A. Jumping Champa

Editorial

Let's consider any valid journey visiting all the cities. If cities are numbered from 1 to N, any such journey can be represented by a permutation of numbers from 1 to N. Let p be any such permutation. Then $p = [i_1, i_2, \ldots, i_n]$ and its cost can be written as $Q \cdot (|h[i_2] - h[i_1]|) + Q \cdot (|h[i_3] - h[i_2]|) + \ldots + Q \cdot (|h[i_n] - h[i_{N-1}]|)$.

The first observation we can make here, is that we can rewrite it as $Q \cdot (|h[i_2] - h[i_1]| + |h[i_3] - h[i_2]| + \ldots + |h[i_N] - h[i_{N-1}])$, so in order to get the minimum cost, we just need to choose the best permutation and we can ignore the Q multiplier for now - to get the final result, we will multiply the cost of the best permutation by Q at the end.

How to find the best permutation of the cities?

Notice that the minimal cost, in terms of height differences, to visit all the cities cannot be smaller than $M - m$, where M is the height of the highest city and m is the height of the lowest city. This is true, because at some point, you have to visit the highest city, at a different point, you have to visit the lowest city, and it is not possible to visit both of them at cost less than the difference of their heights.

The second observation is that, if you sort the cities in a ascending order, you can achieve the exact cost of $M - m$. This is true, because you are never traveling down, and the sum of costs to travel any such ascending sequence, equals the height of the last city minus the height of the first city, which equals in our case $M - m$.

To sum up, in order to solve a single test case, we just need to compute the minimum and the maximum height which can be done of course in $O(N)$ time.

B. Sonya clears the array

Editorial

This problem has a very straightforward recursive approach, but first things first.

Let $P[i, j]$ be the minimum cost of transforming $A[i]$ and $A[j]$ to consecutive primes. How to compute the minimum cost of transforming two numbers a, b to two consecutive primes, where the cost is the number of incrementations of both numbers? Well, since integers in the array are not very big, you can use sieve to precompute prime numbers and then quickly find the first prime p not greater than a and the first prime greater than p using lookup tables. From now, let's assume that we can compute $P[i, j]$ quickly.

Let's get back to the problem and define $F[i][j]$ as the minimum cost of clearing the subarray $A[i, j]$ of A. If we have just two numbers in the array, the answer is obvious. What if there are $N > 2$ numbers? Well, we can try any element at position $k > i$ as a pair for $A[i]$, such that there are even number of elements in $A[i, j]$ between $A[i]$ and $A[k]$. For a fixed k, the minimum cost of clearing $A[i, j]$, i.e. $F[i][j]$, equals $F[i + 1][k - 1] + F[k + 1][j] + P[i][k]$, because this is the minimum cost of clearing subarray $A[i + 1][k - 1]$, the minimum cost of clearing subarray $A[k + 1][j]$ and finally, since after these clearings $A[i]$ and $A[k]$ are adjacent, the cost of transforming them into consecutive primes. Now, iterating over all possible k, we can easily compute the answer.

One more thing, solving the problem by just using this recursive equation will easily lead to exponential time complexity, because we will be solving the same subproblems many times. In order to handle this, we can use dynamic programming or memoization to achieve polynomial solution.

Time complexity

Since there are $O(N^2)$ subarrays, and we solve a subproblem for a single one in $O(N)$ time, assuming that we have $P[i, j]$ precomputed, the overall time complexity is $O(N^3)$.

C. Sonya wants more equal numbers

Editorial

The first observation which we can make is that the maximum sum of elements in the array A is at most 10^5. Moreover, since N is not so large here, we can try the following approach.

Let's define $next[i][S]$ as the smallest ji, such that the suffix of subarray $A[i..j]$ sums up to S. For now, let's assume that we can compute $next[i][S]$, for each $1 \leq i \leq N$ and for each $1 \leq S \leq 10^5$. Can we solve the problem based on these values? Well, it is pretty straightforward, for each possible sum $1 \leq S \leq 10^5$, we can use $next$ table and act greedily to compute the number of non overlapping subarrays of A, which sum up to S. For example, if $next[1][S] = j$, then we know that the leftmost subarray which sums up to S ends in j, so we can take it into account and start searching the next one at index $j + 1$, i.e. follow the value of $next[j + 1][S]$ and so on. So far so good, we showed that if we have computed the next table, we can compute the result in $O(10^5 \cdot N)$ time.

How to compute the $next$ table? We will show how to compute it for a fixed sum S. In order to fill the whole next array, you need to run the below method for each possible S. First, we can compute $pref[i]$ as the i^{th} prefix sum of A, i.e. $pref[i] = A[1] + A[2] + \ldots + A[i]$. Having this computed, we are able to get the sum of any subarray of A in a constant time. Moreover, we can notice that computing $next[i][S]$ starting from $i = N$ is quite simple. Let's assume, that for some i, we have computed $next[i][S] = j$ and j is well defined, i.e. suffix of $A[i, j]$ sums up to S. Then, in order to compute $next[i - 1][S]$, we can first initialize it to j, and begin search for a smaller value starting from index $j - 1$. Since we can have precomputed $pref$ table, we can fill up $next$ table for a fixed S in $O(N)$ time.

To sum up, computing the whole next table takes $O(10^5 \cdot N)$ and computing the result based on next table also takes $O(10^5 \cdot N)$ time.

Extra credit:

You can notice that the above solution depends strongly on size of elements of A, but we can easily make this solution independent of these values. Just notice, that since there are $O(N^2)$ subarrays of A, there are at most $O(N^2)$ different sums of these subarrays. Knowing that, rather than iterating over all sums from 1 to maximum sum, we can iterate over only all possible sums. This transform our solution to strongly polynomial algorithm of running time $O(N^3)$. You may ask why we described a solution depending on values in A in the editorial? Well, it is easier to implement, and since you can code something faster, it is always the best to do it during the competition.

D. Sonya puts the blocks in the box

Editorial

We have N arrays of integers. The total number of elements in these arrays is at most 10^5. The goal is to pick at most one subarray from each of the arrays in such a way, that their total length is not greater than M and the sum of their elements is maximum. Notice that M is at most 1000 here. Of course, input arrays can contains negative elements.

If you are familiar with knapsack problem, you are good to go with the problem. For each input array, we can compute its best subarray for any size between 0 and M, where best subarray means the subarray with the greatest sum. Knowing these values, we can use the standard dynamic programming algorithm for the knapsack problem to solve the problem.

In more details, we process input arrays one by one. First, we compute the best subarrays of processed array for all sizes from 0 to M. Then we update our dynamic programming table with these values and we start processing next array.

E. Sonya and the graph with disappearing edges

Editorial

You are given a graph G, with N nodes and M edges. Your task is to find the minimum time needed to travel from node 1 to node N, starting at time 0, knowing that traveling trough any edge takes 1 unit of time. In addition, each edge e of K distincted edges, becomes unavailable from time $t[e]$, so you cannot use it at any time $t \geq t[e]$.

This problem can easily be solved using slightly modified `BFS`. You can think of a queue used in `BFS` like of a set of active nodes, to which you can travel for sure, and you know the minimum time to do it. Using these nodes and edges adjacent to them, you can try to extend the set of achievable nodes. In order to simulate disappearing of edges, while considering each edge e at some time t, we just need to check if $t < t[e]$, and if it is, we can use this edge, otherwise it is unavailable.

The crucial observation is that we perform the check in the earliest possible time, so if some edge is unavailable at any time t, we know that there is no way to go through it earlier. This is true, because `BFS` computes the shortest paths in unweighted graph.

Time complexity:

The total time complexity of this solution is $O(N + M)$, because we just need to add a check for availability of an edge during a standard `BFS` execution

F. Legendary Graph

Editorial

Let's analyze what happens to type 2 queries when queries of type 1 are processed. If the new edge connects two vertices already in the same connected component, effectively nothing will happen and we can discard the edge. So in the end, the set of vertices that exist in our graph will form a forest. This sounds quite similar to the disjoint-set data structure - query of type 1 will ask to merge two trees and query of type 2 will ask to answer a question on all the vertices in the subtree of the root.

Let's modify our way of thinking of how to merge two trees. Normally, we just attach an edge between the roots of the trees. However, we can also add a virtual node, and set that as the parent of both the vertices we want to merge. Now the reason we say subtree of the root is clear - that subtree will never change (gain new vertices) with future queries.

Suppose we process all queries of type 1 beforehand. Now for each query of type 2, we can switch the query from "Z's component at some point in time" to "the subtree of vertex Z' in the final forest", where Z' is the root of the tree of Z at the point in which it was queried. Our problem becomes much easier - we can use offline methods to solve the problem instead of being forced to answer queries online.

Let's perform a `DFS` through the tree and increment a time counter each time we enter a vertex. We will remember the time when we entered this vertex and the time when we left this vertex. If we assign each vertex to an index equal to the time when it was first visited, each subtree now corresponds to a contiguous range of indices. So we have reduced our original problem from queries on a dynamic graph to a simple array.

The next step is to further decompose the type 2 query from a contiguous range $[left, right]$ to $[1, left - 1]$ and $[1, right]$. If we add the intervals of the vertices in order by their index, we can answer all queries in the form $[1, index]$ when we process the vertex with visit time "index".

To do so, we now have to support two operations:

- Add 1 to each element in a subarray.
- Get the sum of a subarray (we will treat the natural numbers as an array to get the intersections efficiently)

These operations can be done with a segment tree, but that will most likely receive Time Limit Exceeded, due to the high constant. To optimize, we can use two binary indexed trees that support range add and range sum. You can determine how to update these trees by carefully analyzing what happens to a query when we add to a range. You can read more about this in Petr's blog (http://petr-mitrichev.blogspot.com/2013/05/fenwick-tree-range-updates.html).

G. Exchanging Letters

Editorial

In this problem, you have $N + 2$ people, numbered from 0 to $N + 1$, standing in a line. Each of K people, chosen from people with numbers from 1 to N, have given a letter. Each person who has a letter, has to pass it to his left or right neighbor. After this happens, let X be the sum of numbers assigned to people who have letters, and if one person has two letters, we count his number twice. You are interested how many ways of passing letters are there, such that $X = T$ for a given T.

This problem requires some analysis. First, we can compute what the minimum possible X is. Since people are numbered from left to right with increasing numbers, the smallest possible X is created when each person, who initially has a letter, decides to pass the letter to his left neighbor. Let T_0 be this minimum value. If we want to increase our X, we have to pick some number of people, and tell them to pass their letters to their right neighbors instead to left ones. Notice that, if we pick M people and change their initial decisions, we will increase T_0 by $2 \cdot M$, because we increase M numbers in X by 2. So let $Y := T - T_0$, in other words we have to increase the minimum possible result T_0 by Y to get the initial T. Based on the previous observation, there are $\binom{K}{Y/2}$ ways to do that.

To sum up, the only thing which remains it to compute the value of binomial coefficient modulo $10^9 + 7$ fast, and we can do this using factorials and modular inverse.

H. Xenny and Travel

Editorial

In this problem, you are given a graph G with N nodes and two distance matrices, A and B, between them. $A_{i,j}$ is the length of a road between nodes i and j, while $B_{i,j}$ is the length of a railway between them. Your task is to compute the length of the shortest path between two distincted nodes U and V, such that there exists another node Z, such that the path can be written as $U \to \ldots \to Z \to \ldots \to V$ and one of the following conditions is true:

- the path between U and Z consists of road only while the path between Z and V consists of railroads only
- the path between U and Z consists of railroads only while the path between Z and V consists of roads only.

How to solve this problem? If you are familiar with the single source shortest path problem, you are good to go.

Let $C_U(i)$ be the shortest path from U to i using only roads .
Let $D_U(i)$ be the shortest path from U to i using only railroads.

Moreover,

Let $E_V(i)$ be the shortest path from i to V using only roads.
Let $F_V(i)$ be the shortest path from i to V using only railroads.

Notice that if we can compute the above values, we can iterate through all nodes Z in the graphs and compute the result as the minimum over all Z from $\min\{C_U(Z) + F_V(Z), D_U(Z) + E_V(Z)\}$.

How to compute the distance functions? This is very straightforward, because in order to compute C_U and D_U we can just run Dijkstra algorithm on the original graph, and in order to compute E_V and F_V, we can reverse all edges in the graph and use Dijkstra to compute the shortest paths from V as a source vertex.

Time Complexity:

The overall complexity is dominated by running 4 times Dijkstra algorithm, which gives $O(N^2 \cdot \log N)$ time here.

I. Big travel

Editorial

In this problem, we are dealing with manhattan distance, which is defined for two points, (x_1, y_1), (x_2, y_2), as the sum of absolute difference between their coordinates, i.e. $|(x_1 - x_2)| + |(y_1 - y_2)|$.

Since we want to compute the sum of manhattan distances of all N points, the exact formula is the following: $\sum_{i=1}^{N} \sum_{j=i+1}^{N} |(x_i - x_j)| + |(y_i - y_j)|$. Notice that this formula can be decomposed into two independent sums, one for the difference between x coordinates and the second between y coordinates. If we know how to compute one of them, we can use the same method to compute the other, so from now, we will stick to computing the $\sum_{i=1}^{N} \sum_{j=i+1}^{N} |(x_i - x_j)|$. If we try to compute its value straight from the equation, it will take $O(N^2)$ time, which is too much to pass test cases. We need a faster approach.

Let's assume that we know all distances from a point xi to all values of $x's$ smaller than xi. Let's consider other point, the first one not smaller than xi, and call it x_j. How to compute the distances from x_j to all smaller points? Well, we can use the corresponding distances from x_i. Notice that each distance from x_j to some x_k, where $x_k < x_j$ equals the distance from x_i to x_k plus the distance between x_j and x_i. If there are A points smaller than x_j and S is the sum of distances from x_i to these smaller points, then the sum of distances from x_j to smaller points equals $S + (x_j - x_i) \cdot A$. If we sort all points in non-decreasing order, we can easily compute the desired sum of distances along one axis between each pair of cities in $O(N)$ time, processing points from left to right and using the above method. One more thing, notice that I used words smaller/greater points, but any two points might have equal coordinates, but do not be concerned about that. After sorting points in non-decreasing order, we say that a point x_i is smaller than x_j if and only if it appears earlier in the sorted array.

Time complexity:

Since computing sum of distances in 1 dimension takes $O(N \cdot \log N)$, because the time needed to sort these values dominate other computations, and the fact that we have to compute two such sums, one for x coordinates and the second for y coordinates, the total time complexity is $O(N \cdot \log N)$.

J. Game of sweets

Editorial

This problem can be rewritten as follows:

You have N non-negative integers, you randomly pick 2 of them, let's say A and B. The outcome of your choice is $|B - A|$. You are interested in how many different outcomes $|B - A|$ you can get.

For example, if we have 3 integers: 4 8 6, we can get 2 picking for example 8 and 6, or we can get 4, if you pick 8 and 4. Notice than, if there are at least two equals integers in the input, you can always get 0, this is very important while returning the final answer.

You may notice that one can try to solve this problem by trying all possible A and B, but this method has a running time of $O(N^2)$, which is too much for this problem. However, you may speed up this solution using bitwise operations to run in $O(N^2/32)$, which, if implemented fast, should pass all test cases. For more details, please refer to tester's solution.

The intended solution, without bitwise tricks, is to represent the sequence and its slightly modified copy as polynomials $P(x)$ and $Q(x)$ and use `FFT` to multiply these polynomials in order to get all possible outcomes as a subset of coefficients of $P(x) \cdot Q(x)$.

However, first thing first, let's reduce our problem from computing the difference between two elements to computing the sum of them.

Let's consider the following example, as above, let's assume that we have 3 integers: 4, 8 and 6. Let S be the the original sequence $[4, 6, 8]$ and W be the sequence $[M - 4, M - 6, M - 8]$, for big enough M, at least greater than the greatest integer in the input. Then, if we choose A from S and B from W, and we are able to compute all possible values of $A + B$, we just need to subtract M from these values in order to get the values of differences of elements picked only from original sequence.

What is remaining now, is to show how to compute all possible sums of elements A, B such that A is picked of one sequence of size N, and B is picked from the other one of the same size. Here comes the `FFT`, if we represent both arrays as polynomials $P(x)$ and $Q(x)$, for example $[4, 6, 8]$ can be represented as $P(x) = x^8 + x^6 + x^4$, and $Q(x)$ as $x^M - 4 + x^M - 6 + x^M - 8$, and we multiply these polynomials using `FFT` in $O(N \cdot \log(N))$ time, we are done, because we can extract all possible sums from coefficients of polynomial $P(x) \cdot Q(x)$.

K. Parity Game

Editorial

This problem is in fact a graph related one, which is the first step to come up with the solution. We have N boxes arranged in a line. Let's create our graph G now. First, we have to define what nodes are. We place a single node between each pair of boxes. In addition, we place one node before the first box, and one after the last box. After that, G has $N+1$ nodes. Next, for each query $[i, j]$ asked by Marcus, we connect the node placed just before the i^{th} box with the node placed just after the j^{th} box. We say that these nodes bound a range $[i, j]$.

After creating G, we can formulate the fact which is crucial to solve the problem.

The game is finished, i.e. Marcus can point out all boxes with candies without any mistake, if and only if G is connected.

In fact, we can prove something stronger, we for a set of queries $S = [(i_1, j_1), (i_2, j_2), \cdots, (i_k, j_k)]$, we know the parity of candies inside any range $[i, j]$ whose bounding nodes are connected by edges from S.

The intuition is that, if we have two ranges, A, B, sharing one node, and we know the parity of candies in boxes in A and B, we can deduct the parity of candies in boxes in a range $A \bigcup B$, A/B, and B/A. In order to prove the above fact, you can extend this intuition to a full proof by induction.

With the required knowledge, we can notice that, in order to make the game not finished, we have to disconnect the graph and we want to do it removing the smaller number of edges. This is a very classical and well studied problem in computer science. The edge connectivity of graph G is the largest number k, for which removing k edges from G do not disconnect it. In order to solve our problem, we want to compute the edge connectivity of G, or in the other words, compute the size of the smaller set of edges C, such that removing C from G disconnects it.

There are many solutions to this problem, and you can use any of them:

- Reduce the problem to finding `max-flow/min-cut` in a graph with unit cost edges from node s to node t, by picking arbitrary node as s and try any other from remaining nodes as t to minimize the size of the cut. This requires $O(N \cdot F(G))$, where $F(G)$ is the time complexity of used `max-flow/min-cut` algorithm. You can use either `Dinic's` algorithm, `Ford-Fulkerson` algorithm, `push-relabel` method, or any other to solve it.
- Use deterministic algorithm to find `min-cut` directly. Possibly the algorithm by `Stoer and Wagner` is the simplest to implement and very fast.
- Use randomized algorithm by `David Karger` to find `min-cut` directly with some probability, and then increase this probability by repeating the algorithm.