Problems summary

Recap: 310 teams, 12 problems, 5 hours. This analysis assumes knowledge of the problem statements (published separately on http://nerc.itmo.ru/ web site).

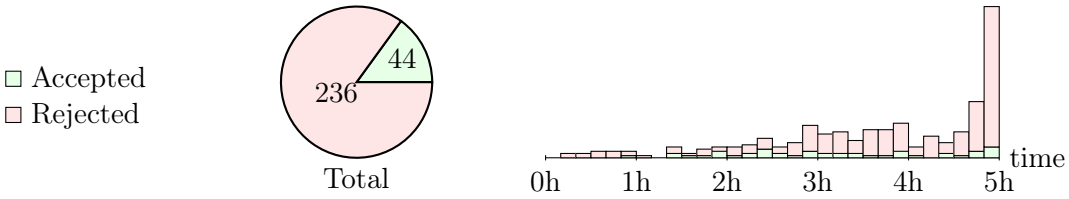
Summary table lists problem name and stats:

- **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
Apprentice Learning Trajectory	Vitaliy Aksenov	Mikhail Dvorkin, Ilya Zban	44 /280	15%
Balls of Buma	Vitaliy Aksenov	Vitaliy Aksenov	273 /575	47%
Cactus Revenge	Gennady Korotkevich	Gennady Korotkevich	0 /35	0%
DevOps Best Practices	Dmitry Yakutov	Dmitry Yakutov	18 /113	15%
Elections	Pavel Mavrin	Pavel Mavrin	197 /516	38%
Foolprüf Security	Artem Vasilyev	Artem Vasilyev	15 /38	39%
Game Relics	Niyaz Nigmatullin	Niyaz Nigmatullin	3 /32	9%
Help BerLine	Mikhail Mirzayanov	Mikhail Mirzayanov, Borys Minaiev	0 /14	0%
Intriguing Selection	Petr Mitrichev	Petr Mitrichev	36 /495	7%
Just Arrange the Icons	Mikhail Mirzayanov	Mikhail Mirzayanov, Pavel Kunyavsky	159 /645	24%
Key Storage	Elena Kryuchkova	Pavel Kunyavsky	105 /214	49%
Lexicography	Georgiy Korneev	Georgiy Korneev	190 /715	26%

Problem A. Apprentice Learning Trajectory

Author: Vitaliy Aksenov
Statement and tests: Mikhail Dvorkin, Ilya Zban



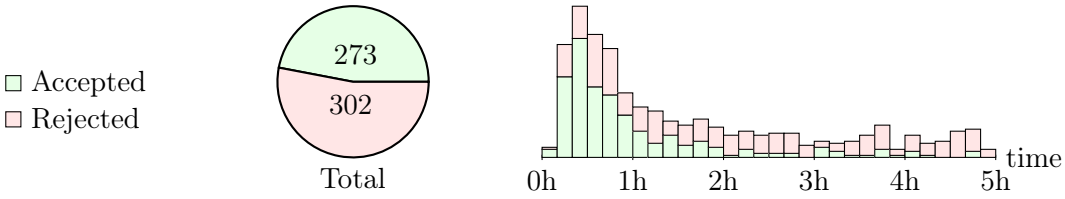
solution	team	att	time	size	lang
Fastest	SPbITMO: Reduce	1	55	1934	C++
Shortest	IntITU: 2	1	181	1123	C++
Max atts.	MIPT: LinkCat	9	292	2351	C++

There is an obvious greedy solution in $\mathcal{O}(nt)$: each time we can forge the sword which will be forged earlier than the other.

This solution can be optimized to the $\mathcal{O}(n \log n)$: greedy algorithm don't often change the chosen master, it can happen when one master starts or ends working. There are $\mathcal{O}(n)$ events when masters open and close their forges, and between these events we can just use the fastest available master.

Problem B. Balls of Buma

Author: Vitaliy Aksenov
Statement and tests: Vitaliy Aksenov



solution	team	att	time	size	lang
Fastest	NN SU: Retired	1	8	1210	C++
Shortest	KyrTurkMU: 8	1	50	394	Python
Max atts.	NovSPedU: 1	17	238	757	Python

To start with we "compress" the neighbouring balls of the same color to segments. For example, '`AAABBAAAAC`' becomes '`(3, A), (2, B), (4, A), (1, C)`'. Suppose that we insert a ball into i-th segment. If our ball is of different color or the number of balls in i-th segment is less than 2, then nothing happens and the elimination stops. After the elimination of this segments, $i-1$-th and $i+1$-th segments become neighbours. If they are of the same color and have total length at least 3, then "new" joint segment becomes eliminated, otherwise, nothing happens and the elimination stops. Then, $i-2$-th and $i+2$-th segments become neighbours, and so on.

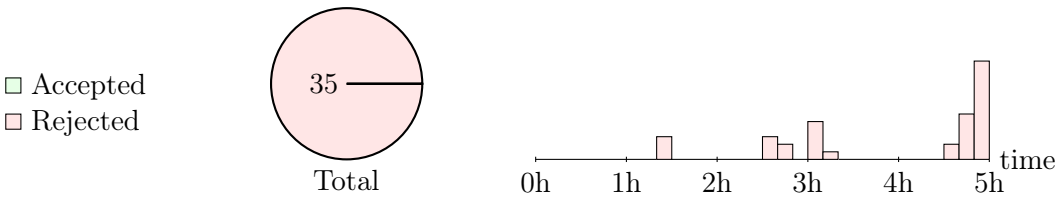
Thus, we can see that to eliminate all the balls we should insert a ball into the middle segment and the answer should be either zero or the number of balls in the middle segment plus one (we can insert a ball either between the balls or to the left and to the right of the segment).

Summing everything up, we should check the following criteria. If at least one is not satisfied then we cannot eliminate all the balls and the answer is zero:

- The total number of segments m is odd.
- The number of balls in the middle ($m/2+1$-th) segment is not one, i.e., by adding a ball the length of the segment should become at least 3.
- The paired segments, i.e., $m/2+1-i$-th and $m/2+1+i$-th, should have at least three balls in total and all the balls in them should be of the same color.

Problem C. Cactus Revenge

Author: Gennady Korotkevich
Statement and tests: Gennady Korotkevich



The number of edges in the cactus $m = \frac{1}{2}\sum_{i=1}^{n} d_i$. If m isn't an integer or $m < n-1$, there is no solution.

If $m = n-1$, a tree can always be built inductively: remove any leaf v (i.e. a vertex s.t. $d_v = 1$), decrement the degree of the vertex with the largest degree u, build a tree on the remaining $n-1$ vertices, and restore v connecting it to u with an edge.

Otherwise, the number of simple cycles our cactus must have is $c = m - (n - 1)$. Any spanning tree of a cactus contains $n - 1$ edges, while each of the remaining c edges creates a simple cycle that passes through at least 2 edges of the tree. As every tree edge must belong to at most one simple cycle, a solution might only exist if $c \le \left\lfloor \frac{n-1}{2} \right\rfloor$.

If all degrees are even, the $c \le \left\lfloor \frac{n-1}{2} \right\rfloor$ condition is actually sufficient. We can build a cactus inductively: if $d_i = 2$ for all i, just build a cycle; otherwise, pick vertices u and v s.t. $d_u = d_v = 2$ (they always exist, otherwise c is too large) and a vertex w s.t. $d_w \ge 4$, remove u and v and decrease d_w by 2, build a cactus on the remaining $n - 2$ vertices, and restore u and v connecting u, v, and w into a new cycle.

If some degrees are odd, we might need a stronger condition. Let's turn our attention to *bridges*. Suppose that we have o vertices with odd degrees, and l of them are leaves. Every vertex with an odd degree must have an incident bridge, and all leaves must have *distinct* incident bridges. Therefore, the number of bridges in our cactus, b, can be bounded as $b \ge \max(\frac{o}{2}, l)$.

As bridges do not belong to any cycles, we can further bound c as $c \le \left\lfloor \frac{n-1-\max(\frac{o}{2},l)}{2} \right\rfloor$. Finally, this condition is sufficient, which can be shown constructively as follows. If $l \le \frac{o}{2}$:

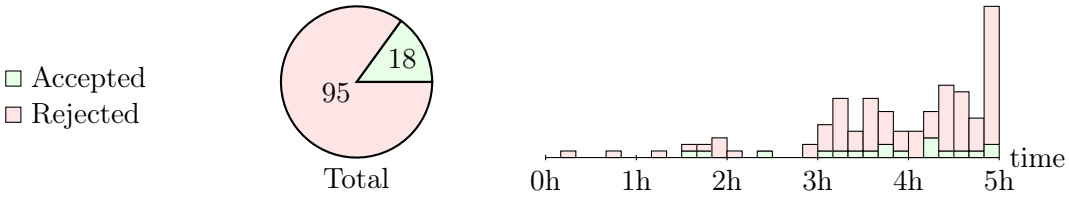
- split vertices with odd degrees into pairs arbitrarily, but make sure not to pair up two leaves;
- connect vertices in every pair with an edge and contract all these edges. In general, when an edge between vertices u and v is contracted, a new vertex with degree $d_u + d_v - 2$ replaces u and v;
- build a cactus on the remaining $n - \frac{o}{2}$ vertices: this cactus must have the same number of simple cycles c, and since $c \le \left\lfloor \frac{n-\frac{o}{2}-1}{2} \right\rfloor$ and all degrees are even, this has been shown to be possible;
- expand the contracted edges. Be careful while splitting the edges incident to the new vertices between the original vertices: pairs of edges belonging to the same cycle must be incident to the same original vertex after expansion.

If $l > \frac{o}{2}$, only the first two steps are different:

- pair $o - l$ leaves with all $o - l$ non-leaf odd-degree vertices arbitrarily, connect vertices in every pair with an edge and contract all these edges. At this point, every degree is either even or equal to 1;
- pair the remaining $2l - o$ leaves arbitrarily, connect leaves in every pair to an arbitrary vertex with degree at least 4 with two edges, contract these edges as well.

Problem D. DevOps Best Practices

Author:	Dmitry Yakutov
Statement and tests:	Dmitry Yakutov



solution	team	att	time	size	lang
Fastest	IntITU: 1	2	97	1757	C++
Shortest	IntITU: 1	2	97	1757	C++
Max atts.	HSE: Sharingan pwr	8	283	4539	C++

Let's set the following target: to deploy all needed features to as many servers as possible using one edge per server. Note the following facts:

1. All edges from server s_i distribitue the same set of features.
2. If CD configuration doesn't deploy all needed features to server s_2 (set F) and server s_1 distributes set of features F, then it is enough to add CD edge from s_1 to s_2 and do not add other edges to s_2.
3. If we deployed all needed features to server s_2 using one edge from server s_1, then we need to turn on CT on s_2. Otherwise server s_2 can't help us to reach our target: all edges from s_2 can be replaced with edge from s_1.

Let's split all servers by number of features cnt that should be deployed to server:

- $cnt = 0$. Such server s doesn't need any features to be deployed, we don't need to add edges to s.
- $cnt = 3$. Such server needs all features, so we can add single edge from server 1 to these servers.
- $cnt = 1$. Such server needs only one feature, so there is no poing to add two or more edges to such server.
- $cnt = 2$. Such server s needs two features, sometimes we need to add two edges to s. In this case edge from s_1 adds feature f_1 and another edge from s_2 adds feature f_2. We don't need to turn on CT on s because there already exists a way to add f_1 using edge from s_1 and a way to add f_2 using edge from s_2. It is better to turn off CT on s and add edge from s to all other servers that need f_1 and f_2 to be deployed.

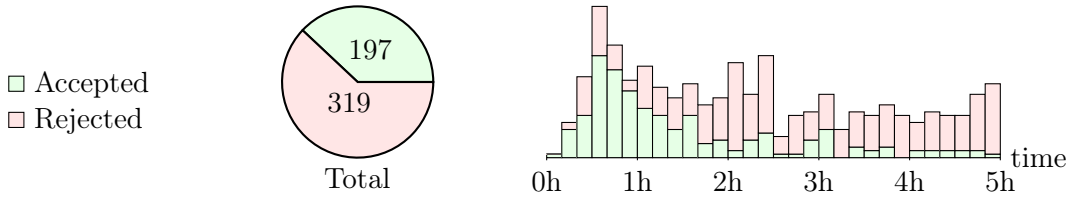
So the solution is the following repeating process.

- Add edge from s_1 to s_2 if s_1 distributes the exact set of features that is needed by s_2 and continue the process. Fact (2) gives us that we need to turn on CT on s_2 in this case.
- If it is impossible to perform previous step then try to deploy features to any server s with $cnt = 2$ using two edges and continue the process. Do not turn on CT on s in this case.
- If it is impossible to perform previous step then stop the process.
- If at the end of the process there exists server s with some needed undeployed feature f then it is impossible to configurate CD/CT.

This algorithm gives us at most $(n-1)+3 = n+2$ edges because we deploy features to s using two edges at most three times: to one server $s_{1,2}$ that needs features f_1 and f_2, to one server $s_{1,3}$ that needs features f_1 and f_3 and to one server $s_{2,3}$ that needs features f_2 and f_3.

Problem E. Elections

Author: Pavel Mavrin
Statement and tests: Pavel Mavrin



solution	team	att	time	size	lang
Fastest	SPbITMO: 1 StdDev	1	8	1698	C++
Shortest	Moscow SU: char* a	1	95	805	Python
Max atts.	Tomsk PU: 3	9	279	1447	C++

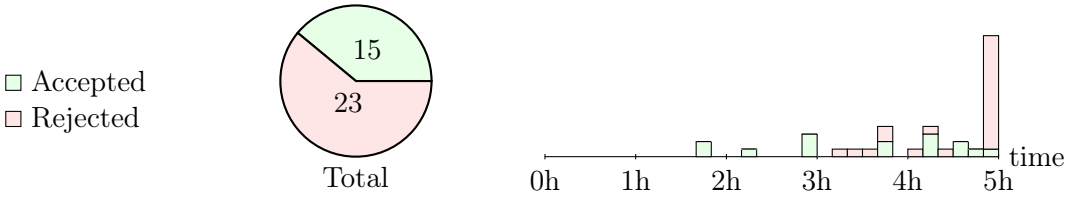
Let's fix the candidate who will be above the opposition one. We should check all candidates and choose minimal result among them.

When we have two candidates, the only important thing is difference between number of votes casted for them. Let's sort all polling stations by this difference. We need to remove biggest difference, until, number of votes casted for non-opposition candidate become smaller.

No optimization of this process required, naive implementation will work with complexity $O(n \cdot m \log m)$.

Problem F. Foolprüf Security

Author: Artem Vasilyev
Statement and tests: Artem Vasilyev



solution	team	att	time	size	lang
Fastest	MIPT: Godnotent	1	105	2192	C++
Shortest	Latvia: 2	1	175	1768	C++
Max atts.	MIPT: Fennecs	2	296	5585	C++

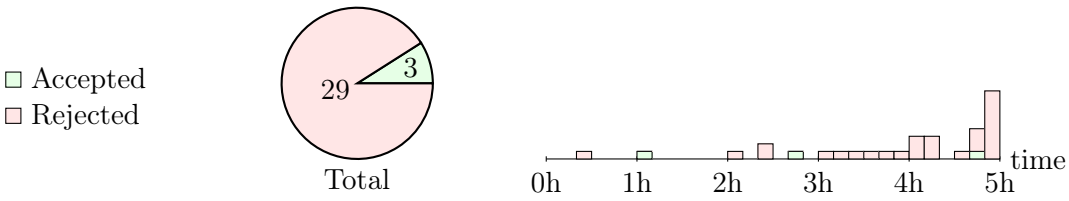
The answer is "`Yes`" if and only if $k_a \le m-1$ and $k_b \le n-1$. Let's look at the Prüfer code for an arbitrary spanning tree of a complete bipartite graph $K_{n,m}$ (which is what was described in the statement). Every time we delete a vertex from the left part, we add a vertex from the right into the code, and the other way around. In the end, only one edge remains, which means that $n-1$ vertices from the left part and $m-1$ vertices from the right part were deleted. This proves that the Prüfer code contains exactly $n-1$ numbers from $n+1$ to $n+m$ and exactly $m-1$ numbers from 1 to n.

Let's prove that any sequence of length $m-1$ consisting of numbers $[1, n]$ and any sequence of $n-1$ numbers $[n+1, n+m]$ can be uniquely interleaved to produce a spanning tree of $K_{n,m}$. Let's find the minimum number that doesn't appear in any one of these two sequences, call it v. This number is a vertex that was deleted on the first step. If v is between 1 and n, then it was connected to b_1 (first number in Bob's sequence). If v is between $n+1$ and $n+m$, then it was connected to a_1 (first number in Alice's sequence). After we figure out the first edge, we delete the neighbor of v (either a_1 or b_1) and continue restoring edges the same way. In the end, we will connect two remaining vertices with an edge.

Thus, if k_a is less than $m-1$, we can add arbitrary numbers to Alice's sequence and it will still be possible to restore the tree. The same is true for Bob's sequence. The tree restoration process can be implemented in $O(n \log n)$ time, if we store all numbers that don't occur in any sequence in priority queue.

Problem G. Game Relics

Author: Niyaz Nigmatullin
Statement and tests: Niyaz Nigmatullin



solution	team	att	time	size	lang
Fastest	SPbSU: 25	1	66	1825	C++
Shortest	MIPT: Fennecs	3	167	1453	C++
Max atts.	NN SU: Retired	4	287	3578	C++

When k relics are bought, the expected number of shards spent for buying one more relic by keeping paying x shards for random relic is $\left(\frac{n}{n-k}+1\right)\cdot\frac{x}{2}$.

The optimal strategy is to never random after buying any relic for its price. It means that when it's optimal to buy a relic for its price, we can buy the remaining relics in arbitrary order. So, the way of buying relics can be changed:

1. Gloria chooses either she wants to buy for its price, or she wants to keep getting random relic for x shards.
2. Then if Gloria chooses the former, she will buy a random relic for its price that she doesn't own yet.
3. She goes to step 1.

So the probability that at some point Gloria will own a certain subset of relics is equal for all subsets of k relics, and it equals to $\frac{1}{\binom{n}{k}}$.

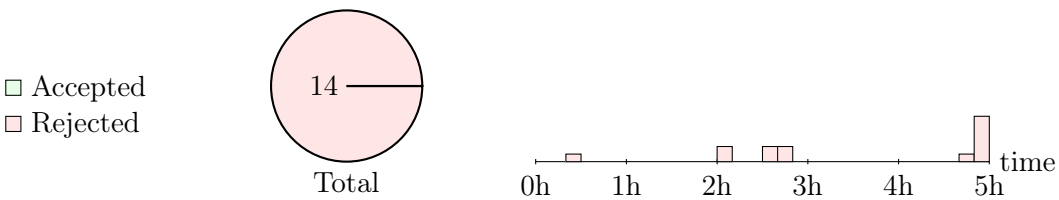
Let's say there are b relics that Gloria doesn't own yet with costs $a_1, a_2, \ldots, a_b$. It turns out that if $\frac{\sum a_i}{b} \le \left(\frac{n}{b}+1\right)\cdot\frac{x}{2}$ then Gloria should buy a random relic for its cost with expected cost $\frac{\sum a_i}{b}$, otherwise Gloria should buy a random relic for expected cost $\left(\frac{n}{b}+1\right)\cdot\frac{x}{2}$.

So after coming up with the facts above, the solution is to calculate $f_{s,k}$ — the number of item subsets of size k with total sum of item prices s. This function can be calculated as in knapsack problem.

The answer is: $\sum \frac{f_{s,k}}{\binom{n}{k}} \cdot \min\left(\frac{S-s}{n-k}, \left(\frac{n}{n-k}+1\right)\cdot\frac{x}{2}\right)$, where $S = \sum c_i$.

Problem H. Help BerLine

Author: Mikhail Mirzayanov
Statement and tests: Mikhail Mirzayanov, Borys Minaiev



First, let's constructively prove that three frequencies are enough for up to four base stations.

1) For each of the (chronologically) first three base stations, use a new frequency that wasn't used before. So far all frequencies are unique, thus all nonempty subsegments are fine.

2) For the (chronologically) fourth base station, use a frequency that is not present among its neighbour(s). Without loss of generality, before its turning on, the other three base stations had frequencies $[1, 2, 3]$. Thus, after turning on the fourth one, we will have $[+2, 1, 2, 3]$, $[1, +3, 2, 3]$, $[1, 2, +1, 3]$, or $[1, 2, 3, +1]$, which are all fine.

Now, having solved $n \le 4$ with three frequencies, let's solve $n > 4$. We will reduce this problem to the problem of size $\lceil \frac{n}{3} \rceil$ using three new frequencies. This will allow us to solve the problem of size $n \le 4 \cdot 3^k$ using $3k+3$ frequencies. Thus 24 frequencies will be enough to solve the problem of size $n \le 4 \cdot 3^7 = 8748$.

Divide the n base stations into consecutive triples (geographically, not chronologically). The last one might be less than a triple, there should be no problematic special case there. In each triple mark the base station that will be turned on earlier than the two other ones.

On these $\lceil \frac{n}{3} \rceil$ marked base stations, solve the problem recursively. We now need to solve the entire problem of size n using three more frequencies.

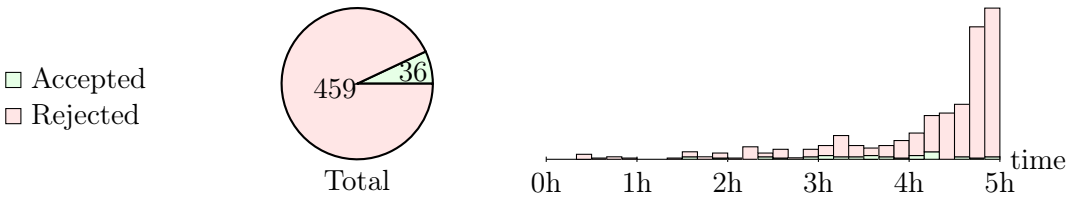
Let's turn on all n base stations in the chronological order. If the considered base station is marked, then use the frequency obtained from the recursive solution. Otherwise, we need to use one of the three new frequencies.

Let's evaluate, how many unmarked turned on base stations are there (including the considered one) strictly between the closest marked and turned on station to its left and the closest marked and turned on station to its right. The maximum value is 4, which is reached in the following situation: [Marked and turned on b. s., unmarked and turned on b. s., unmarked and turned on b. s.], [Turned off triple], [Turned off triple], ..., [Turned off triple], [Unmarked and turned on b. s., unmarked and turned on b. s., marked and turned on b. s.]. Thus in order to select the frequency for the considered base station, we can use the algorithm for $n \le 4$.

Let's prove that after this frequency assignment all subsegments of turned on base stations are fine. If a subsegment contains at least one marked base station, consider all marked base stations it contains. It is a non-empty subsegment in terms of the recursive subproblem, and since the recursive solution is correct for any moment of time, there is a unique frequency on this subsegment. Otherwise, if there is no marked base stations in this subsegment, then we're inside a local problem of size at most 4, the correctness of solution for which was shown above.

Problem I. Intriguing Selection

Author: Petr Mitrichev
Statement and tests: Petr Mitrichev



solution	team	att	time	size	lang
Fastest	NN SU: Retired	1	97	1885	C++
Shortest	SPbITMO: 4	1	187	1235	C++
Max atts.	SPbSU: Havka	13	254	3633	C++

We are aware of multiple working approaches for this problem, here is the simplest one: let us take arbitrary $n+1$ players and split them arbitrarily into two groups of size at least 2, for example of size 2

and $n - 1$.

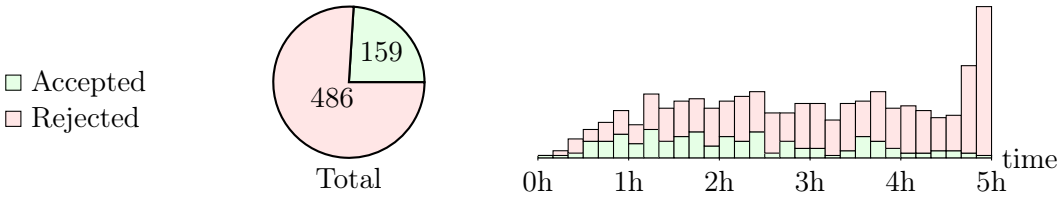
Now let's find the player with the smallest strength in each group in any possible way (using only comparisons within the group), and then compare those two players between themselves. The one which compares smaller is the player with the smallest strength among the $n + 1$ chosen players, and therefore is not among the n players with the highest strength, so we can discard them from consideration.

Now let's add one more player to one of the two groups in such a way that both have size at least 2, and repeat the step above, discarding one more player from consideration. We repeat this until there are no more players to add (discarding n players in total).

In the end we're left with the n players with the highest strength split into two groups, and we have never compared any player from the first group to any player of the second group, therefore there are at least two (more precisely, at least three) possible orderings of those n players.

Problem J. Just Arrange the Icons

Author: Mikhail Mirzayanov
Statement and tests: Mikhail Mirzayanov, Pavel Kunyavsky



solution	team	att	time	size	lang
Fastest	IntITU: 1	1	9	1024	C++
Shortest	BurSU: 4	2	191	836	C++
Max atts.	Astana ITU: 1	13	260	1898	C++

Note that we don't need the colors of the icons, but only the number of icons for each color is needed. Let's count these numbers, ignoring colors that don't appear in the input. Also, let's sort these numbers in order of non-decreasing. Consider, $f = [f_1, f_2, \ldots, f_k]$ is the resulting sequence ($0 < f_1 \le f_2 \le \cdots \le f_k$), where k is number of colors which appear in the input and f_j is number of icons for some color.

Let's iterate over all possible screen sizes: from 1 to $f_1 + 1$. Consider the current screen size is s. Let's count the total number of screens of size s needed to fit all the icons.

Obviously, for fixed s for each color the number of screens can be calculated independently and the required total number of screens is just a sum of the number of screens for each color.

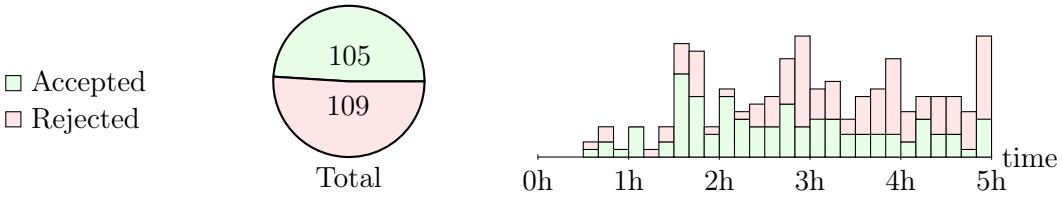
Consider, f_j is the number of icons with some color and s is the screen size. Let's find the number of screens to fit all of them or report that it is impossible to do. The total number of screens to fit f_j icons is $q_j = \lceil f_j / s \rceil$ is the solution exists. On q_j full or almost full screens we can place at least $q_j \cdot (s - 1)$ icons. So if $f_j < q_j \cdot (s - 1)$ the size s is unsuitable to fit f_j icons in the required way (and the screen size s should be discarded). If s is suitable for all the numbers $f_1, f_2, \ldots, f_k$ then the total number of screens is $q_1 + q_2 + \cdots + q_k$.

Print the minimal value over all possible screen sizes s from 1 to $f_1 + 1$.

The number of operations of the main part of the solution is $O(f_1 \cdot k)$, but $f_1 + f_1 + \cdots + f_1 \le f_1 + f_2 + \cdots + f_k = n$. So, the actual compexity of the main part is $O(n)$.

Problem K. Key Storage

Author: Elena Kryuchkova
Statement and tests: Pavel Kunyavsky



solution	team	att	time	size	lang
Fastest	SPbSU: 25	1	36	1273	C++
Shortest	BurSU: 4	2	245	938	C++
Max atts.	Ulyanovsk STU: 1	7	291	2257	C++

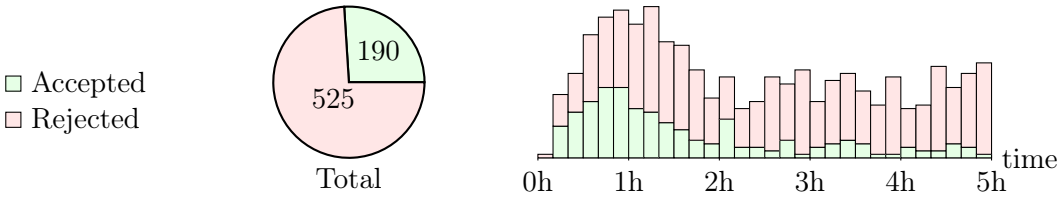
Let's calculate fingerprint by definition. Key can be uniquely reconstructed if we fix order of remainders, so we can calculate number of valid orders instead. Order is valid, if it's not finishes with zero, and each remainder is strictly less than divider.

Let n be size of fingerprint. For the first try let's forget about "not finish with zero" restriction. To calculate number of orders, let's choose position for remainder in decreasing order. For each value v, there is set of valid positions, which size is equal to $n - max(0, v - 1)$. Also, order of same values doesn't matter, so, answer should be divided by number of equal remainders, which was already set, including this one. All already chosen positions are in this set. So if $f_0, f_1, \ldots, f_{n-1}$ is fingerprint sorted in non-increasing order, answer is $\prod_{i=0}^{n-1} \frac{\max(0, n - \max(0, f_i - 1) - i)}{\texttt{number of } j \le i\texttt{, such that } f_i = f_j}$.

To handle "not finish with zero" restriction, one can calculate number of fingerprints, which finishes with zero, but valid for all other conditions, and subtract them. This number is equal to number of valid fingerprints, with multi-set after removing one zero from original one.

Problem L. Lexicography

Author: Georgiy Korneev
Statement and tests: Georgiy Korneev



solution	team	att	time	size	lang
Fastest	SPbSU: Quick Burg	1	11	965	C++
Shortest	BurSU: 4	2	55	661	C++
Max atts.	ChelSU: admin	28	288	1459	Python

Let's count the number of times each letter occurs in the input: c_p for p from '`a`' to '`z`'.

We will construct the answer letter-by-letter while maintaining the value t — the index of the first word that has the same prefix as the k-th word. Initially, all words are empty, so $t = 1$.

The solution contains two phases.

On the first phase, let's consider the minimal available letter p. We have c_p instances of this letter. There are two possible cases:

- $c_p > k - t$. In this case, we will add a single letter p to all words from t to k (inclusively) and solve the same problem for $c_p := c_p - (k - t + 1)$.
- $c_p \le k - t$. In this case, we will add single letter p to c_p words, starting from t-th word, update $t := t + c_p$, and move to the next available letter.

The first phase stops when k-th word contains l letters.

On the second phase, some words before k-th may have less than l letters, and all words after k-th have no letters at all. The distribution of the remaining letters does not affect the order of the first k words, so we may arrange them in any way, for example, just appending each word until it has l letters.

We build a non-decreasing sequence of words w_i ($i = 1..n$), let's proof that w_k is the minimal possible word. If it is false, we consider a counterexample: W_i — non-decreasing sequence of n words, build from the same letters, where $w_k > W_k$. Let's consider the first difference in the construction order: $w_{ij} \ne W_{ij}$. There are two possibilities:

- $w_{ij} > W_{ij}$. In this case, we have available letter W_{ij}, and do not use it. This is not possible, as we append all letters in the lexicographical order, withouth skips.
- $w_{ij} < W_{ij}$. In this case $w_i < W_i$ and we "saved" letter w_{ij}. Let's consider the place where this letter is used $W_{i'j'} = w_{ij}$ and $w_{i'j'} \ne w_{ij}$. As we using letter in the lexicographical order $W_{i'j'} < w_{ij}$, so if we replace $W_{i'j'}$ by $w_{i'j'}$, the word $W_{i'}$ will either keep it's place, or will be moved towards the head of the list. In the first case, the only change is that $w_i < W_i$, so we either $w_k < W_k$, or we have an example where first difference comes later than at (i, j) and we will iterate the proof. In the second case