assumption the time of the day simply determines a direction, and from this direction parallel sun rays are falling from the whole sky. A given point is in the shadow if and only if the half-line from this point "in the direction of the sun" intersects some building or the earth. Notice that if a given point is sunlit, all points above it also are sunlit — this follows from the fact that if a given point belongs to a building, then all point below it belong either to the earth or to the building. In particular this means that if we want to check whether any particular apartment is sunlit, we can consider only the lowest points on the east and west walls and not the whole walls.

The sun in Shanghai is in zenith on 11:57 AM. Before that hour the sunrays fall from "up and east" and after that hour — from "up and west". Notice, that if a given point is sunlit on some hour g before 11:57 AM, then it is also sunlit on any hour between g and 11:57 AM, because the later the hour is, the more steeply the sunrays fall, and so the half-line "in the direction of the sun" lies above the half-line from the hour g, and if any building was to intersect the higher half-line, it would also have to intersect the lower half-line, a contradiction with the assumption our point was sunlit on g. The situation after 11:57 is obviously symmetric. Thus any point not lying inside some building is sunlit in some interval of time containing the hour 11:57 AM.

The lower end of the eastern wall of a given apartment is in the shadow of its building after 11:57 AM, and the lower end of the western wall is in the shadow before 11:57 AM. Thus, to learn when an apartment is sunlit it is enough to check from what time is the lower end of the eastern wall sunlit, and till what time is the western lower end sunlit. These problems are symmetric, so to simplify the reasoning we will concentrate on one, say the first, and solve the second in the same way. Let us consider a given apartment in the building A, and let p denote the lower end of its eastern wall.

On the crucial moment the half-line going from p "in the direction of the sun" intersect the upper-west corner of some building lying to the east of our building, as it is easy to check that from all the points of any building

the upper-west corner is the last to cast a shadow on p. Let (a_n) denote the sequence of the upper-west corners of buildings lying to the west of A. From all points a_n the last to cast a shadow on p is the one for which the angle between the line connecting a_n and p and the horizontal line is the greatest. Hence the following straightforward algorithm:

1.2 The brute-force algorithm — a non-exemplary solution

Author: Jakub Wojtaszczyk

For each complex of apartment building we read from the input and store in arrays the heights of the buildings and the positions of their eastern walls (we put the eastern end of the easternmost building at, say, zero). The positions are not given in the input, but we can compute them online for each building by adding $w + d(i)$ to the position of the previous building. Next, for each apartment we are asked about we check the angle between its lower-east corner p and a_n for all buildings to the east of the apartment, remembering the maximum angle (in practice it is easier to calculate and remember the tangent of the angle instead of the angle itself). Next on the basis of this angle we compute the hour (the angles from 0 to 90 degrees translate linearly to the interval $5:37 - -11:57$), and output that hour. We perform the same operations to the west of the apartment to find the hour at which it stops being sunlit.

1.2.1 Complexity analysis

For each query we loop over all the buildings, so the time complexity of our algorithm is $O(ln)$, where l is the number of queries, and n is the number of buildings. The memory complexity is $O(n)$ for the arrays in which we store the heights and positions of the buildings.

1.2.2 Comments

The author of the problem obviously did not overly ponder the possible ranges of n. We shall later give an exemplary solution, which works in the time $O(n^2 + l \log n)$. It is also possible to construct algorithms working in $O(n+l \log^2 n)$ and $O(n+l(\log l+\log n))$. Unfortunately, with $n \leq 100$ a well-written brute-force algorithm is not slower enough for the test to capture the

difference, especially as with each query we have to perform an output of over twenty characters, which is a very time-consuming operation. If the problem statement bounded n by a thousand and l by 100000, then the exemplary solution working in time $O(n^2 + l \log n)$ would be noticeably quicker than the brute-force. If we had $n \leq 10000$ and $l \leq 10000$, then one would have to use one the two last algorithms. Unfortunately the data is such as it is, thus during the competition the best solution was to quickly type the brute-force algorithm. This algorithm, without any non-standard optimalizations, is implemented as **brute.cpp**

1.3 A slightly more advanced solution

To get to the exemplary algorithm we have to do a bit more work. Let us notice that if some corner a_l lies between (in the east-west sense) a_k and a_m and below the line connecting a_k and $_m$, then it cannot be the last point to cast a shadow on p, for the line connecting p to a_l has to go either below a_m or below a_k. Thus we can throw out all these points from (a_n) which line below some line connecting to other points from the sequence.

Now let a_k and a_{k+1} be two subsequent points in our new sequence (a_n), and suupose that the line connecting p and a_k lies above a_{k+1}. This is enough to know that for $l > k$ the line connecting a_k and p lies above a_l (so none of the points a_l is the last to cast a shadow on p), for otherwise the line connecting a_l to a_k would be higher than the line connecting a_k to p over a_{k+1}, so a_{k+1} could not be in the new sequence. Similarly if a_{k+1} lies above the line connecting p and a_k, then for $l \leq k$ none of the points a_l can be the last to cast a shadow on p.

We can see a binary search beginning to take shape here. The last observation to make is that the "clean" sequence (a_n) does not depend on a particular apartment, but only on the building in which the apartment is. Thus we can compute this sequence for any of the buildings, and then for a given query do a binary search on it.

1.4 Exemplary algorithm 1

Author: Jakub Wojtaszczyk

For any apartment complex we read, as in the brute-force, all the buildings from the input. Next, before we start reading the queries in, we create the "clean sequences". For each building A we consider all the buildings to the

east of A, beginning from the east. For each considered building we add its upper-west corner to the end of the sequence as a_k. Next while a_{k-1} lies below the line connecting a_{k-2} and a_k we throw a_{k-1} out of the sequence, and put a_k on its place (decrementing k by 1). We stop when we reach A, and for each building copy the created clean sequence onto a separate array. Similarly we create the "western clean sequences".

For each query we look at the "eastern clean sequence" for the building containing the queried apartment. In this sequence we perform a binary search to find the largest such k, that the line connecting a_{k-1} with a_k lies above p. We compute the angle between the line connecting p and a_k and the horizontal line and transform it (as in the brute-force algorithm) into hours, minutes and seconds. We do the same thing on the western side.

1.4.1 Correctness

First let us think why our algorithm does in fact find the "clean sequence". For this, we have to understand what the "clean sequence" really represents. Let S be a point infinitely low and below the western corner of the eastern-most building, and T be a point infinitely low and below the western corner of the building just east of A. We claim that the clean sequence contains these and only these upper-west corners of buildings which are on the boundary of the convex hull of the set containing all upper-west corners, S and T. Recall the convex hull is the smallest convex set, in which a given set is contained.

On one hand, if a point a_l lies below a certain line connecting a_k and a_l, then it lies in the interior of the quadrangle S, a_k, a_m, T and thus cannot lie on the boundary of the convex hull. On the other hand, the boundary of the convex hull will consist of two vertical lines going down to S and T, the interval ST and the upper edges a_i, a_j. Thus if some point does not belong to the boundary of the quadrangle, it has in particular to lie below one of the edges, and thus does not belong to the clean sequence.

Now we can prove the correctness of the algorithm of finding the clean sequence. If some element belongs to the clean sequence, it will obviously not get thrown out during the construction. On the other hand we can see by induction that in the kth step we have created the sequence of boundary points of the set $\{S, a_1, \ldots, a_k, T_k\}$, where T_k is the point infinitely far below a_k. When we add a_{k+1} to this sequence, we will find the first point a_l such that a_{l-1} will lie below the line connecting a_l and a_{k+1} (this is what the algorithm does), and then the polygon with vertices $S, a_1, a_2, \ldots, a_l, a_{k+1}, T_{k+1}$ will be

convex, so the induction step can be completed.

On the clean sequence, that is on the boundary of the convex hull, the binary search works because the inclinations of the edges are sorted non-increasingly. We have already proved that the vertices of the convex hull are the only candidates for the point which casts the last shadow on p, so the binary search on the clean sequence will find this point.

1.4.2 Complexity analysis

The memory complexity is of the order of $O(n^2)$ as that is the space needed for the clean sequence arrays for all the buildings. The time complexity of building the clean sequences is $O(n^2)$ — for a given building A when we compute its eastern clean sequence, a given building B is considered at most twice — once when it is added to the sequence, and once when it is removed. Thus a single convex hull is computed in $O(n)$ time, and we build $2n$ such convex hulls. A clean sequence obviously has at most n elements, so a single binary search works in time $O(\log n)$. Thus the whole algorithm has time complexity $O(n^2 + l \log n)$.

1.4.3 Comments

This algorithm was implemented as **subtle.cpp**. As already noted in the comments to the brute-force algorithm, the bound $n \leq 100$ means in practice there is no meaningful difference in the running time of this algorithm when compared to the running time of the brute-force algorithm. Also, when such a bound is given, it is not necessary to try to reduce the $O(n^2)$ building time for the convex hulls. To satisfy the readers curiosity we comment on two other solutions.

First note that the convex hull for the building k is constructed inductively by modifying the convex hull for the building $k-1$. If we sort queries by the number of building at the moment when we have constructed the convex hull for building $k-1$ the could answer all queries about "eastern sides" for this building and next forget about this convex hull and constructing the convex hull for next buildings inductively. We proceed similarly on the werstern side, and then sort the answers by the question number and output the results. The time complexity of such a solution is $O(n + l \log l + l \log n)$ (the first part corresponds to the building of the convex hull, the second to sorting the queries and answers and the third for the binary search for each

query), which is equal to $O(n + l \log l)$. The downside of this solution is its memory complexity, which is of the order of $O(l + n)$.

The second solution is a little bit more tricky. It's the second exemplary solution.

1.5 Exemplary solution 2

Authors: Wojciech Czerwiski, Jakub Wojtaszczyk

1.5.1 Algorythm

For each and every building we remember the range of the levels, for which the shadow is cast by the same building. For the building k we can get those ranges by analyzing the convex hull of this building. We check where the line, which connects the right top corners of the adjacent buildings at the hull of the building k crosses this building. Over the crossed point the shadow is cast by the left building from this pair, or some building more to the left. In this way when we have the ranges of the levels and the concrete query we will be able to search binary for the given building. Lets see, that if having the convex hull of the building k we consider pairs of buildings creating lines from the right side (these closer to the building k), this lines connecting the corners of buildings in pair will cross the building k higher and higher. Hence at the moment, when the line goes over the building k we may stop checking the pairs to the left because for those pairs the lines will be even higher and it's not relevant for the question about the building k.

1.5.2 Complexity analyzis

Lets see, that following that strategy we will build the structure of ranges on the buildings at the time $0(n)$. Those buildings, that will generate lines crossing the building will be deleted from the next hull (for building $k + 1$) because they will be hidden behind the building k. So the number of operations of putting into the convex hull will be equal to the number of operations of checking the line plus/minus the constant number of operations for each building, which means plus/minus $O(n)$ operations. We should remember however that we can build the convex hull in $O(n)$ time, so the entire process of building the structure will take us $O(n)$. Answering queries

will take $O(l \log(n))$, so the entire algorythm will be working in time $O(n + l \log(n))$, which is the best result until now.

1.5.3 Implementation

Given algorythm was implemented in C++ in the file **sunlight.cpp** and in Java in the file **LotsOfSunlight.java**.

2 Tests

Test 0 is the sample input.
Tests 1 and 2 are the small random tests, which check elementary correctness of program without special cases.
Test 3 includes cases of buildings increasly higher or increasly lower and the case of identical buildings. In this test there is also a case, which checks correctness of the answer to the question about non existing building.
Test 4 checks some special cases, buildings getting smaller with the same difference of height, buildings getting smaller with increasing differences and decreasing differences of height. Besides in this set of tests there is also another test for the non existing level (in the building whose number is one bigger than the biggest number of building).
Test 5 is the single set checking only, whether the big numbers where considered, whether the height of building counted in levels multiplied by the height of the leves does not exceed the range.
Tests 6 and 7 are the medium size tests.
Tests 8 and 9 are the big performance tests.