Problems summary

Recap: 299 teams, 13 problems, 5 hours. This analysis assumes knowledge of the problem statements (published separately on http://neerc.ifmo.ru/ web site).

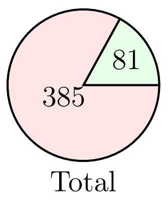
Summary table lists problem name and stats:

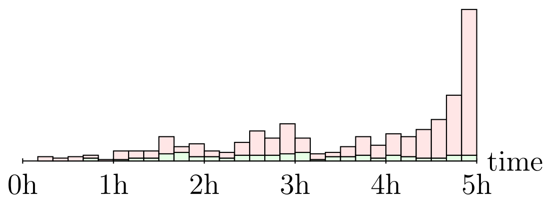
- **author** — author of the original idea
- **developer** — developer of the problem statement and tests
- **acc** — the number of teams that had solved the problem (gray bar denotes a fraction of the teams that solved the problem)
- **runs** — the number of total attempts
- **succ** — overall successful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

problem name	author	developer	acc/runs	succ
Alice the Fan	Oleg Hristenko	Niyaz Nigmatullin	81 /466	17%
Bimatching	Pavel Irzhavski	Pavel Irzhavski	0 /53	0%
Cactus Search	Borys Minaiev	Borys Minaiev	26 /121	21%
Distance Sum	Gennady Korotkevich	Gennady Korotkevich	0 /19	0%
Easy Chess	Mikhail Dvorkin	Mikhail Dvorkin	249 /565	44%
Fractions	Dmitry Yakutov	Dmitry Yakutov	148 /677	21%
Guest Student	Mikhail Mirzayanov	Mikhail Mirzayanov	225 /589	38%
Harder Satisfiability	Andrey Stankevich	Artem Vasilyev	1 /11	9%
Interval-Free Permutations	Andrey Stankevich	Pavel Kunyavsky	2 /5	40%
JS Minification	Roman Elizarov	Roman Elizarov	5 /50	10%
King Kog's Reception	Vitaliy Aksenov	Vitaliy Aksenov	20 /65	30%
Lazyland	Pavel Mavrin	Pavel Mavrin	247 /490	50%
Minegraphed	Mikhail Dvorkin	Mikhail Dvorkin	66 /278	23%

Problem A. Alice the Fan

Author:	Oleg Hristenko
Statement and tests:	Niyaz Nigmatullin





	Java	Kotlin	C++	Python	Total
🟩 Accepted	1	0	80	0	81
🟥 Rejected	2	0	377	6	385
Total	3	0	457	6	466

solution	team	att	time	size	lang
Fastest	ITMO 4	3	42	2819	C++
Shortest	BelarusianSU 5	2	157	1907	C++
Max atts.	MISiS 4	10	294	5557	C++

One of the solutions is to use dynamic programming.

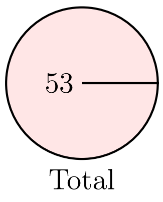
Let $f_{s,t,a,b}$ be whether it is possible to play $s + t$ sets, such that current match score is $s:t$ and the total number of points scored is $a:b$.

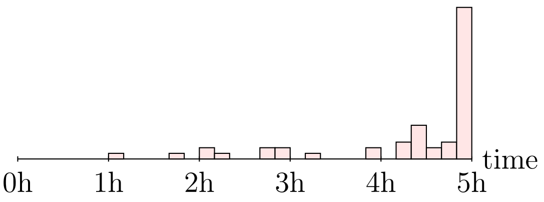
To calculate $f_{s,t,a,b}$ we can iterate over all possible set scores $x:y$ and check if it's possible $f_{s-1,t,a-x,b-y}$, if $x > y$, or $f_{s,t-1,a-x,b-y}$, if $x < y$.

As the number of matches is large, it's required to pre-calculate f once. For each given match it's possible to iterate over all match scores $s:t$ in the decreasing order $s-t$, and find first match score that $f_{s,t,a,b} = 1$. And then find the set scores using any known dynamic programming answer restoring technique. For example, one of the ways to do that is to store the last set score for each $\{s, t, a, b\}$.

Problem B. Bimatching

Author:	Pavel Irzhavski
Statement and tests:	Pavel Irzhavski





	Java	Kotlin	C++	Python	Total
🟩 Accepted	0	0	0	0	0
🟥 Rejected	0	0	53	0	53
Total	0	0	53	0	53

Consider the following graph G. The graph G has a vertex v_ℓ corresponding to the lady ℓ for $\ell = 1, 2, \ldots, m$ and two vertices u_c and u'_c corresponding to the cavalier c for $c = 1, 2, \ldots, n$. The vertex v_ℓ is adjacent to both vertices u_c and u'_c if and only if the lady ℓ can dance with the cavalier c. The vertices u_c and u'_c are adjacent for the each cavalier $c = 1, 2, \ldots, n$.

Lemma. The cardinality of the maximal matching in graph G is equal to $n + k$, where k is equal to the maximal bimatching in the original graph.

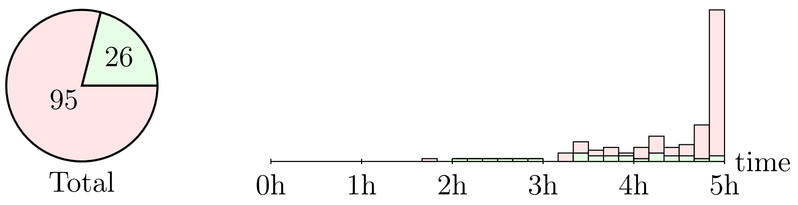
Proof. If the maximal bimatching in the original graph is equal to k, we can construct the matching of size $n+k$ in graph G following way: for each bimatching $(u_c, v_{\ell_1}, v_{\ell_2})$ take edges (u_c, v_{ℓ_1}) and (u'_c, v_{ℓ_2}) to a matching. For each unmatched by bimatchings vertex u_c take edge (u_c, u'_c) to a matching. These $n + k$ edges form a correct matching in G.

If we have a matching of size $n + k$ (we always can make a matching with n edges just by edges between (u_c, u'_c), then we can make a bimatching in original graph of size k. We have $2n$ vertices, so there are at least k pairs u_c, u'_c, covered by a matching, such that the edge (u_c, u'_c) is not in matching. We can make a bimatching from such edges, and get exactly correct k-bimatching in the original graph.

So, we reduced the original problem to finding a maximal matching in a general graph. It can be done by a lot of different algorithms, running between $\mathcal{O}(n^{5/2})$ and $\mathcal{O}(n^4)$, but any of them should get accepted.

Problem C. Cactus Search

Author:	Borys Minaiev
Statement and tests:	Borys Minaiev



	Java	Kotlin	C++	Python	Total
🟩 Accepted	0	0	26	0	26
🟥 Rejected	0	0	95	0	95
Total	0	0	121	0	121

solution	team	att	time	size	lang
Fastest	ITMO 2	2	127	2417	C++
Shortest	BelarusianSU 5	1	269	1230	C++
Max atts.	KBTU 1	8	248	2372	C++

Let's maintain a set of vertices which could be picked by Chloe. If the size of the set becomes at least two times smaller after each query than before it, then after 9 queries the set will contain at most one vertex.

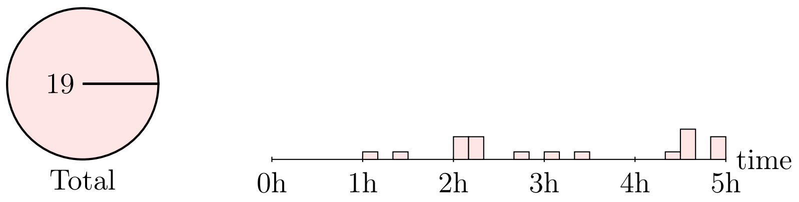
It is always possible to choose a vertex to query in such a way that the set of candidates decreases two times. For example, we can choose a vertex with a minimal sum of distances to the candidates set.

It could be proved by contradiction that such a vertex is good. If we asked a vertex u and Chloe responded with a vertex w and more than half of paths from u to candidates goes through w, it means that the sum of distances from w to candidates if less than similar sum from u. So initial assumption (u has smallest sum of distances) is incorrect.

The complexity of the algorithm is $O(n^2)$, but we need to find a hidden vertex n times, so the overall complexity is $O(n^3)$.

Problem D. Distance Sum

Author:	Gennady Korotkevich
Statement and tests:	Gennady Korotkevich



	Java	Kotlin	C++	Python	Total
🟩 Accepted	0	0	0	0	0
🟥 Rejected	2	0	8	9	19
Total	2	0	8	9	19

Let's solve a generalized version of the problem: each vertex i has weight w_i, we need to find $\sum\limits_{u=1}^{n-1}\sum\limits_{v=u+1}^{n} w_u w_v d(u,v)$. The original problem corresponds to $\forall i : w_i = 1$.

Consider any vertex x of degree 1, and let its only neighbor be y. Clearly, every shortest path from x to another vertex starts with edge $x \sim y$. Thus, the contribution of this edge to the answer is equal to

$w_x \cdot \sum_{i \neq x} w_i$. Let's increase the answer by this value, increase w_y by w_x, and remove vertex x together with edge $x \sim y$ from the graph. This way we reduce the size of the graph without changing the answer, the graph stays connected, and $m \le n + 42$ still holds.

Once we eventually get rid of vertices of degree 1 altogether, there might be just one vertex of degree 0 left if the original graph was a tree — in this case we're done. Otherwise every vertex has degree at least 2.

Note that $\sum_i \deg(i) = 2m \le 2n + 84$. Thus, there are at most 84 vertices of degree more than 2. We'll call these vertices *special*. If there are no special vertices, the graph is just a cycle — then we'll pick any vertex on the cycle and call it special. Now the graph consists of special vertices connected by paths of vertices of degree 2.

Let's run breadth-first search from every special vertex to find the distances to all other vertices.

For every vertex v, let $s(v) = \sum_{u \neq v} w_u d(u, v)$, then the answer is $\frac{1}{2} \sum_v s(v)$.

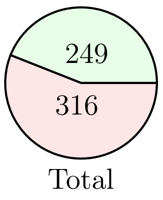
Finally, let's calculate $s(v)$. For each special vertex u, the distance from v to u is known from BFS. Consider a non-special vertex u lying on a path between special vertices u_1 and u_2; let this path be $u_1, x_1, x_2, \ldots, x_k, u_2$.

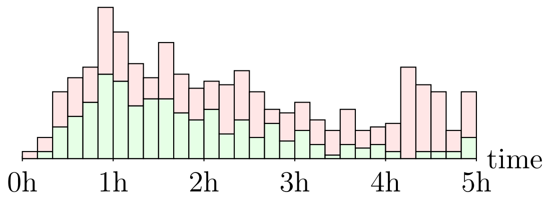
If v doesn't lie on the same path, the shortest path from v to u visits either u_1 or u_2. To be exact, for some $t \in [0; k]$, the shortest path from v to $x_1, x_2, \ldots x_t$ visits u_1, and the shortest path from v to $x_{t+1}, x_{t+2}, \ldots, x_k$ visits u_2. The value of t can be calculated based on $d(v, u_1)$, $d(v, u_2)$, and k. The part of $s(v)$ dependent on $x_1, x_2, \ldots, x_k$ can be calculated based on prefix sums of $w_{x_1}, w_{x_2}, \ldots, w_{x_k}$ and $1 \cdot w_{x_1}, 2 \cdot w_{x_2}, \ldots, k \cdot w_{x_k}$.

Case $v = x_i$ can be handled in a similar fashion.

Problem E. Easy Chess

Author:	Mikhail Dvorkin
Statement and tests:	Mikhail Dvorkin





	Java	Kotlin	C++	Python	Total
🟩 Accepted	8	1	219	21	249
🟥 Rejected	67	1	198	50	316
Total	75	2	417	71	565

solution	team	att	time	size	lang
Fastest	MIPT 1	1	17	2237	C++
Shortest	Ataturk-Alatoo 2	2	167	417	Python
Max atts.	IrkutskNRTU 1	54	293	3542	Java

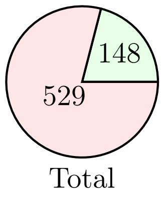
One possible short-to-code solution is to move from left to right, visiting 0 or more cells in each column. The minimal amount of visited cells per column would be [1, 0, 0, 0, 0, 0, 0, 2]. If n (the desired number of moves) is greater than 2, increase these numbers by 1 arbitrarily $n - 2$ times, but never exceeding 8.

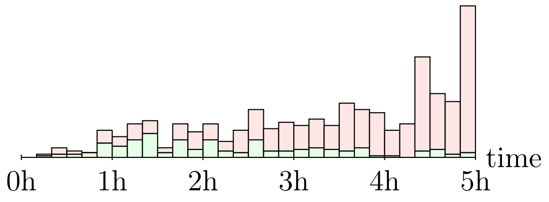
Now, for each column, let m be the desired amount of visited cells in it. If $m = 0$, just skip the column. Otherwise, enter this column in the row that you know from the previous columns (or initialize at row 1 in column `a`), and then visit $m-1$ other arbitrary cells in this column, and remember in which row you left it.

The choice of visited cells is indeed arbitrary, with two exceptions: forbid yourself to leave columns `a`–`g` in row 8 (because in that case you wouldn't be able to process column `h` correctly), and make sure you finish in row 8 in column `h`.

Problem F. Fractions

Author:	Dmitry Yakutov
Statement and tests:	Dmitry Yakutov





	Java	Kotlin	C++	Python	Total
🟩 Accepted	0	0	143	5	148
🟥 Rejected	4	2	512	11	529
Total	4	2	655	16	677

solution	team	att	time	size	lang
Fastest	SPbSU 3	1	14	1224	C++
Shortest	CrimeanFU 1	1	123	381	Python
Max atts.	RybinskSATU	10	298	2515	C++

Consider the following cases:

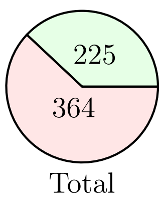
- $n = p^m$ where p is a prime number. It means that for $i = 1 \dots k$ b_i is less than n and b_i divides n, so b_i divides $p^{m-1} = \frac{n}{p}$. Therefore sum of all fractions $\frac{a_i}{b_i}$ has a denominator which divides $\frac{n}{p}$, so the sum cannot be equal to $1 - \frac{1}{n}$.
- n is not power of a prime number. It means that n can be expressed as product of two numbers $x > 1$ and $y > 1$ such that $\gcd(x, y) = 1$. For example, if $n = p_1^{m_1} \cdot \ldots \cdot p_t^{m_t}$ then x can be $p_1^{m_1}$ and y can be $p_2^{m_2} \cdot \ldots \cdot p_t^{m_t}$. Without loss of generality we can say that $x \le y$, otherwise we can swap values of x and y. Let's find the sequence of fractions of the following form: $k = 2$, $\frac{a_1}{b_1} = \frac{c}{x}$, $\frac{a_2}{b_2} = \frac{d}{y}$. We need to solve an equation: $\frac{c}{x} + \frac{d}{y} = 1 - \frac{1}{n}$ or $c \cdot y + d \cdot x = n - 1$.

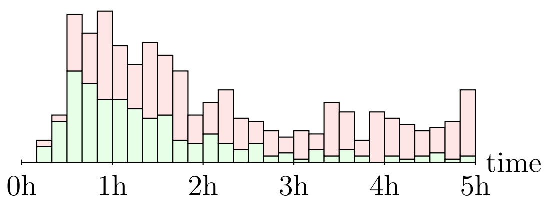
  Lets iterate over all values of c from 1 and $x - 1$. Value of $(c \cdot y) \mod x$ is iterating from 1 to $x - 1$ in some order because $c \cdot y$ is not divisible by x and values of $c \cdot y$ differ if values of c differ because x and y are coprime. Therefore there exists c such that $c \cdot y + 1$ is divisible by x. Let's take this value of c and let's say $d = \frac{n-1-c \cdot y}{x}$. n is divisible by x, so $n - 1 - c \cdot y$ is divisible by x. Also $1 + c \cdot y \le 1 + (x - 1) \cdot y = n - (y - 1) < n$, so $d > 0$. It means that pair of fractions $(\frac{c}{x}, \frac{d}{y})$ is correct sequence of fractions.

It is needed to factorize number n to find the answer. It can be done using $\mathcal{O}(\sqrt{n})$ time. After it is needed to iterate over values of c. There are $x - 1$ possible values and $x = \sqrt{x \cdot x} \le \sqrt{x \cdot y} = \sqrt{n}$, so this part can be done using $\mathcal{O}(\sqrt{n})$ time. The total complexity of solution is $\mathcal{O}(\sqrt{n})$.

Problem G. Guest Student

Author:	Mikhail Mirzayanov
Statement and tests:	Mikhail Mirzayanov





	Java	Kotlin	C++	Python	Total
🟩 Accepted	4	1	208	12	225
🟥 Rejected	25	0	324	15	364
Total	29	1	532	27	589

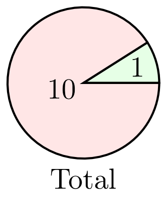
solution	team	att	time	size	lang
Fastest	IvanovoSPowU	1	14	810	C++
Shortest	YerevanSU 2	2	64	443	Python
Max atts.	IrkutskSU 3	9	242	1456	C++

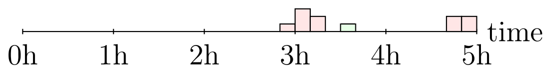
Let's iterate over all possible days of a week to start studying. We will solve problem independently for each starting day of a week and choose the best result.

To solve a problem for fixed starting day let's find `n = max(0, k / sum - 1)` — lower bound for the number of whole weeks (where `sum` is $a_1 + a_2 + \cdots + a_7$). After it you can iterate day by day to study exactly k days.

Problem H. Harder Satisfiability

Author:	Andrey Stankevich
Statement and tests:	Artem Vasilyev





	Java	Kotlin	C++	Python	Total
🟩 Accepted	0	0	1	0	1
🟥 Rejected	0	0	10	0	10
Total	0	0	11	0	11

solution	team	att	time	size	lang
The only	MSU 3	3	210	2318	C++

Let's convert given `OR` clauses into an implication digraph, the same way as in classic 2-SAT solution. Build a condensation of that graph. We'll call a variable *universal* if the corresponding quantifier is $\forall$, and an *existential* if the corresponding quantifier is $\exists$. Now a fully quantified formula is false if and only if at least one of the three following properties is true:

1. A variable x_i is in the same strongly connected component (SCC) as its negation $\overline{x_i}$
2. An existential variable x_i is in the same SCC as a universal variable x_j such that x_i precedes x_j (so, $i < j$).
3. A universal vertex (corresponding to some variable x_i or its negation) is reachable from another universal vertex.

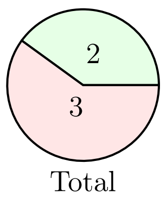
It's easy to see that if one of these three conditions is true, then the formula is false. Otherwise, there is an algorithm that marks every SCC as `false`, `true` or `any` (for those components that contain a universal vertex). Process SCCs in order of reverse topological order. If the current component S wasn't already marked, mark it as `any` if it contains a universal vertex. Otherwise, if S has a `false` or `any` successor, mark it as false. In the other case, mark it as `true`. Mark the negative component, corresponding to negations of all vertices in S accordingly. If this marking does not yield a solution, then it's possible to find either a proof of one of three statements above.

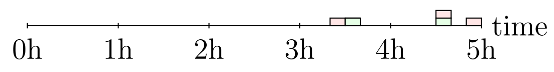
Now we have to check if any of these three conditions is true. To check the first condition iterate over all variables and check if x_i and $\overline{x_i}$ are in different SCCs. To check the second condition, again, iterate over all variables, mark a component whenever you encounter an existential variable, and check whenever you see a universal variable. To check the third condition, iterate over all components in reverse topological order and keep track of components, from which you can reach a universal vertex. If there is a component with at least two universal vertices, or a component with a universal vertex, from which another universal vertex is reachable, then the formula is false.

The implication graph can be build and compressed in $O(n+m)$ time. Checking all the required conditions can be also implemented in $O(n+m)$ time, so in the end we have a linear time algorithm for this problem.

Problem I. Interval-Free Permutations

Author:	Andrey Stankevich, Pavel Kunyavskiy
Statement and tests:	Pavel Kunyavsky





	Java	Kotlin	C++	Python	Total
🟩 Accepted	0	0	2	0	2
🟥 Rejected	0	0	3	0	3
Total	0	0	5	0	5

solution	team	att	time	size	lang
Fastest	MIPT 6	2	216	2063	C++
Shortest	MSU 3	2	279	1468	C++
Max atts.	MSU 3	2	279	1468	C++

We will use dynamics programming. Good permutations, are all except bad. To count bad permutations we will first need to understand some of their structure.

Let's call interval maximal if it isn't inside any bigger interval, except full permutation. It's easy to see, that if two intervals intersects, their union is interval too. So, if two maximal intervals intersects, they cover full permutation. This yields two possibilities: bad permutation is either concatenation of 3 or more maximal intervals, or two maximal intervals that cover all permutation. We will count this two cases separately, as each of them excludes other.

Easier case is for two intervals. We have two symmetric cases — when first element is bigger than last, and visa versa. Number of permutation for both cases is same, let's consider first of them. This requires that none of prefix of left permutation is prefix of integers. Number of such permutations can be found be dynamics programming, using same idea — good permutations are all except bad, bad is good prefix + anything else. And right permutation can be anything.

For the other case, we can see, that nothing depends on permutation inside each interval, so they can be any permutations. The only restriction we have — none of our intervals can form bigger interval. But this means, that permutation of relative order of intervals is interval-free. As their number is less than n (otherwise all of them have length 1), we already know number of such permutations. And only thing to do, is multiply them by number of ways to choose given number of permutations with given total length, which can be easily precomputed in cubic time.

To summarise, we need to count three following recurrences:

- Number of permutation, none of prefixes of which is permutation.

  $I_n = n! - \sum\limits_{k=1}^{n-1} I_k \cdot (n-k)!$

- Number of ways to choose k permutations of total length n.

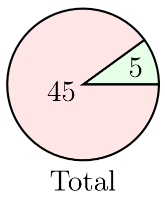
  $B_{n,k} = \sum\limits_{t=1}^{n} B_{n-t,k-1} \cdot t!$

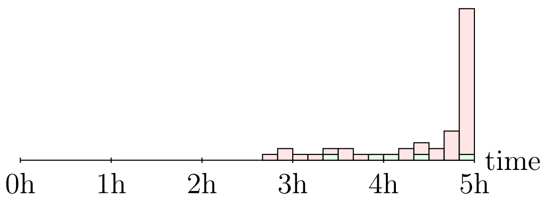
- The answer

  $A_n = n! - 2 \cdot \sum\limits_{k=1}^{n-1} I_k \cdot (n-k)! - \sum\limits_{k=3}^{n-1} B_{n,k} \cdot A_k$

Problem J. JS Minification

Author:	Roman Elizarov
Statement and tests:	Roman Elizarov





	Java	Kotlin	C++	Python	Total
🟩 Accepted	0	0	5	0	5
🟥 Rejected	0	0	44	1	45
Total	0	0	49	1	50

solution	team	att	time	size	lang
Fastest	MIPT 6	2	203	7724	C++
Shortest	MSU 3	1	242	4493	C++
Max atts.	ITMO 4	5	261	6512	C++

The first part of the problem is to write code that parses the input. Process the input source line by line. On each like you should repeatedly skip space and parse tokens. The start of a potential word or a number is determined by its first character according to the problem statement, parsing the longest possible word or number while the corresponding characters are encountered. However, there could a case when there is a longer reserved token that had started at the same position. In the limits of this problem all n possible reserved tokens can be exhaustively checked at the current parsing position. The longest matching reserved token or the longest word/number token shall be chosen. Thus, the whole input is transformed into a sequence of tokens.

The second part of the problem is word renaming. Maintain a map from source word to output word and use renaming from this map when a source word is encountered for a second time. To handle a new source word you need a procedure to generate a *target word list*. This can be done by keeping the value of the last used target word and using a procedure to generate the next target word from the previous one, which is basically like incrementing an integer in base 26. You will need to check each candidate target word against a list of reserved tokens, and repeat looking at the next target word until it is not in the list of reserved tokens.

The third part of the problem is writing the resulting sequence of tokens (after rename) to the output while inserting the minimum number of spaces. This can be done greedily, because a space is a universal separator and cannot be part of any token. Write tokens to the output and check if a space must be inserted before a token to prevent erroneous parsing of the resulting string. To streamline this check, you should keep a list of all the tokens that were output after the last space (or after the beginning of the source).

Note, that if a previous token conforms to a number token rule (regardless of whether it matches some reserved token or not), then adding a token that starts with a digit affects the parsing, because the previous token could have been parsed to a longer number token. Thus, a space must be inserted in this case.

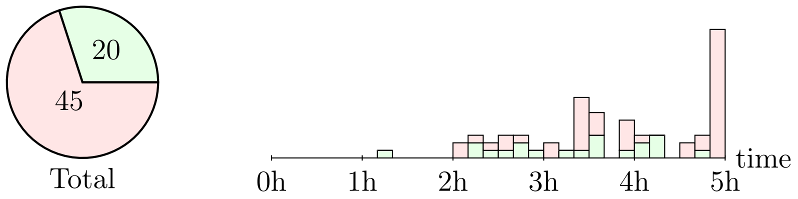
Similarly note, that if a previous token conforms to a word token rule (even if it matches reserved token), then adding a token that starts with a letter, digit, underscore or dollar sign would similarly affect the parsing. Thus, a space must be inserted in this case, too.

Now, the remaining case of parsing confusion comes from reserved tokens. This can happen at the beginning of any of the previous tokens. To detect this case, scan the list of the previous tokens (since the last output space) starting from the most recently output token backwards and check if parsing from that position could result in recognition of a reserved token because of the new token you are about to output without a space. There is a limit of 20 characters on the length of a reserved token, so this backward iteration can be terminated after going more than 20 characters backwards in the output. This avoids $O(k^2)$ time, where k in the number of tokens in the input source.

There are different working approaches to this check. In particular, you can avoid separate checks for words and numbers and just do a straightforward attempt to parse a token from each test position, assuming that you output a new token without a space. If that check shows that a different token can get parsed, then a space must be output before the token you are about to write. Don't forget to clear a list of recently output tokens as you write a space to the output.

Problem K. King Kog's Reception

Author:	Vitaliy Aksenov
Statement and tests:	Vitaliy Aksenov



	Java	Kotlin	C++	Python	Total
🟩 Accepted	0	0	20	0	20
🟥 Rejected	0	0	45	0	45
Total	0	0	65	0	65

solution	team	att	time	size	lang
Fastest	MIPT 6	1	79	3191	C++
Shortest	SPbHSE 1	2	250	2168	C++
Max atts.	UofLatvia 1	5	219	3819	C++

Let t_i be the time of arrival of i-th knight, and d_i be his duration time.

Then, it can be seen that the answer to the query with the parameter T is equal to the $\max_{i:\ t_i \le T}(t_i + \sum_{j: t_i \le t_j \le T} d_j)$. In other words, we take the maximum among the sums for each knight i that come before Teabeanie: the time at which i comes plus the duration of all the visits that initiate after i comes, inclusive, but before Keabeanie comes, inclusive.

This value is a function of T, but there is a trick to eliminate this. In the second term, instead of the sum between t_i and T, we can rather calculate the sum from t_i to the infinity and subtract the sum from $T + 1$ to the infinity.

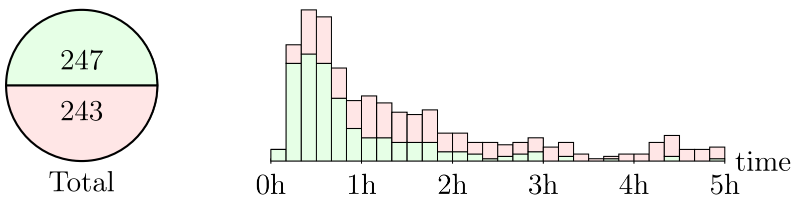
The subtracted part can be calculated with an interval tree, or a similar data structure.

Now, the sum from t_i to the right is a formula that doesn't depend on T. So we can use an interval tree (or a similar data structure) to store the values $(t_i + \sum_{j: t_i \le t_j} d_j)$.

This interval tree should support taking maximum (which is needed to calculate the answer for Keabeanie's queries). Also, when a knight joins or cancels, this event affetcs all cells to the left of his t_i in an additive way, and the cell t_i itself. So the interval tree should support addition on a segment, as well as single cell alteration.

Problem L. Lazyland

Author:	Pavel Mavrin
Statement and tests:	Pavel Mavrin



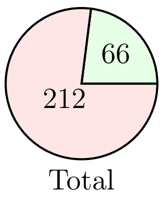
	Java	Kotlin	C++	Python	Total
🟩 Accepted	7	1	226	13	247
🟥 Rejected	14	0	204	25	243
Total	21	1	430	38	490

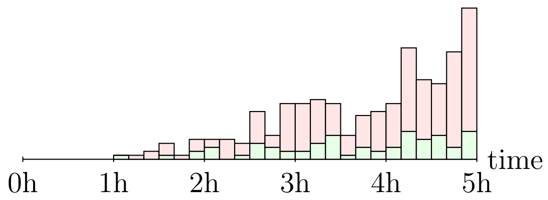
solution	team	att	time	size	lang
Fastest	BelarusianSU 1	1	7	965	C++
Shortest	SUrSU 3	1	30	367	Python
Max atts.	RybinskSATU	8	121	955	C++

For each job, find the idler with the maximum value of b_i, assign this job to this idler. Put all remaining idlers in the array, sort them by the value of b_i and assign idlers with minimum values to remaining jobs.

Problem M. Minegraphed

Author:	Mikhail Dvorkin
Statement and tests:	Mikhail Dvorkin





	Java	Kotlin	C++	Python	Total
🟩 Accepted	1	0	65	0	66
🟥 Rejected	6	0	206	0	212
Total	7	0	271	0	278

solution	team	att	time	size	lang
Fastest	ITMO 1	1	67	2304	C++
Shortest	UrFU 7	3	254	1844	C++
Max atts.	TbilisiIBSU 1	9	255	2286	C++

Here's one possible simple construction that needs only a $3n \times 3n \times 3$ parallelepiped.

The field is generally filled with obstacles, with some exclusions, of course.

The vertex i will correspond to a north-south full-length tunnel in the bottom layer plus a west-east full-length tunnel in the top layer. It's enough to have two obstacle cells between neighbour tunnels. (Thus the tunnels are $x = 3i \wedge z = 0$ and $y = 3i \wedge z = 2$).

For each i, to make these two tunnels connected, build a two-step "staircase" near their almost-intersection.

Now for each edge $i \to j$ in the graph, go to the almost-intersection of the i-th vertex's top tunnel and the j-th vertex's bottom tunnel. Simply dig a full-height vertical hole near this place, so that it's possible to fall from top tunnel of i to bottom tunnel of j, but not possible to climb back.