Problems summary

Recap: 274 teams, 12 problems, 5 hours. This analysis assumes knowledge of the problem statements (published separately on http://neerc.ifmo.ru/ web site).

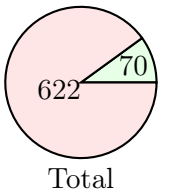
Summary table lists problem name and stats:

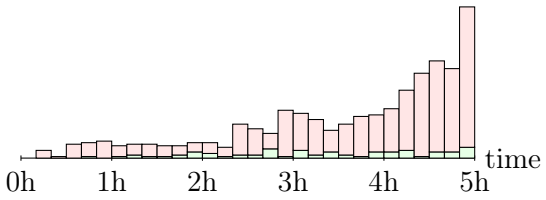
- **author** — author of the original idea
- **developer** — developer of the problem statement and tests
- **acc** — the number of teams that had solved the problem (gray bar denotes a fraction of the teams that solved the problem)
- **runs** — the number of total attempts
- **succ** — overall successful attempts rate (percent of accepted submissions to total, also shown as a bar)

problem name	author	developer	acc/runs	succ
Archery Tournament	Maxim Akhmedov	Artem Vasilyev	70 /692	10%
Box	Georgiy Korneev	Niyaz Nigmatullin	234 /701	33%
Connections	Pavel Irzhavski	Pavel Irzhavski	127 /892	14%
Designing the Toy	Maxim Akhmedov	Maxim Akhmedov	91 /437	20%
Easy Quest	Pavel Mavrin	Pavel Mavrin	235 /796	29%
The Final Level	Georgiy Korneev	Pavel Kunyavsky	25 /138	18%
The Great Wall	Vitaliy Aksenov	Vitaliy Aksenov	3 /29	10%
Hack	Petr Mitrichev	Petr Mitrichev	0 /1	0%
Interactive Sort	Borys Minaiev	Borys Minaiev	20 /163	12%
Journey from Petersburg to Moscow	Gleb Evstropov	Gleb Evstropov	3 /34	8%
Knapsack Cryptosystem	Mikhail Dvorkin	Mikhail Dvorkin	3 /69	4%
Laminar Family	Maxim Akhmedov	Maxim Akhmedov	23 /152	15%

Problem A. Archery Tournament

Author: Maxim Akhmedov
Statement and tests: Artem Vasilyev



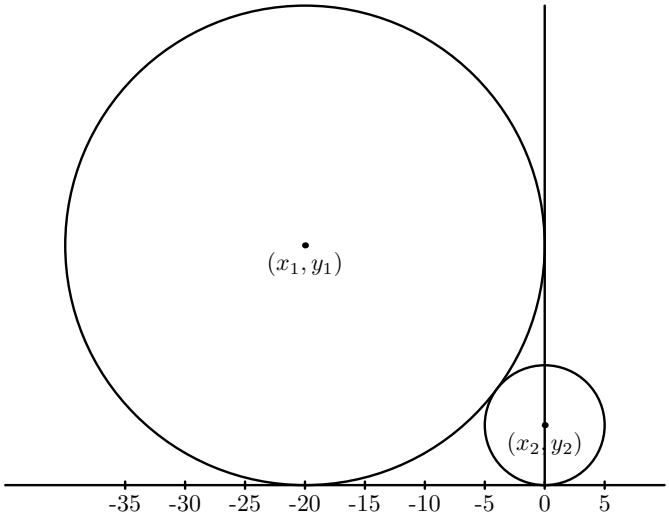


	Java	Kotlin	C++	Python	Total
Accepted	1	0	69	0	70
Rejected	50	12	531	29	622
Total	51	12	600	29	692

solution	team	att	time	size	lang
Fastest	Kazakh-British TU 1	1	41	2,606	C++
Shortest	Tartu U 1	3	115	1,116	C++
Max atts.	Belarusian SU 2	11	199	2,018	C++

The main observation for this problem is that you do not have to check many circles. More specifically, there are $\mathcal{O}(\log C)$ ($C = 10^9$) circles intersecting any vertical line.

For simplicity, let's only consider circles which center is to the left of the vertical line $x = 0$. Consider this «critical» position of two circles. In this picture, $x_1 = -y_1$ and $x_2 = 0$. Then, formula describing the touching of these two circles $(x_1 - x_2)^2 + (y_1 - y_2)^2 = (y_1 + y_2)^2$ becomes $y_1^2 + (y_1 - y_2)^2 = (y_1 + y_2)^2$ which simplifies to $y_1^2 = 4y_1y_2$ and $y_2 = \frac{1}{4}y_1$. So, the number of circles to the left of $x = 0$ does not exceed $\log_4 C$, the same is true for circles on the right.



In this picture, $y_1 = 20, y_2 = 5, y_2 = \frac{1}{4}y_1$

Now the remaining part of the solution is to build an efficient data structure that allows to extract all candidate circles crossing the given vertical line and check all of them. It is equivalent to maintaining the set of segments and retrieving all segments containing the given point. This can be done with a simple segment tree.

Let's compress all x values and build a segment tree T with a `max` operation. For segment $[l, r]$ set the value of $T[l]$ to r (assuming all x-coordinates are different). To answer the «2 x y» query, we need to check all such $l \le x$ that $T[l] \ge x$. It is possible to do with a single tree descent. Suppose we are currently at a vertex corresponding to segment $[L, R]$:

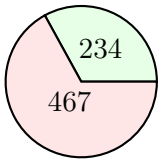
- If $L > x$ or $max[L..R] < x$, then exit.
- If the current vertex is a leaf, report L as the possible candidate.
- Otherwise, go to the left and right subtrees recursively.

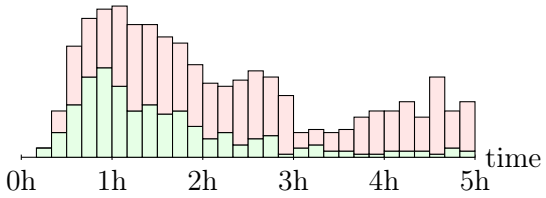
Since there are only $\mathcal{O}(\log C)$ circles covering any vertical line, this procedure works in $\mathcal{O}(\log n \log C)$ time. When a new segment $[l, r]$ appears, set $T[l] = r$. To delete a segment, set $T[l] = -\infty$. Both of these operations can be done in $\mathcal{O}(\log n)$ time.

The total time complexity of this solution is $\mathcal{O}(n \log n \log C)$.

Problem B. Box

Author:	Georgiy Korneev
Statement and tests:	Niyaz Nigmatullin

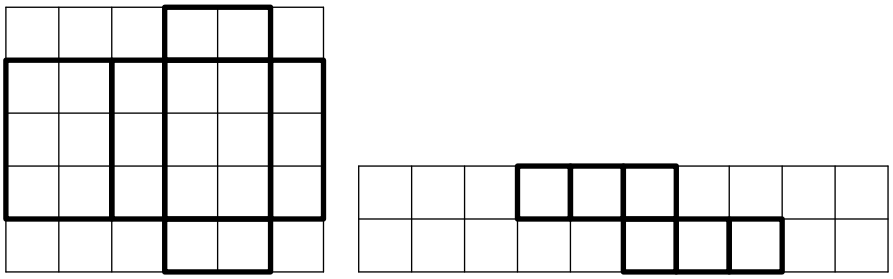




	Java	Kotlin	C++	Python	Total
□ Accepted	11	1	211	11	234
□ Rejected	12	5	432	18	467
Total	23	6	643	29	701

solution	team	att	time	size	lang
Fastest	Vilnius U 1	1	16	1,046	C++
Shortest	Omsk STU	2	59	424	Python
Max atts.	Vilnius U 2	20	245	5,547	C++

One can see that only two of the given 11 types of cube nets are required to be checked. For parallelepipeds the same is true: one can check all the possible orderings of (a, b, c) and (w, h) and check if the same two nets fit into rectangle.



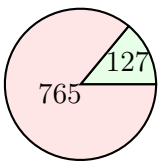
The formulae for these two cases are:

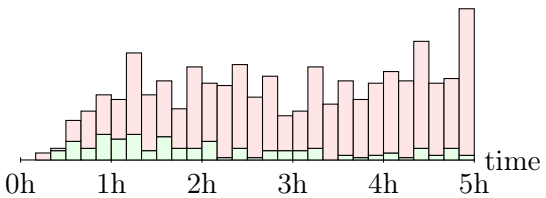
1. $2a + 2b \le h$ and $b + 2c \le w$
2. $a + c \le h$ and $3b + a + c \le w$

These two cases' pictures were given in the problem statement, and both cases were covered in sample testcases. It's possible to generate all the nets in your program to prove the fact, but judges were expecting teams to try most of the cases on paper ending up with two interesting ones.

Problem C. Connections

Author:	Pavel Irzhavski
Statement and tests:	Pavel Irzhavski





	Java	Kotlin	C++	Python	Total
□ Accepted	3	0	124	0	127
□ Rejected	34	1	726	4	765
Total	37	1	850	4	892

solution	team	att	time	size	lang
Fastest	Moscow IPT 1	2	20	1,725	C++
Shortest	American U in CA 1	1	163	1,080	C++
Max atts.	Grodno SU	23	264	2,443	C++

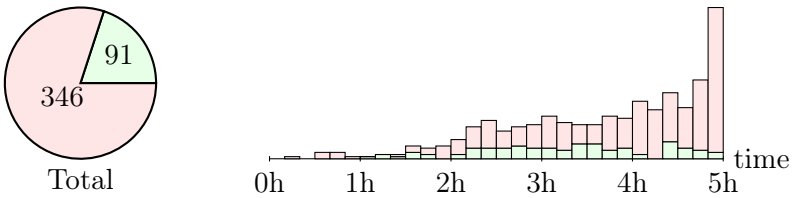
Let's consider the cities as vertices and the roads as arcs in a directed graph G. Now we choose an arbitrary city v.

Then we build a spanning arborescence (directed rooted tree) with root v (using BFS or DFS etc.). Further we change the direction of each arc and build another arborescence with root v. The first arborescence exists because each city is reachable from city v and the second one does because city v is reachable from any other city.

Now it is easy to see that restoring the direction of the arcs of the second arborescence and uniting it with the first arborescence we get a subgraph H of graph G with at most $2(n-1)$ arcs. Also in subgraph H it is possible both to go from city v to any other city and to get to city v from any other city. Therefore it is possible to go from city u to city w for any cities u and w, since it is possible to go from city u to city v and from city v to city w.

Problem D. Designing the Toy

Author: Maxim Akhmedov
Statement and tests: Maxim Akhmedov



	Java	Kotlin	C++	Python	Total
□ Accepted	2	0	87	2	91
□ Rejected	6	0	328	12	346
Total	8	0	415	14	437

solution	team	att	time	size	lang
Fastest	SPb SU 1	3	66	3,578	C++
Shortest	Tyumen SU 1	1	168	1,010	Python
Max atts.	SPb ITMO U 4	25	275	1,567	C++

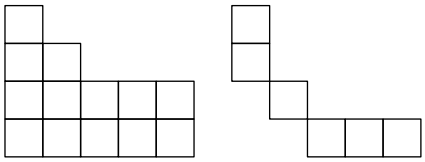
Let's start with a necessary condition for the desired figure to exist. Let's prove that if $a > bc$ holds (or any of the two symmetric inequalities), then it is impossible to construct such a figure.

For the sake of simplicity, denote the orthogonal projection areas onto the corresponding coordinate planes as S_{xy}, S_{xz} and S_{yz}, and the orthogonal projection lengths onto the corresponding coordinate axes as l_x, l_y and l_z.

Denote our figure as F. Note the following relation:

$$\text{proj}_{Ox}(F) = \text{proj}_{Ox}(\text{proj}_{Oxz}(F))$$

This equality immediately implies that $l_x \le S_{xz}$ and, symmetrically, $l_y \le S_{yz}$.



In the first case $S_{xz} = 5$, $S_{yz} = 4$, $S_{xy} = 13$.
In the second case $S_{xz} = 5$, $S_{yz} = 4$, $S_{xy} = 6$

On the other hand, the orthogonal projection $\text{proj}_{Oxy}(F)$ consists of the unit squares such that their x-coordinates belong to the set of size l_x and their y-coordinates belong to the set of size l_y. Hence, $S_{xy} \le l_x l_y$.

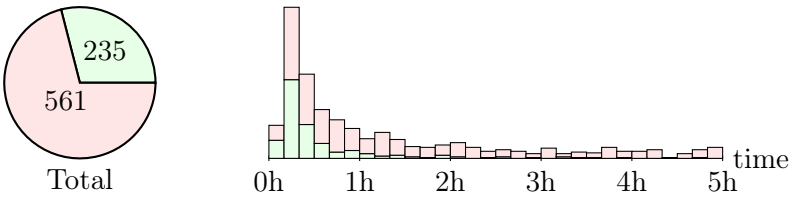
Combining these inequalities, we immediately get that $S_{xy} \le S_{xz} S_{yz}$.

It turns out that the system of three inequalities that are obtained from $S_{xy} \le S_{xz} S_{yz}$ by renaming the coordinate axes is not only the necessary condition, but also a criterion. Suppose that S_{xy} is the maximum of three orthogonal projection areas. We will now build a desired figure under the assumption of $S_{xy} \le S_{xz} S_{yz}$. The figure will be effectively two-dimensional, i.e. all the voxels will have z-coordinate equal to 0.

There are two essential cases: either S_{xy} is not smaller than $S_{xz} + S_{yz} - 1$ or not. In the former case we will draw an L-shaped corner ensuring that S_{xz} and S_{yz} are taking the required values, and then fill the interior of the corner with the remaining number of $S_{xy} - (S_{xz} + S_{yz} - 1)$ voxels. In the latter case we will "cut" the corner of the L-shaped figure, reducing the total number of voxels in the figure while keeping S_{xz} and S_{yz}. Refer to the pictures below for the details.

Problem E. Easy Quest

Author:	Pavel Mavrin
Statement and tests:	Pavel Mavrin



	Java	Kotlin	C++	Python	Total
Accepted	12	2	210	11	235
Rejected	63	13	437	48	561
Total	75	15	647	59	796

solution	team	att	time	size	lang
Fastest	SPb ITMO U 1	1	5	1,431	C++
Shortest	Altai STU 2	4	43	461	Python
Max atts.	ADA U 2	11	231	1,157	Java

Let's simulate the quest, remembering the current state of our inventory. For each creature:

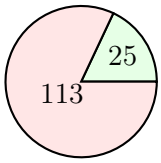
- If the creature gives us an item, add this item into inventory.
- If we meet a unicorn, add special «joker» item into inventory. Later, we will decide, what item it should become.

- For an evil creature, check the inventory. If it contains required item, use it, otherwise use one of the joker items. If we have neither required item nor joker items, then it's impossible to defeat this enemy, so output «No».

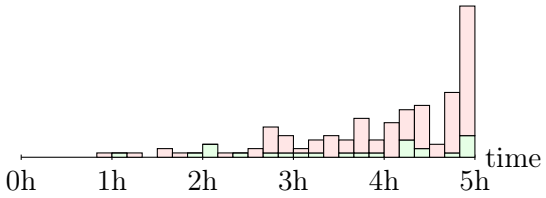
In the end, if there are still some joker items in the inventory, assign any valid item type (for example, 1) to them.

Problem F. The Final Level

Author: Georgiy Korneev
Statement and tests: Pavel Kunyavsky



Total

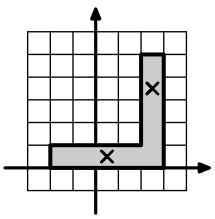


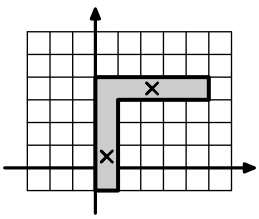
	Java	Kotlin	C++	Python	Total
□ Accepted	0	0	25	0	25
□ Rejected	0	0	113	0	113
Total	0	0	138	0	138

solution	team	att	time	size	lang
Fastest	Vilnius U 1	2	69	2,840	C++
Shortest	Belarusian SUIR 2	4	256	1,159	C++
Max atts.	SPb SU 1	7	280	12,064	C++

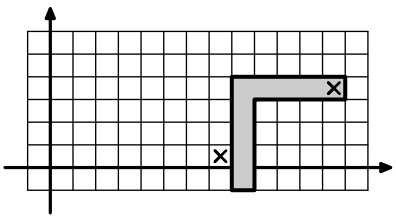
First, reflect the plane in such a way, that $a, b \geq 0$. Now place corners one by one, reducing the problem to the same problem for smaller values of a and b. We will maintain the following invariant: either all cells (x, y) such that $x \leq a$ are empty, or all cells (x, y) such that $y \leq b$ are empty.

Case 1. If $a < n$ and $b < n$, then one corner is enough:


if cells $x \leq a$ are empty, or

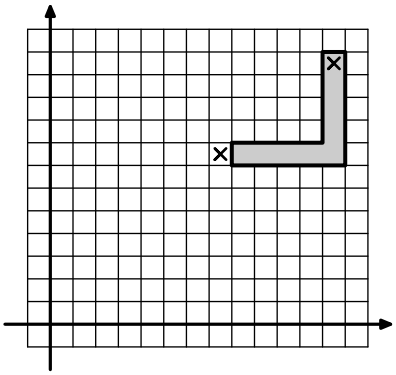

if cells $y \leq b$ are empty.

Case 2. If $a \geq n$ and $b < n$, then reduce problem to $(a - n, 0)$:



Case 3. If $a < n$ and $b \geq n$, then reduce problem to $(0, b - n)$ in the same way.

Case 4. If $a \geq n$, $b \geq n$, and $a \geq b$, then reduce problem to $(a - n, b - n + 1)$:

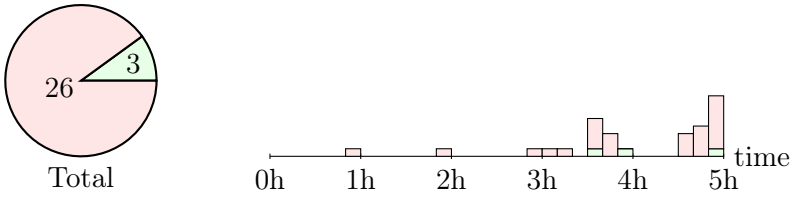


Case 5. If $a \geq n$, $b \geq n$, and $a < b$, then reduce problem to $(a - n + 1, b - n)$ in the same way.

It's easy to show that the lower bound for the number of corners is $max(\lfloor \frac{a}{n} \rfloor + 1, \lfloor \frac{b}{n} \rfloor + 1, \lfloor \frac{a+b}{2n-1} \rfloor + 1)$, and all reductions maintain this lower bound.

Problem G. The Great Wall

Author:	Vitaly Aksenov
Statement and tests:	Vitaly Aksenov



	Java	Kotlin	C++	Python	Total
□ Accepted	0	0	3	0	3
□ Rejected	2	0	24	0	26
Total	2	0	27	0	29

solution	team	att	time	size	lang
Fastest	Moscow IPT 1	2	216	4,202	C++
Shortest	Belarusian SU 4	5	237	3,433	C++
Max atts.	Belarusian SU 4	5	237	3,433	C++

As a first step, we replace b_i and c_i by $b_i - a_i$ and $c_i - a_i$ and set all a_i to zero. Suppose, for a moment, that we have a function of s that returns the number of walls with strength less than s. Then, using the binary search and this function we can find the strength of the k-th wall.

Now, we describe how to build such a function of s. For that we need a data structure D that supports three operations in logarithmic time: insert, remove and how many elements are less than given k. This could be, for example, cartesian trees or, simply, sorted_set in C++.

At first, we explain how to calculate the number of walls with strength less than s and for which two segments do not intersect, that is, none of c_i are used. We move the right segment to the right and maintain a data structure D that contains all the segments on the left that do not intersect the current right segment. We calculate the prefix sums $pb_i = b_1 + \ldots + b_i$. Suppose that the right segment is $[y, y + r - 1]$. We simply make a query to D: the number of elements less than $s - (b_y + \ldots + b_{y+r-1}) = s - (pb_{y+r-1} - pb_{y-1})$. Then, we move the segment to the right and add a new segment $[y - r + 1, y]$ to D (insert $pb_y - pb_{y-r}$). Summing up the answers by D on the queries we get the desired number of walls.

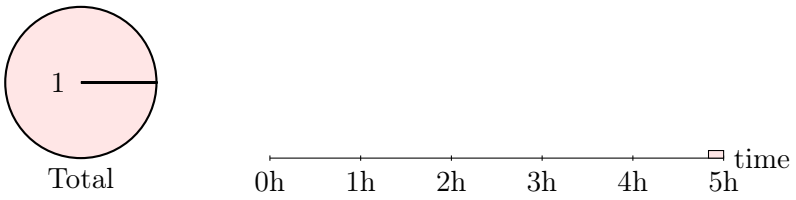
Secondly, we explain how to calculate the number of walls with strength less than s and for which two segments intersect. Now, we make a trick: we represent the strength of the wall with two intersecting segments $[x, x+r-1]$ and $[y, y+r-1]$ as $g_x + f_y$. We argue that $g_x = \sum_{i=1}^{x-1}(c_i - 2b_i) + \sum_{i=x}^{x+r-1}(c_i - b_i)$ and $f_y = \sum_{i=1}^{y-1}(2b_i - c_i) + \sum_{i=y}^{y+r-1} b_i$ satisfy us. All f_y and g_x could be calculated in linear time.

$$
\begin{array}{lllllllll}
f_y = & b_{y+r-1} + \ldots + b_{x+r} + b_{x+r-1} & + \ldots & + b_y & + (2b_{y-1} - c_{y-1}) & + \ldots & + (2b_x - c_x) & + (2b_{x-1} - c_{x-1}) + \ldots & + (2b_1 - c_1) \\
g_x = & (c_{x+r-1} - b_{x+r-1}) & + \ldots & + (c_y - b_y) & + (c_{y-1} - by - 1) & + \ldots & + (c_x - b_x) & + (c_{x-1} - 2b_{x-1}) + \ldots & + (c_1 - 2b_1) \\
 & b_{y+r-1} + \ldots + b_{x+r} + c_{x+r-1} & + \ldots & + c_y & + b_{y-1} & + \ldots & + b_x & &
\end{array}
$$

We move the right segment to the right and maintain a data structure D that contains g_x for all intersecting segments on the left. Suppose that the right segment is $[y, y+r-1]$. We simply make a query to D: the number of elements less than $s - f_y$. Then, we move the segment to the right, add a new segment $[y, y+r-1]$ to D (insert g_y) and remove a segment $[y-r+1, y]$ from D (remove g_{y-r+1}, $[y-r+1, y]$ does not intersect with $[y+1, y+r]$). Summing up the answers by D on the queries we get the desired number of walls.

Problem H. Hack

Author: Petr Mitrichev
Statement and tests: Petr Mitrichev



	Java	Kotlin	C++	Python	Total
Accepted	0	0	0	0	0
Rejected	0	0	1	0	1
Total	0	0	1	0	1

Let us start with sending 30 000 random queries a_i and getting the computation time for each of them back. We can then subtract the time to repeatedly square the query 60 times from each computation time, as the repeated squaring (line 7 in the program listing in the problem statement) does not depend on d. The remainder would be the total time of multiplications in line 5 of the program.

Then we will find the number d bit by bit, from the least significant to the most significant. First, the 0-th (parity) bit is always equal to 1, since the number d is coprime with the even number m.

Now consider the 1-st bit. If it is equal to 1, then line 5 will execute for it, and we will multiply a_i by a_i^2 (here and below, we will omit (mod n) part — it is implied for all powers of a_i). If it is equal to 0, then we will never multiply a_i by a_i^2. Now, we know the time of multiplication of a_i by a_i^2, and total time of all multiplications in line 5 — and we need to determine if the former is a part of the latter.

Here comes the key idea: for numbers a_i such that a_i and/or a_i^2 have less bits than usual, multiplying a_i by a_i^2 takes less time than usual. For such values a_i the total time is also likely to be a bit less than usual if it includes multiplying a_i by a_i^2, as one of the summands is smaller than usual.

Of course, since it's just one of the summands, *is likely* is the strongest thing we can say. For each particular a_i, the overall time might still end up being higher than usual just because of randomness. However, if

we look at all 30 000 queries together, statistical methods can tell us the answer with extremely high probability.

More precisely, let us look at *sample covariance* of the time p_i it takes to multiply a_i by a_i^2, and the total time q_i of all multiplications in line 5 for this a_i.

$$cov(p_i, q_i) = \frac{\sum p_i \cdot q_i}{k} - \frac{\sum p_i}{k} \cdot \frac{\sum q_i}{k}$$

(in our case $k = 30\,000$)

In case the 1-st bit is 1, $q_i = p_i + r_{i,1} + r_{i,2} + ...$, where $r_{i,j}$ are the times of other multiplications in line 5. Note that the arguments to other multiplications will be some other powers of a_i, and *not* a_i or a_i^2, so we can expect $cov(p_i, r_{i,j})$ to be around zero. Since covariance is linear in each argument, $cov(p_i, q_i)$ should be roughly equal to $cov(p_i, p_i)$ in this case.

In case the 1-st bit is 0, $q_i = t_i + r_{i,1} + r_{i,2} + ...$, where t_i is the time of the next multiplication in line 5, and $r_{i,j}$ correspond to the following multiplications. As above, we can expect $cov(p_i, r_{i,j})$ to be around zero, but not so for $cov(p_i, t_i)$ since t_i is the time of multiplying a_i by some other power a_i^k, so p_i and t_i share one of the arguments (a_i), and thus the covariance should also be positive. Altogther we can expect $cov(p_i, q_i)$ to be roughly equal to $cov(p_i, t'_i)$, where t'_i is the time of mulplication of a_i by a random number between 0 and $n - 1$. In one of the reference solutions, we used the next square as an approximation for such random number (in this case a_i^4).

We're going to do the following to determine the 1-st bit: compute $cov(p_i, q_i)$, $cov(p_i, p_i)$ and $cov(p_i, t'_i)$. If the first number is closer to the second than to the third one, then the 1-st bit is 1, otherwise it is 0.

Having determined the 1-st bit, we can continue with the 2-nd, since now we know both arguments to the possible multiplication corresponding to it: it's either a or a^3 being multiplied by a^4. We can apply the same covariance computation to determine the 2-nd bit now, and so on until we know the entire d.

That completes the solution to the problem, but one might wonder: how certain are we that it works? What is the exact meaning behind *extremely high probability*, *roughly* and *is likely* in the above text?

We do not claim a completely formal argument for that, but we do have one modulo a reasonable assumption. More precisely, we will assume that when u and s are different numbers between 1 and $m - 1$, the numbers $bits(a_i^u)$ and $bits(a_i^s)$ to be independent random variables that are distributed in the same way as just $bits(x)$, where x is a random number between 0 and $n - 1$. This assumption does not hold precisely, but it is a very reasonable approximation.

Now let's denote as $\xi_{i,j,l}$ a family of independent random variables, each distributed as $bits(x) - \mathbb{E}(bits(x))$ — in other words as $bits(x)$, but shifted to have an expectation of 0. We denote the index of the query as i, the index of the multiplication within the query as j, and enumerate different numbers used in one multiplication using l. In case the corresponding bit of d is 1 (the case of this bit being 0 is handled similarly), the numbers we're computing can be approximately modeled as:

$$cov(p_i, q_i) = \frac{\sum_i \sum_j \xi_{i,1,1} \cdot \xi_{i,1,2} \cdot \xi_{i,j,1} \cdot \xi_{i,j,2}}{k}$$

$$cov(p_i, p_i) = \frac{\sum_i \xi_{i,1,1}^2 \cdot \xi_{i,1,2}^2}{k}$$

$$cov(p_i, t'_i) = \frac{\sum_i \xi_{i,1,1}^2 \cdot \xi_{i,1,2} \cdot \xi_{i,1,3}}{k}$$

(where j iterates over 1 bits of d, in other words the upper bound on j is 60)

And then compute the differences between the first one and the other two:

$$cov(p_i, p_i) - cov(p_i, q_i) = -\frac{\sum_i \sum_{j>=2} \xi_{i,1,1} \cdot \xi_{i,1,2} \cdot \xi_{i,j,1} \cdot \xi_{i,j,2}}{k}$$

$$cov(p_i, q_i) - cov(p_i, t'_i) = \frac{\sum_i (\xi_{i,1,1} \cdot \xi_{i,1,2} \cdot (-\xi_{i,1,1} \cdot \xi_{i,1,3} + \sum_j \xi_{i,j,1} \cdot \xi_{i,j,2}))}{k}$$

And our solution will make a correct decision if the first difference is smaller than the second.

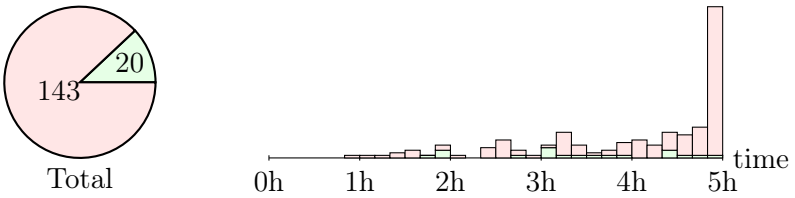
Both those differences, and the difference between them, are expressions of the form "mean of k copies of independent equally distributed random variables", and the Central Limit Theorem tells us that such expressions are almost normally distributed for large values of k. So in order to determine the probability of the first number being greater than the second number we just need to find the mean and variance of their difference, and use the normal distribution quantiles. Moreover, we can instead find the mean and variance of the difference for $k = 1$, and then just divide the variance by k (or the standard deviation by $\sqrt{k}$).

Those mean and variance are the easiest to estimate empirically. Doing that shows that depending on n in the worst case of $j \leq 60$ the mean is between -4 and -6, the standard deviation is between 60 and 80, and the ratio of mean to standard deviation is at least 0.05. After adjusting for the averaging of k experiments, the ratio increases $\sqrt{k}$ times to around 8. So our algorithm making a wrong decision would amount to a normal distribution sampling above 8 standard deviations, which happens with probability around 10^{-15}. Even accounting for 30 tests and 60 bits per test, the overall probability of error is still within $2 \cdot 10^{-12}$, which is a virtual impossibility.

The reasoning above relied on a few assumptions and approximations, but they were not so radical, so the probability of error should be around the number we computed. In more concrete terms, we've ran this solution on 10000 testcases, and the correct difference was always at least 2.5x smaller than the incorrect one.

Problem I. Interactive Sort

Author: Borys Minaiev
Statement and tests: Borys Minaiev



	Java	Kotlin	C++	Python	Total
Accepted	1	0	19	0	20
Rejected	1	0	142	0	143
Total	2	0	161	0	163

solution	team	att	time	size	lang
Fastest	Belarusian SU 5	3	103	3,336	C++
Shortest	Kazakh-British TU 1	2	118	1,631	C++
Max atts.	Ulyanovsk STU	7	163	2,200	C++

Let's iterate over all even numbers in a random order and split odd ones using them. First, we choose some even number $2x$ and compare it with all odd numbers. This splits odd numbers into two groups —

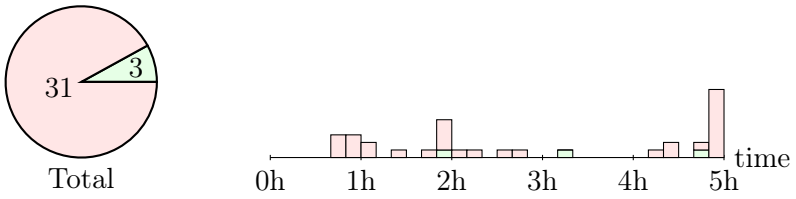
one with all numbers smaller than $2x$ and group with numbers larger than $2x$. This also lets us know the exact value of current even number (because we know the number of odd numbers less than it).

After iterating over k even numbers all odd numbers will be split into $k + 1$ groups. We can split odd numbers by $k + 1$-th even number with less than n comparisons. We can do a binary search on groups in $O(\log k)$ operations comparing $2x$ with arbitrary element from that group. In the end, we are left with two groups. Split all numbers from those two groups into three groups and continue the process.

It could be proven that total number of operations is $O(n \log n)$.

Problem J. Journey from Petersburg to Moscow

Author:	Gleb Evstropov
Statement and tests:	Gleb Evstropov



	Java	Kotlin	C++	Python	Total
Accepted	0	0	3	0	3
Rejected	0	0	31	0	31
Total	0	0	34	0	34

solution	team	att	time	size	lang
Fastest	Moscow SU 1	3	115	1,866	C++
Shortest	Moscow SU 1	3	115	1,866	C++
Max atts.	Moscow SU 1	3	115	1,866	C++

First we are going to consider the case when optimal answer consists of less than k edges. We claim that this means the answer is the shortest path from vertex 1 to vertex n. Indeed, for any path of no more than k edges its k-sum weight is equal to its weight in conventional definition, while for any path consisting of more than k edges its k-sum weight is strictly smaller than conventional one.

Now we consider only paths of at least k edges. For the purpose of simplicity we suppose no two edges have equal weights. Equal weights can be resolved by comparing such edges by their indices. Consider the optimal path in k-sum metric. Let x be the k-th maximum weight along this path. Now we use this value of x as a cutoff: find connected components of vertices that can be reached from each other by edges of weights less than x, then consider only edges greater than or equal to x. In this graph find the shortest path from component containing vertex 1 to component containing vertex n that has length exactly k. This can be done in $O(mk)$ time for a fixed cutoff x.

The above procedure will find the optimal answer, if the value of x is set to the weight of k-th maximum edge in it. To find the proper value of x we can simply try all possible weights. The complexity of such solution is $O(m^2k)$.

The general idea of a model solution is to tune weights in such a way to make the path that is optimal in k-sum terms optimal in conventional distance. Define G_x as the graph that has the same set of vertices and edges but the weight $w(e)$ of edge $e \in E(G)$ is replaced with value $w_x(e) = \max(0, w(e) - x)$, i.e. edges that are below cutoff x are replaced with zero-weight edges, while edges greater than x have their weight decreased by x. Let $d(x)$ be the shortest path from 1 to n in graph G_x. Consider $f(x) = d(x) + x \cdot k$. We claim that shortest path in k-sum distance $ksumSP(G) = \min\limits_{x \geq 0} f(x)$.

First we claim that $ksumSP(G) \geq f(x)$ for any $x \geq 0$. Consder any path p, let $c_1 \geq c_2 \geq c_3 \geq \ldots \geq c_l$ be individual weights of its edges. If $l \leq k$ the claim is obviously true for any $x \geq 0$. Consider the function $d_p(x)$ — length of this particular path depending on the value of x increasing from 0 to infinity ($f_p(x) = d_p(x) + x \cdot k$):

- Let $x < c_l$. Inrecemnting x by 1 will inrease $f_p(x)$ by $l - k$.
- While $c_{i+1} \leq x < c_i$, where $i \geq k + 1$ incrementing x by 1 change $f_p(x)$ by $i - x > 0$.
- For $c_{k+1} \leq x < c_k$, $f_p(x + 1) - f_p(x) = 0$. Moreover,

$$f_p(x) = \sum_{i=1}^{l} \max(0, c_i - x) + x \cdot k = \sum_{i=1}^{k} (c_i - x) + x \cdot k = \sum_{i=1}^{k} c_i$$

  , that is the length of this path in k-sum terms.
- For $c_{i+1} \leq x < c_i$, where $i < k$ the difference $f_p(x + 1) - f_p(x)$ is equal to $k - i$.

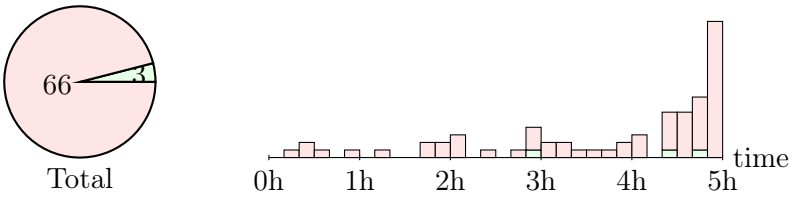
In other words, for any fixed path its value of function $f_p(x)$ goes down, reaches minimum value equal to the sum of k maximum edges at c_{k+1}, stays at this level for $x \in [c_{k+1}; c_k]$ and then starts to increase. Thus as $f(x)$ is the minimum of $f_p(x)$ among all individual paths it never exceeds the legnth of shortest path in k-sum metric.

The last observation to make is that minimum of $f(x)$ is indeed equal to $ksumSP(G)$. Consder any path p, let $c_1 \geq c_2 \geq c_3 \geq \ldots \geq c_l$ be individual weights of its edges. If $l \leq k$ then for $x = 0$, $f_p(0) = ksumSP(G)$. Otherwise, consider $x = c_k$. The behaviour of $f_p(x)$ provided above means that $f_p(c_k)$ is equal to k-sum length of this particular path, thus is equal to the shortest k-sum path in G.

The resulting solution is: try all possible $w(e)$ as values of x, build G_x and compute $d(x)$ using Dijkstras algorithm implementation with binary heap. The overall time complexity is $O(m^2 \log n)$.

Problem K. Knapsack Cryptosystem

Author:	Mikhail Dvorkin
Statement and tests:	Mikhail Dvorkin



	Java	Kotlin	C++	Python	Total
Accepted	0	0	3	0	3
Rejected	1	0	48	17	66
Total	1	0	51	17	69

solution	team	att	time	size	lang
Fastest	SPb SU 1	10	171	12,455	C++
Shortest	Moscow SU 1	15	264	2,925	C++
Max atts.	Moscow SU 1	15	264	2,925	C++

If $n \le \frac{2}{3}\log_2 q$, we solve the knapsack problem instance with an exponential algorithm. We divide the n given numbers into two sets of size $\frac{n}{2}$, and for each set we calculate a sorted array of sums of all its subsets. Then these two arrays of size $O(2^{n/2})$ can be traversed with a linear search: one pointer moves by one position in one array, while another pointer moves in the other array searching for the desired sum s. The time is $O(2^{n/2}) = O(2^{\frac{1}{3}\log_2 q}) = O(\sqrt[3]{q})$.

If $n \ge \frac{2}{3}\log_2 q$, there is an algorithm that uses the nature of the sequence $\{b_i\}$ to break the cryptosystem. (Note that the previous paragraph works for any sequence $\{b_i\}$).

Consider the original Alice's sequence $\{a_i\}$. Note that $a_2 > a_1$; $a_3 > a_1 + a_2 > 2a_1$; $a_4 > a_1 + a_2 + a_3 > 4a_1$; and so on, and finally $a_n > 2^{n-2}a_1$. But q is greater than a_n (since it's greater than the sum of all a_i). Therefore, a_1 is less that $q/2^{n-2}$, call this number t.

Since Alice's value r is odd, b_1 has the same number of trailing zeroes in binary system as a_1, call that number z; we know z from the input data. We iterate over all possible values for a_1 in the range $[1, t]$ which have exactly z trailing zeroes. For each candidate, we suppose that multiplication of this candidate by r resulted in b_1. This gives us the knowledge about all bits of r except for the upper z bits. We examine all possible masks in the z upper bits.

This process iterates over all possilbe values of r. For each possible value of r, we calculate $\{a_i\}$ from $\{b_i\}$, and check whether this $\{a_i\}$ is a superincreasing sequence, in which case we use greedy algorithm from the largest value to the lowest in order to decompose the given sum s.

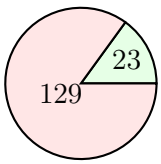
We iterate over $O(t/2^z)$ possible candidates, for each of which we iterate over 2^z possible masks in the upper bits. Therefore, the time is $O(t) = O(q/2^{n-2}) = O(2^{\log_2 q - \frac{2}{3}\log_2 q}) = O(\sqrt[3]{q})$.

(Formally, for each possible value of r, we need $O(n)$ time to calculate and process the sequence a_i, but, in reality, we will calculate $O(1)$ first elements on average before we learn that this sequence is not superincreasing.)

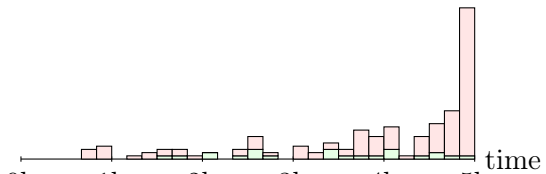
The asymptotic time of this algorithm is $O(2^{min(n/2, \log_2 q - n)}) = O(\sqrt[3]{q})$.

Problem L. Laminar Family

Author:	Maxim Akhmedov
Statement and tests:	Maxim Akhmedov



Total



	Java	Kotlin	C++	Python	Total
□ Accepted	0	0	23	0	23
□ Rejected	3	0	126	0	129
Total	3	0	149	0	152

solution	team	att	time	size	lang
Fastest	Moscow SU 1	1	94	2,824	C++
Shortest	Moscow SU 1	1	94	2,824	C++
Max atts.	Belarusian SU 2	12	291	3,770	C++

Consider the set of **edges** that are covered with the given paths and their endpoints. It is easy to see that no vertex may be adjacent to more than two covered edges, otherwise there are at least two paths

passing through this vertex that are not subsets of each other (when we treat path as a set of vertices) that is not allowed by the definition of a laminar family.

Thus, if the given family is laminar, all the covered edges consist of multiple disjoint simple tree paths. Finally, let's notice that the problem restricted to some of these simple paths is similar to checking if the given bracket sequence is correct: sort all the path endpoints along the path. Filter the single vertex paths, after that each path endpoint becomes either opening or closing.

If there is a vertex that contains both the opening and the closing endpoints, the set family is not laminar. Otherwise, sort the endpoints located at the same vertex in the descending order of path length if the vertex contains opening endpoints, and in the ascending order otherwise. Iterate over all the endpoints in the obtained order, all currently opened endpoints in the stack with a standard correctness-checking procedure for bracket sequence. By running this procedure for all the connected components of covered edges, we solve the problem.

The final question is to find all the covered edges. It can be done by making the tree rooted, expressing each path as a union of two vertical paths (that involves finding the highest point of each path with using an LCA query), and then running a DFS. But there is an easier and shorter trick for doing that: we can assign each path an random 64-bit mask using the unsigned int64 data type, assign each vertex the xor value of all paths that start in this particular vertex and then calculate for each edge the xor sum of all vertices in any of its subtrees. If the edge is not covered, then this xor-sum will be definitely zero, otherwise it will be non-zero with a probability of $1 - 2^{-64}$ that is indistinguishable from 1.