Problem A. Bank Transfer

Problem author and developer: Andrew Stankevich

This is the easiest problem of the contest, it can be solved with simple formula:

```
int k;
cin >> k;
double v = 25.0 + k / 100.0;
if (v < 100) {
    v = 100;
}
if (v > 2000) {
    v = 2000;
}
cout.precision(20);
cout << fixed << v << endl;
```

It is possible to avoid using floating point numbers:

```
int x;
scanf("%d", &x);
x += 2500;
x = max(x, 10000);
x = min(x, 200000);
printf("%d.%02d\n", x / 100, x % 100);
```

Problem B. Bacteria

Author: Ildar Gainullin

Developer: Ivan Volkov

First of all, let's note that the DNA of the bacteria obtained at the i-th step is simply the results of splitting the DNA of the original bacterium into2^{i-1} equal parts. So knowing (with a fixed order) DNA of all bacteria on the i-th step, it is easy to restore bacterial DNA in subsequent and previous steps, in particular, the DNA of the original bacterium.

Next, let's note that at each step we have two constraints on the number of different DNA: on the one hand, it is not more than total number of bacteria on this step (that is, 2^{i-1}), and on the other hand it is not more than number of different binary strings of appropriate length (that is, not more than 2^{len_i}, where $len_i = 2^{n-i+1}$). It is easy to see that first restriction is getting weaker at each step, and the second, on the contrary, becomes only stronger.

To find an answer, let's look at the first step at which $2^{len_i} \le 2^{i-1}$, that is, at which the number of bacteria is not less than the total number of possible DNA (i.e, number of possible binary strings of length len_i). Then we just need to "arrange" the DNA of the bacteria at this step so that two conditions will be satisfied. First is that there are all the different binary strings of length len_i among DNA of bacteria on i-th step. Second is that among the bacteria at the $i-1$-st step (we can easily restore them), all the DNA is pairwise different. In this case, at all steps before i-th, we reach the maximum because the DNA of all bacteria are pairwise different. And on i-th and next steps, we reach the maximum because we reach the limit of different possible DNA of the corresponding length.

But arranging the DNA in the right way (when we already found the proper step) is quite a technical task. You can, for example, first "give" all different DNA to the bacteria on the proper step, and then go to the previous step and "arrange" those DNA were not been used yet.

Problem C. Check Markers

Problem author and developer: Nikolay Kalinin

Consider a situation when Alexander Markovich does not find two good markers of different colors. Then all markers, except for those that remain in the heap, can be divided into pairs of markers of different colors in such a way that at least of the markers in each pair is bad. Also note that if at some point among the *good* markers there are only markers of the same color x, then the professor will not be able to start the lecture regardless of further actions. This means that we can "return" to the heap all such pairs of markers in which both markers are bad, this will not help the professor. However, this allows us to simplify the criterion for a possible disruption of the lecture: if we can select several pairs of markers of different colors, where in each pair *exactly* one marker is bad, in such a way that among good markers in the heap only markers of color x will remain, then the answer is "`YES`".

Let's formulate the criterion in a different way: if it is possible to match each good marker, except for the markers of color x, to some bad marker of a different color (and all such bad markers should be different), then it is possible for the lecture to fail. Imagine a bipartite graph in which each vertex of the left part corresponds to a good marker, and each vertex of the right part corresponds to a bad one. Let's draw edges between vertices from different parts if these are markers of different colors (that is, the professor can take them simultaneously). Then the criterion looks like this: if after removing all the vertices of the left part corresponding to the markers of color x, some matching can cover all the vertices of the left part (all good markers), then it may happen that at the end among good markers only markers of color x will remain. Thus, if this criterion is satisfied for at least one color x, then the answer is "`YES`".

To solve the problem, it remains to learn how to check the criterion for a given color x. Since the matching must cover all the vertices of the left part, we can use Hall's lemma: a matching that covers all the vertices of the left part exists if and only if, for any subset of vertices A from the left part, the union of the sets of their neighbors in the right part contains at least as many vertices as there are in the set A. Now note that in this problem it is possible to check not all subsets of A, but only some of them. Indeed, let the criterion be satisfied for the set of all vertices of the left part (that is, there are no less bad markers than good ones). Then for any set A that includes at least two markers of different colors the set of their neighbors contains all bad markers, and there are definitely no less of them than markers in A. If A contains only markers of the same color, then the set of their neighbors does not depend on their number, and it is sufficient to check only the set of all markers of the same color.

Thus, for a fixed color x, it suffices to check that:

- the number of good markers, except those of color x, is not greater than the number of bad ones: $(\sum b_i) - b_x \le \sum a_i$;
- for any other color y there are no more good markers of this color than bad markers of other colors: $b_y \le (\sum a_i) - a_y$.

The criterion for the color y does not depend on the choice of the color x, so all of them can be checked in advance. Then, when iterating over the color x, you can check both criteria in $O(1)$. The total complexity of the solution is $O(n)$.

Problem D. Multiple Subject Lessons

Problem author and developer: Andrew Stankevich

Consider the algorithm of calculating the number of partitions. We would consider less efficient algorithm that is running in $O(n^3)$.

Denote the number of partitions of n into terms that do not exceed p as $d[n][p]$. To get a reccurence,

iterate over the number of terms equal to p exactly, let their number be equal to t. Then

$$d[n][p] = \sum_{t=0}^{\lfloor n/p \rfloor} d[n - tp][p - 1].$$

Now we need to choose colors for these t terms. For each of the t terms we must choose one of k colors, order doesn't matter. The number of ways to do it is the number of combinations with repetition, from k choose t, which is equal to $\binom{k+t-1}{t}$.

To calculate the number of combinations one can use, for example, Pascal's triangle, or use factorials formula.

So we have a dynamic programming solution. Denote as $a[n][p]$ the number of k-colored partitions of n to terms not exceeding p. Then

$$a[n][p] = \sum_{t=0}^{\lfloor n/p \rfloor} \binom{k+t-1}{t} \cdot a[n - tp][p - 1].$$

Problem E. Prank at IKEA

Problem author and developer: Nikolay Budin

Let's call two additional cells that would be occupied by an unfolded couch *needed cells* for this couch.

We will consider only couches that could potentially be unfolded, i.e. such couches that both of their needed cells are empty. Let's call such couches *interesting*.

Let's build a graph. Nodes of this graph correspond to interesting couches. Two nodes are connected by an edge if the corresponding two couches conflict with each other, i.e. their sets of needed cells are intersecting. Notice, that we have to find the maximum independent set in the constructed graph.

Notice that each empty cell could be needed for no more than 2 interesting couches. Suppose it's needed for at least 3 interesting couches. Then there are two couches at opposite sides of the cell. Without loss of generality, they are to the left and to the right of the cell. Then a couch to the top from the cell can't be unfolded downwards, nor a couch to the bottom from the cell can be unfolded upwards. That's because at least one of its needed cells would be occupied by a couch.

Each interesting couch has exactly two needed cells and each cell is needed for no more than two interesting couches. So, each couch conflicts with no more than two other interesting couches. Thus, the degree of each node in the built graph is no more than 2. Therefore, every connected component in the graph is either a simple path or a simple cycle. Finding the maximum independent set is trivial in both cases. We should take alternating nodes — take one, skip one, take one, and so on.

One could prove several more properties of the built graph. For example, there could only be cycles of length 2 or 4. But it's not needed to solve the problem.

The complexity of the solution is $O(n \cdot m)$.

Problem F. SMS from MCHS

Author: Andrew Stankevich

Developer: Mikhail Anoprenko

Just check the described conditions and print the required message. The following program solves the problem:

```
t1, v1 = map(int, input().split())
t2, v2 = map(int, input().split())

if t2 < 0 and v2 >= 10:
    print('A storm warning for tomorrow!')
    print('Be careful and stay home if possible!')
elif t2 < t1:
    print('MCHS warns! Low temperature is expected tomorrow.')
elif v2 > v1:
    print('MCHS warns! Strong wind is expected tomorrow.')
else:
    print('No message')
```

Problem G. Cooking

Author: Ildar Gainullin

Developer: Sergey Khargelia

Firstly, the answer is -1 if and only if $\sum_{i=1}^{n} a_i$ is odd (in this case it's impossible to split dishes into pairs).

Let's assume that the answer is not -1 and it's optimal to cook the following pairs of dishes: $(a_1, b_1), (a_2, b_2), \ldots, (a_m, b_m)$. Then, we can choose the order of dishes in each pair such that i-th dish was first in its pairs $\lceil \frac{a_i}{2} \rceil$ or $\lfloor \frac{a_i}{2} \rfloor$ times.

This fact is equivalent to the following: consider undirected graph with n vertices and edges $\{(a_1, b_1), (a_2, b_2), \ldots, (a_m, b_m)\} \setminus \{(a_i, b_i) : a_i = b_i\}$. It's possible to choose directions of the edges such that $\forall v : |indeg(v) - outdeg(v)| \leq 1$. The number of vertices with the odd degree is always even, let's say that the vertices $v_1, v_2, \ldots, v_{2k}$ have the odd degree. Then we can add to the graph dummy edges $(v_1, v_2), (v_3, v_4), \ldots, (v_{2k-1}, v_{2k})$, in the resulting graph all vertices have even degree $\Rightarrow$ it can be splited into cycles (it follows from the algorithm of constructing Euler cycle). Then we can transform each undirected cycle into a directed one, now $\forall v : indeg(v) = outdeg(v)$. Each vertex is incedent to no more than one dummy edge, so after removal of the dummy edges $\forall v : |indeg(v) - outdeg(v)| \leq 1$ will be true.

Let's define b_i as the number of times that i-th dish was first in its pairs. For each b_i we can choose its value from $\{\lceil \frac{a_i}{2} \rceil, \lfloor \frac{a_i}{2} \rfloor\}$. Consider the bipartie graph where each dish corresponds to two vertices — this dish is first/second in the ordered pairs. Let's add dummy vertices S and T and edges $(i, 1) \to (j, 2)$ with the weight $c_{i,j}$ and the capacity $+\infty$, $S \to (i, 1)$ with the weight 0 and the capacity b_i and $(i, 2) \to T$ with the weight 0 and the capacity $a_i - b_i$.

Then, for a fixed set of b_i optimal answer is equal to the minimum cost of the maximum flow from S to T and the total answer is the minimum value over all suitable sets. Note that only sets with maximum flow equal to $\frac{1}{2} \sum_{i=1}^{n} a_i$ are suitable.

Maximum flow with minimum cost can be found using Dijkstra algorithm with potentials. There are $O(n^2)$ edges, also let's define U as $\sum_{i=1}^{n} a_i$. Then for a fixed set of b_i time complexity is $O(n^3 + Un^2) = O(n^3 \max a_i)$. There are no more than 2^n possible sets, so total time complexity is $O(2^n n^3 \max a_i)$.

Problem H. Hard Work

Author: Andrew Stankevich

Developer: Daniel Oreshnikov

To solve this problem let's divide the numbers between l and r into segments, each of which has the form "from $\overline{\texttt{pref}0\ldots0}$ to $\overline{\texttt{pref}9\ldots9}$". In particular we'll denote the segment from 3400 to 3499 as `34**`.

For an example let us consider $l = 3378$ and $r = 3621$. The segment from l to r can be divided into the following segments: `3378`, `3379`, `338*`, `339*`, `34**`, `35**`, `360*`, `361*`, `3620`, `3621`.

Let's say that l and r consist of the same number of digits (otherwise we can fill l with leading zeros). Note that (as seen in the example above) at first the number of "`*`" in segments' descriptions increases and then decreases. We will the build such segments one by one:

- if the current segment starts at x then `pref` is the prefix of x obtained by dropping trailing zeros from x
- if $y = \overline{\texttt{pref}9\ldots9} \leqslant r$ we add a segment `pref*...*` and go to the next segments with $x = y + 1$
- otherwise we are at the second phase where the amount of asterisks decreases: let's recursively repeat the second step for prefixes $\overline{\texttt{pref}0}, \overline{\texttt{pref}1}, \ldots, \overline{\texttt{pref}9}$

There are no more than $2 \cdot 10 \cdot 18 < 400$ such segments in total, because in the part with non-decreasing number of asterisks in each successive segment either *1.* the number of stars is the same as in the previous segment but the last digit in prefix is greater or *2.* the number of stars is greater. Symmetrically almost the same is true for the part with non-decreasing number of asterisks. In the same time there are no more than 18 transitions of the second type because that is the maximum length of the number and no more than 9 transitions of the first type in each digit because no digit can be increased by 10.

For each segment it is not difficult to find the answer independently: just consider two cases:

1. the answer is in a fixed prefix: then it is enough to iterate through the prefix and find the maximal sequence of consecutive identical digits (and the quantity of numbers with such a sequence will be equal to the length of the segment)
2. the answer is in the "tail": then the maximum is reached when the tail consists of digits equal to the last digit in the fixed prefix. The length of the answer can be calculated by iterating through the end of the prefix and the quantity of satisfying numbers is 1

After that we have to look at the answers in each segment and sum the count of satisfying numbers in the segments where the answer is maximal.

Problem I. Points and Segments

Author: Andrey Chulkov

Developer: Andrew Stankevich

Surprisingly it turns out, that the moves that the players make are not important. The number of moves until the game is over is always the same.

To prove it first recall two facts from geometry:

1. Euler's formula. For every connected planar graph embedded into a planethe number of faces F, the number of vertices V and the number of edges E satisfy the equality $F + V - E = 2$.
2. Every polygon that has more than three vertices has an internal diagonal.

Consider a situation when the player cannot make a move. Construct the graph: take original points as vertices and the segments drawn as edges. Since the segments do not intersect, the graph is planar and embedded into plane. All the faces are triangles, because in the other case it is possible to make a move by drawing a diagonal of a non-triangle face.

Therefore the number of faces and the number of edges satisfy two linear equations: $F + V - E = 2$ and $3(F-1)+C = 2E$, here C is the number of edge of the external face, it is equal to the number of vertices of the convex hull of the given points.

From the first equation $F = E + 2 - V$, substitute it to the second one and get $E = 3V - 3 - C$. So the number of edges is actually always the same.

If follows that in order to solve the problem you should only find the number of edges and consider its parity. If it's odd the first player would win, if it's even the second player would win.

The only remaining problem is to find any valid move. To do it one way is to store a set of all possible moves. Moves are only removed from the set, so to make a move, iterate over the set. If the move is invalid, remove it. If the move is valid, make it and remove it.

The complexity of the solution is $O(n^3)$, because there are $O(n^2)$ possible segments that we store, and each of them is checked for intersection with $O(n)$ segments from actual moves.

Problem J. Straight

Author: Niyaz Nigmatullin

Developer: Dmitriy Gnatyuk

Let's sort the community cards and remove the duplicated. Denote the resulting array a.

Let's iterate over all community cards and calculate for each one, what is the set of possible starts of the straight in which $a[i]$ is the smallest used community card.

First, for this card to be the smallest used one, we can't touch the previous card, so the smallest card of the straight should be at least $a[i-1]+1$, and in case of $i = 1$ it should just be at least 1.

Second, note that at least $m - s$ community cards must be used. Therefore, cards $a[i], a[i+1], \ldots, a[i+m-s-1]$ should exist and fit into the straight. So if element $a[i+m-s-1]$ doesn't exist, we just skip this i. Otherwise, the largest card of the straight must be at least $a[i+m-s-1]$. Thus, the smallest card of the straight must be at least $a[i+m-s-1]-m+1$.

Third, the smallest card of the straight must be at most $a[i]$, so that $a[i]$ could be used in the straight.

Finally, the smallest card of the straight must be at most $n-m+1$, since the largest card of the straight should exist and therefore be at most n.

This way, we have found the lower and the upper bound for the start of the straight. If this segment is empty, skip this i. Now, the answer to the problem is the sum of the lengths of all such segments over all i.

Problem K. New Level

Author and developer: Alexander Morozov

For the given colouring, leave only edges of the given graph that connect edges with adjacent colours.

Consider the following algorithm:

1. If the graph is connected, then the current colouring is the answer, you can finish the algorithm.
2. Otherwise, increase colours of all vertices in the connected component of the first vertex by 1 (modulo k). Return to (1).

Lemma: Current colouring is always proper.

Proof: If there is an edge (u, v) with $c_u = c_v$, then on the previous iteration (before the last increase), colours c_u and c_v were adjacent, so these vertices should've been in the same connected component, so at the next iteration their colours should increase simultaneously. Contradiction.

Lemma: This algorithm will finish after the finite number of iterations.

Proof: Note that after k additions, a new edge from the connected component of the first vertex will be added to the graph. It means that after $\geq k(n-1)$ additions graph should be connected.

To optimize this algorithm, we can store all edges outgoing from the connected component of the first vertex in `std::set`, with priorities equal to the differences between colours. Each time you can find the minimum time at which a new edge will be added to the graph, and then you need to add all minimums from the set to the DSU.

The total complexity is equal to $\mathcal{O}((n+m)\log(n+m))$

Problem L. The Firm Knapsack Problem

Author and developer: Sergey Kopeliovich

Let's split our items into two groups – small items ($w_i \leq \frac{W}{2}$) and big items ($w_i > \frac{W}{2}$).

In optimal solution to Knapsack problem (we denote this optimal solution as *opt*) there is at most one big item (more no longer fit into W). Lets iterate big item, which we are going to take (or just not take any big item). We denote index of this item as i.

For each i we will fill our knapsack in following way. Firstly we take i-th item, then we take small items greedily in descending order of $\frac{cost_j}{w_j}$.

To make working time of this approach $O(n \log n)$, we iterate big items in order of descending w_i, and use two-pointers-method. When w_i decreases, amount of small items, which fit into knapsack increases, that's the second pointer.

Proof of correctness.

Let we guess i correctly (big item in *opt*). In *opt* there are no other big items. Consider first moment, when out greedy do not takes some small item $j \in opt$. Lets denote sum of weights of already taken items as W_{cur}. We know that $w_j \leq \frac{W}{2}$ (small) and $W_{cur} + w_j > \frac{3}{2}W$ (does not fit), so $W_{cur} > W$ so cost of our solution is already strictly greater than cost of *opt* (we have taken all items from *opt* with average cost more than $\frac{cost_j}{w_j}$ and some items with average cost not less, out summary weight is more than W).