Problem A. Armor and Weapons

Author: Ivan Androsov

Preparation: Ivan Androsov

Among two armor sets, one with the greater index is always better. The same can be said about two different weapons. So, it is always optimal to use and obtain the best possible weapon or armor.

This observation allows us to model this problem with dynamic programming or shortest paths: let $dp_{x,y}$ be the minimum time in which Monocarp can obtain the armor x and the weapon y; and in each transition, we either get the best weapon we can or the best armor we can. Similarly, we can build a graph where the vertices represent these pairs (x, y), and the edges represent getting the best possible weapon/armor, and find the shortest path from $(1, 1)$ to (n, m) using BFS.

Unfortunately, it is $O(nm)$. But we can modify the BFS in the following fashion: let's analyze each layer of BFS (a layer is a set of vertices with the same distance from the origin). In each layer, there might be some redundant vertices: if two vertices (x, y) and (x', y') belong to the same layer, $x' \le x$ and $y' \le y$, then the vertex (x', y') is redundant.

If we filter each layer, removing all redundant vertices from it and continuing BFS only from non-redundant ones, the solution will be fast enough. To prove it, let's analyze the constraints on the answer. Suppose $n \ge m$. The answer can be bounded as $O(\log m + \frac{n}{m})$, since we can reach the pair (m, m) in $O(\log m)$ steps using something similar to Fibonacci sequence building, and then go from (m, m) to (n, m) in $\frac{n}{m}$ steps. And the number of non-redundant states on each layer is not greater than m (because, of two states with the same weapon or the same armor set, at least one is redundant). So, if we don't continue BFS from redundant vertices, it will visit at most $O(m \cdot (\log m + \frac{n}{m})) = O(m \log m + n)$ vertices. There might be another logarithm in the asymptotic complexity of the solution, if you use something like a set to store all combinations that synergize well, but this implementation is still fast enough.

Problem B. Special Permutation

Author: Mike Mirzayanov

Preparation: Mike Mirzayanov

There are many different constructions that give the correct answer, if it exists. In my opinion, one of the most elegant is the following one.

a should always be present in the left half, and b should be present in the right half, but the exact order of elements in each half doesn't matter. So, it will never be wrong to put a in the first position, and b in the second position.

As for the remaining elements, we want elements of the left half to be as big as possible (since they shouldn't be less than a), and elements of the right half — as small as possible (since they shouldn't be greater than b). Let's put the elements n, $n-1$, $n-2$, …, 1 (excluding a and b) on positions 2, 3, 4, …, $n-1$, respectively, so the elements in the left half are as big as possible, and the elements in the right half are as small as possible.

After constructing a permutation according to these rules, we should check if it meets the constraints (and print it if it does).

Problem C. Athletes

Author: Mike Mirzayanov

Preparation: Mike Mirzayanov

Suppose we want to include exactly k athletes from sport A and exactly k athletes from sport B (it's not always possible, but nevertheless). Obviously, each athlete should compete in their own sport. It's also quite obvious that we need to choose k best athletes from sport A and k best athletes from sport B, so let's sort the athletes in each sport in non-ascending order of their skills, and pick k first values from each of the two obtained lists of athletes.

It's not always optimal (and even not always possible) to choose exactly k athletes from each sport. So, let's consider that we choose z athletes from sport A and $2k - z$ athletes from sport B. In each of these cases, it's suboptimal to force an athlete of sport A to compete in B and an athlete of sport B to compete in A at the same time. So, there are three cases:

- $z < k$: we should pick z best athletes from sport A, $2k - z$ best athletes from sport B, and force $k - z$ athletes of sport B to compete in A, so the total skill decreases by $y(k - z)$;
- $z = k$: already discussed in the beginning of the editorial;
- $z > k$: we should pick z best athletes from sport A, $2k - z$ best athletes from sport B, and force $z - k$ athletes of sport A to compete in B, so the total skill decreases by $x(z - k)$.

We should try all values of z from 0 to $2k$ and find which of them gives the maximum possible sum of skills.

Note: if you sort the athletes and try to get the sum of several best skills for each value of z using a for-loop, your solution will be too slow (it will run in something like $O(kn \log n)$ or $O(k^2 + n \log n)$). Instead, we can sort both lists of athletes beforehand, and then either maintain pointers to the last chosen athletes in both lists to update the sum of skills in $O(1)$ when the value of z increases by 1, or prepare partial sum array for each of the sorted lists to get the sum of several first values without running a loop.

Problem D. Max Sum Array

Author: Adilbek Dalabaev

Preparation: Adilbek Dalabaev

Firstly, let's prove that at first and last positions of a the most frequent elements should be placed (but not necessary the same). WLOG, let's prove that $c_{a_1} \ge c_{a_i}$ for any $i > 1$.

By contradiction, let's p be the smallest index such that $c_{a_1} < c_{a_p}$. What happens if we swap them? Since p is the first such index then there are no $a_i = a_p$ for $i < p$, so "contribution" of a_p will increase by exactly $inc = (c_{a_p} - 1) \cdot (p - 1)$. From the other side, contribution of a_1 consists of two parts: pairs with elements from $(p, n]$ and from $(1, p)$. For all elements from $(p, n]$ decrease will be equal to $dec_1 = c_{a_1}[p+1, n] \cdot (p-1)$ and from elements in $(1, p)$ $dec_2 \le c_{a_1}[2, p-1] \cdot (p-1)$.

So, the total decrease dec after moving a_1 to position p equal to $dec = dec_1 + dec_2 \le (c_{a_1} - 1) \cdot (p - 1)$. The total difference in such case is equal to $inc - dec \ge (p - 1) \cdot (c_{a_p} - c_{a_1}) > 0$. So, our placement is not optimal — contradiction.

Let's suggest that there is exactly one e with maximum c_e. According to what we proved earlier, both a_1 and a_n must be equal to e. Contribution of the first and last elements will be equal to: $(n - 1)$ for pair $(1, n)$ and for each element i $(1 < i < n)$ with $a_i = e$ we add $(i - 1) + (n - i) = (n - 1)$ for pairs $(1, i)$ and (i, n). So, the total contribution of $a_1 = a_n = e$ is equal to $(n - 1) \cdot (c_e - 1)$.

Note that this contribution is independent of positions of other e in the array a, so that's why we can cut first and last elements of a and solve the task recursively.

Unfortunately, in the initial task we may have several $e_1, e_2, \dots, e_k$ with maximum c_{e_i}. But we in the similar manner can prove that the first (and last) k elements $a_1, \dots, a_k$ should be some permutation of $e_1, \dots e_k$. Now, let's prove that any permutation of first and last k elements is optimal.

Suppose, positions of e_i are l_i $(1 \le l_i \le k)$ and r_i $(n-k+1 \le r_i \le n)$. Then the contribution of e_i is equal to $(c_{e_i} - 1) \cdot (r_i - l_i)$. The total contribution of all e_i is $\sum\limits_{i=1}^{k} (c_{e_i} - 1) \cdot (r_i - l_i) = (c_{e_1} - 1)(\sum\limits_{i=1}^{k} r_i - \sum\limits_{i=1}^{k} l_i) = (c_{e_1} - 1)((n-k)k + \frac{k(k+1)}{2} - \frac{k(k+1)}{2}) = (c_{e_1} - 1)k(n-k)$. This contribution doesn't depend on chosen l_i and r_i, so any permutation of first k elements and any permutation of last k elements give optimal answer.

As a result, the algorithm is following:

- Find all maximums $e_1, \ldots, e_k$ in c.
- If $c_{e_1} = 1$ then any permutation of remaining elements has $f(a) = 0$ (there are $k!$ such permutations).
- Otherwise, add $(c_{e_1} - 1)k(n-k)$ to the total balance, and multiply the number of variants by $(k!)^2$.
- Cut prefix and suffix by making $c_{e_i} = c_{e_i} - 2$ for each e_i (obviously, $n = n - 2k$) and repeat the whole process.

We can implement the algorithm fast if we keep the number of c_i equal to each val from 0 to C ($C \le 10^6$). So the total complexity is $O(n + C)$.

Problem E. Request Throttling

Author: Dmitry Klenov

Preparation: Dmitry Klenov

The first step required to solve this problem is to go through a pretty long statement (it seems that the length of the statement is the main reason why this problem was not getting a lot of accepted solutions). The problem asks us to maintain the throttling limits and the number of processed queries for several IPv4 subnets, and update them when a request appears.

Firstly, it is impossible to explicitly store the number of requests for each subnet, since there are about $2^3 3$ different subnets. Instead, we will use an associative data structure (`map` in C++, `dict` in Python, `HashMap` or `TreeMap` in Java, etc.) to store the subnets where at least one request appeared, and if we need to find out the number of requests for some subnet which doesn't appear in our data structure, we know that it is 0.

Secondly, we need to find an easy way to iterate through all parent subnets of a given IP address. One of the solutions to this subproblem is to convert each IP address into an integer using the formula given in the statement, and obtain the i-th parent subnet by, for example, applying bitwise AND with an integer such that its i least significant bits are zeroes, and the next $32 - i$ bits are ones. Then, we can easily check the throttling limits and the number of processed requests for parent subnets of an IP address, and update them if a request passes.

Small implementation note: it is possible to store all addresses and subnet IDs in 32-bit integers, but you have to take care of integer overflow (for example, overflow is processed differently in `int` and unsigned int types in C++, and this can cause some unexpected behavior). One of the ways to handle these overflow issues is to use 64-bit integers instead of 32-bit ones.

Problem F. X-Magic Pair

Author: Vladimir Petrov

Preparation: Vladimir Petrov

This problem has a GCD-based solution.

Firstly, lets' try to solve it naively. Always suppose that $a > b$. If this is not true, let's swap a and b. Firstly, if $b > a - b$, let's do $b := a - b$. Okay, now let's subtract b from a until $b \ge a - b$ again and repeat this algorithm till $a = 0$ or $b = 0$. If, after some step, we get $a = x$ or $b = x$, we are done, and the answer is `YES`. If $a = 0$ or $b = 0$, and we didn't get x then the answer is `NO`.

Okay, we can see that we always subtract the minimum possible b from a and trying to maintain this condition. It can be proven that this algorithm yields all possible integers that are obtainable by any sequence of the operations from the problem statement (either in a or in b).

Now we have to speed up this solution somehow. Obviously, most operations are redundant for us in this particular problem. The first thing is that we can skip all operations till b becomes greater than $a - b$. The number of such operations is $\lfloor \frac{a-b}{2b} \rfloor$. And the second thing is that we can skip all operations till we get x in a. The number of such operations is $\lfloor \frac{a-x}{b} \rfloor$. For simplicity, this part can be also written as $\lfloor \frac{a-x}{2b} \rfloor$. This doesn't affect the time complexity much, but the formula for the final number of operations we can skip will be simpler. This number equals $cnt = max(1, \lfloor\frac{a-max(b,x)}{2b}\rfloor)$ (in fact, we take the minimum between two values written above, because we don't want to skip any of these two cases). So, we can transform the pair (a, b) to the pair $(a - b * cnt, b)$ and continue this algorithm.

There are also simpler approaches using the same idea but in a cooler way.

Time complexity: $O(\log a)$ per test case.

Problem G. Chat Ban

Author: Vladimir Petrov

Preparation: Vladimir Petrov

This is a pretty obvious binary search problem. If we get banned after y messages, we also get banned after $y + 1$, $y + 2$ and so on messages (and vice versa, if we don't get banned after y messages, we also don't get banned after $y - 1$, $y - 2$ and so on messages).

For simplicity, let's split the problem into two parts: when we check if we're getting banned after y messages, let's handle cases $y < k$ and $y \geq k$ separately.

Recall that the sum of the arithmetic progression consisting of integers 1, 2, ..., y is $\frac{y(y+1)}{2}$. Let it be $cnt(y)$.

The first case is pretty simple: the number of emotes we send with y messages when $y < k$ is $\frac{y(y+1)}{2}$ which is $cnt(y)$. So we only need to check if $cnt(y) \geq x$.

The second case is a bit harder but still can be done using arithmetic progression formulas. Firstly, we send all messages for $y \leq k$ (the number of such messages is $cnt(k)$). Then, we need to add $(k - 1) + (k - 2) + \ldots + (y - k)$ messages. This number equals to $cnt(k - 1) - cnt(2k - 1 - y)$ (i.e. we send all messages from $k - 1$ to 1 and subtract messages from 1 to $2k - 1 - y$ from this amount). The final condition is $cnt(k) + cnt(k - 1) - cnt(2k - 1 - y) \geq x$.

Time complexity: $O(\log k)$ per test case.

Problem H. Messages

Author: Ivan Androsov

Preparation: Ivan Androsov

First of all, let's rewrite the answer using expectation linearity. The expected number of students who read their respective messages is equal to $F_1 + F_2 + \cdots + F_n$, where F_i is a random value which is 1 if the i-th student reads the message m_i, and 0 if the i-th student doesn't do it.

Let's analyze the expected value of F_i. Suppose Monocarp pins the messages $c_1, c_2, \ldots, c_t$. There are three cases:

- if $m_i \notin [c_1, c_2, \ldots, c_t]$, then the i-th student won't read the message m_i, so $F_i = 0$;
- if $m_i \in [c_1, c_2, \ldots, c_t]$ and $t \leq k_i$, then the i-th student will definitely read the message m_i, so $F_i = 1$;
- if $m_i \in [c_1, c_2, \ldots, c_t]$ and $t > k_i$, then $F_i = \frac{k_i}{t}$.

If we iterate on the number of messages we pin t, we can calculate the sum of F_i for each message (considering that we pin it), sort all of the messages and pick t best of them. So, we have a solution working in $O(n^2 \log n)$.

The only thing we need to improve this solution sufficiently is the fact that we don't have to consider the case $t > 20$. Since every k_i is not greater than 20, the sum of F_i for a message in the case $t = 20$ is the same as this sum of F_i in the case $t = 21$, but multiplied by the coefficient $\frac{20}{21}$ — and we pick 21 best values, their sum multiplied by $\frac{20}{21}$ is not greater than the sum of 20 best values. The same holds for $t = 22$ and greater.

Problem I. Tetris

Authors: Mike Mirzayanov and Ivan Androsov

Preparation: Ivan Androsov

Let's analyze each row of the resulting field. Each row will contain several pieces which form an ascending sequence; i. e. if the pieces in a row block cells $[L_1, R_1], [L_2, R_2], \ldots, [L_m, R_m]$, the condition $L_1 \le R_1 < L_2 \le R_2 < \cdots < L_m \le R_m]$ must hold. We can remodel this as the following problem: create a graph with n vertices representing pieces, and edges $i \to j$ meaning that $L_j > R_i$. We have to find k vertex-disjoint paths in this graph such that their total cost is maximum possible, and this problem can be solved with minimum cost flows.

Unfortunately, even building this graph is $O(n^2)$, running a minimum cost flow on the network for this graph can be about $O(n^3 k)$, which is too slow. Let's instead focus on the property of this graph: an edges $i \to j$ means that $L_j > R_i$. We can build a flow network with $10^9 + 1$ vertices, where vertex 0 is the source, vertex 10^9 is the sink, and each vertex $i < 10^9$ has an outgoing edge to the vertex $i + 1$ with capacity k. To handle tetris piece j, we create an edge from L_j to $R_j + 1$ with capacity of 1 and cost of $-c_j$. Finding a minimum cost k-flow in this network allows us to calculate the absolute value of the maximum total cost of the chosen pieces (each unit of flow will go through edges representing the tetris pieces that can appear in the same row, and there will be only k such "paths"). To get rid of the insane number of vertices, we can see that if a vertex i only has an ingoing edge from $i - 1$ and an outgoing edge to $i + 1$, we can get rid of it, directing the edge from $i - 1$ to $i + 1$. So, we only have to consider the vertices of the form L_j and $R_j + 1$ in our network.

The size of the network will be $O(n)$, so the implementation of minimum cost flow with Ford-Bellman algorithm works in $O(n^2 k)$.

Problem J. Bongcloud Opening

Author: Ivan Androsov

Preparation: Ivan Androsov

There are two solutions to this problem, both utilize a knapsack-like dynamic programming.

Solution 1. Implement a dynamic programming of the form $dp[0..n][-k..k]$, where $dp[i][j]$ is true if and only if Hikaru can play i first matches so that his resulting rating is $x + j$. Transition from each state is choosing which opening Hikaru will use in the i-th match, and after that, modeling the next games in $O(n)$ until the rating is again in $[x - k, x + k]$. Don't forget that the answer should be updated from all states, not only from $dp[n][j]$!

Solution 2. Just implement a dynamic programming of the form: $dp[i]$ is the set of all possible ratings Hikaru can have after i first games. The size of each set will be $O(nk)$ since each number in the set i represents some state of the form "Hikaru had rating y after the game j", where $j < i$ and $y \in [x-k, x+k]$. So, the runtime of this solution will be $O(n^2 k \log(nk))$ or $O(n^2 k)$ depending on whether you use a set based on a binary search tree or a hash table.

Problem K. Ice Cream Van

Author: Elena Rogacheva

Preparation: Mikhail Piklyaev

The main idea of the solution to this problem is to iterate on x from 0 to infinity (obviously, considering only values of the form a_i and $a_i - 1$) and maintain the following graph: from the vertex $i \le n$, there will be an outgoing edge to the vertex $i + 1$ if the i-th type of ice cream is mediocre, or to the vertex $i + k_i$ if the i-th type of ice cream is tasty. This graph is a combination of rooted trees, and we have to maintain the sum on the path leading from the vertex 1 when we have to change edges due to increasing of the value x.

There are two data structures which can maintain this graph. One of them is the Link-Cut Tree, which is kind of overkill for this problem (although, if you have an implementation of Link-Cut Tree in your team reference, you can use it straightforwardly). The second data structure is square root decomposition, and I will explain how to use it a bit more in detail.

Divide all n vertices into blocks of size m, the first block should contain vertices $[1, m]$, the second block — $[m + 1, 2m]$, and so on. For each vertex in each block, calculate and maintain two values:

- to_i — the first vertex on the path from i which does not belong to the same block as i;
- $cost_i$ — the sum on the path from i to to_i.

Since the number of blocks is $\frac{n}{m}$, we can use the calculated values of to_i and $cost_i$ to compute the sum on the path leading from 1 in $O(\frac{n}{m})$. To redirect an edge leading from the vertex i, we can utilize the fact that only the values in the block where the vertex i belongs can be changed, so we can recalculate this block in $O(m)$. So, each change of the value x is processed in $O(\frac{n}{m} + m)$, and by choosing m close to $\sqrt{n}$, we can solve the problem in $O(n\sqrt{n})$.

Problem L. Smash the Trash

Author: Elena Rogacheva

Preparation: Dmitry Stepanenko

Obviously, if k workers can do the job, $k + 1$ workers can do it as well. So the main idea of the solution is to use binary search on the answer, finding the minimum required number of people in $\log 2 \cdot 10^5$ queries of the form "can k people clean everything?"

Let's analyze to process each query of this form in $O(n)$. We will iterate on the places where the cleaning must be done, and each day we will try to dispose of as much trash as possible and move as little trash to the next place as possible.

Suppose there are k people involved in the cleaning process and x kilograms of trash on the current place. Then we have to consider three cases:

- if $x \le k$, we can clean all trash from the current place;
- if $x > 2k$ or $x > k$ and the current place is the last one, k people can't get the job done;
- if $k < x \le 2k$ and the current place is not the last one, we should dispose of as much trash as possible. $x - k = t$ means that exactly t people should "process" 2 kilograms of trash each, so exactly $2t$ kilograms of trash should be carried to the next place, and $k - t$ kilograms will be disposed of.

Since the complexity of each query is $O(n)$, and we use binary search to find the answer, the total complexity of the solution is $O(n \log A)$ (where A is up to $2 \cdot 10^5$).

Problem M. Distance

Author: Ivan Androsov

Preparation: Ivan Androsov

There is a solution in $O(1)$, but in fact, a solution that checks all points with x-coordinate from 0 to 50 and y-coordinate from 0 to 50 is fast enough. There's no need to check any other points, since $d(A, C) + d(B, C) = d(A, B)$ implies that point C is on one of the shortest paths between A and B.

Problem N. Haiku

Author: Ivan Androsov

Preparation: Mikhail Piklyaev and Dmitry Klenov

For each phrase in the input, let's calculate two values:

- v is the number of vowels in the phrase (excluding `y` and `Y`);
- y is the number of characters `y` and `Y` in the phrase.

Suppose the phrase should contain exactly V vowels (V is either 5 or 7). The maximum number of vowels we can obtain is $v + y$ (if each `Y` and each `y` is a vowel), and the minimum number is v (if each `Y` and each `y` is a consonant). So, we need to ensure that $v \le V \le v + y$. If this condition holds for every phrase of a three-line testcase, it can match the pattern $5 - 7 - 5$, otherwise it cannot.

Small implementation note: it can be worthwhile to transform each letter into lower (or upper) case using some built-in function, for example, `tolower` in C++.