Long Contest Editorial

November 11, 2015

Moscow International Workshop ACM ICPC, MIPT, 2015

A. Album of numbers

The problem asked to keep a multiset of integers while adding/removing elements, and to find average smallest number of a random multi-subset of the current multiset (that is, every non-empty multi-subset has equal chance to be chosen) after every operation. Results should be accurate within 10^{-6} absolute or relative precision.

A. Album of numbers

Counting

Denote all different numbers in the multiset in the increasing order $b_1, \dots, b_k$. Let a_i be the number of copies of b_i. Then:

- the number of different non-empty multisubsets is $\prod_{i=1}^{k}(a_i+1)-1$.
- the number of multisubsets with b_i as the minimum is $a_i \prod_{j=i+1}^{k}(a_j+1)$ (choose no smaller numbers, at least one b_i, and greater numbers arbitrarily).
- therefore, the average minimal number is

$$\frac{\sum_{i=1}^{k} a_i b_i \prod_{j=i+1}^{k}(a_j+1)}{\prod_{i=1}^{k}(a_i+1)-1}$$

A. Album of numbers

Let's find numerator and denominator of

$$\frac{\sum_{i=1}^{k} a_i b_i \prod_{j=i+1}^{k}(a_j+1)}{\prod_{i=1}^{k}(a_i+1)-1}$$

separately. Divide them both by $P = \prod_{i=1}^{k}(a_i+1)$ so that they don't get too big.

A. Album of numbers

The denominator becomes simply $\frac{P-1}{P}$.
Every operation increases or decreases a_i by 1. Let a'_i by the new value of a_i, then $P' = P/(a_i + 1) \cdot (a'_i + 1)$.
The only problem is that P can overflow all built-in floating point types. To bypass this problem, store $\log P$ instead P. If $\log P$ is too large, then $\frac{P-1}{P}$ is very close to 1, otherwise find it explicitly.

A. Album of numbers

Numerator becomes

$$\sum_{i=1}^{k} \frac{a_i b_i}{\prod_{j=1}^{i}(a_j + 1)}$$

Observation

Let n be the number of operations, and A is the maximal possible value of b_i
Note that if all a_i are positive then i-th summand does not exceed $nA/2^i$. Therefore, to get the answer right up to ε error we have to sum up only first $O(\log(nA\varepsilon^{-1}))$ summands. Under given constraints summing only first 50-60 summands is enough.

Having noticed that, we can store a `std::set` (or a similar structure) of pairs (a_i, b_i) and compute the expression naively for every operation.

A. Album of numbers

The resulting solution has time complexity of $O(n \log(nA\varepsilon^{-1}) \log n)$ ($\log n$ factor for a set operation complexity).

B. Well Off

We are given a set of real variables x_i and a set of constraints of kind $\pm x_i \pm x_j > 0$. We have to check whether the set of constraints is satisfiable, that is, values of x_i can be chosen in such a way that all constraints are met.

B. Well Off

Rewrite all constraints as $\pm x_i > \mp x_j$. Build a directed graph as follows: create vertices for x_i and $-x_i$, then for every constraint $a > b$ add an edge from a to b (note that this also implies $-b > -a$).

Observation

If the constructed graph contains a cycle, then the set of constraints is contradictory, because by following a cycle we obtain that a certain number must be greater than itself.

Noitavresbo (converse observation)

If the graph is acyclic, then the constraints can be met. The graph can be sorted topologically, moreover, due to the symmetry of the graph, we can find such an ordering that is symmetrical upon reversing and changing all variables to their negations. After that, assigning real values to variables is trivial.

B. Well Off

Therefore, just build the graph as described above and look for cycles. It can be done in $O(n + m)$ with a simple DFS.

C. Accurate shots

Given binary representation of n without leading zeros, flip the least number of its bits so that the resulting number is divisible by m.
Also, output the number of different ways to do so along with the minimal divisible number obtainable this way.
This problem has a very tight memory limit (8 megabytes).
If the memory limit were more generous, we could implement a standard meet-in-the-middle solution: generate all possible states of left (with greater bits) and right (with lower bits) halfs of binary representations, then for every case for some half choose the best matching options from the other half (under condition of divisibility by m). Possible implementations use $O(\sqrt{n} \log n)$ time and $O(\sqrt{n})$ memory, which is too much in this problem.

C. Accurate shots

To deal with that, iterate over possible remainders of, say, left half modulo m. Iterate for all possible states of left half with the chosen remainder, store the answer (the minimal number of bits to flip)and the number of best choices for the left half. Independently, iterate over all matching states of right half, store the answers for them as well.
Finally, the minimal number of bits to flip is the sum of the answers for two halves, and the number of ways to obtain it is the product of two numbers for halves.
Clearly, each state of any half is considered only once. It suffices to notice that the right half's remainder cannot exceed $O(\sqrt{n})$, so that's a lower bound on the number of possible remainders to consider.
To sum up, the solution of time complexity $O(\sqrt{n})$ and memory complexity $O(1)$ is obtained.

D. Prom

Given two sequence of numbers a_i and b_j, and a number d, determine the number of pairs i, j such that $|a_i - b_j| \leqslant d$.
First, sort both sequences. For every number a_i we want to find all numbers b_j such that $a_i - d \leqslant b_j \leqslant a_i + d$. Clearly, such b_j form a segment in a sorted sequence, and we can find both ends of the segment using simple binary search.
So, total complexity is $O((n+m)\log(n+m))$ (both sorting and binary searching).

E. Impressive graphs

We are given a permutation of $1, \ldots, n$. Select k pairwise disjoint increasing subsequences of maximal total size.
This is not a full explanation of the solution, but rather an outline of direction to think.

E. Impressive graphs

Recall the algorithm for $k = 1$.

$k = 1$ (largest increasing subsequence)

Store d_j — the smallest number x such that there is an increasing subsequence ending in x. Initially, $d_0 = -\infty$, $d_j = \infty$ for all positive j. Take elements of the sequence from left to right; if the next element is x, find the largest d_j less than x and set $d_{j+1} = x$.

E. Impressive graphs

Generalization for $k > 1$

Store a table $d_{i,j}$. Initially, $d_{i,0} = -\infty$, $d_{i,j} = \infty$ for all positive j.
Take elements of the sequence from left to right. If the next element is x, find the largest $d_{1,j}$ less than x, then swap $d_{1,j+1}$ and x; after that, find the largest $d_{2,j}$ less than x and swap $d_{2,j+1}$ and x; and so on.
The maximal total size is the number of non-∞ elements in first k rows of the table.

Question 1

How would you prove the algorithm for $k > 1$?

Question 2

How can you enhance the algorithm to produce the sequences instead of the total size?

F. Pen

Given a bracket sequence with three types of brackets, determine the minimal number of brackets added in the beginning and/or the end of the sequence such that the sequence becomes correct, and output the resulting sequence, or find that it's impossible to do so.

F. Pen

Look at the sequence from left to right, and maintain a stack to find all unmatched brackets: that is, if the current bracket is closing and it matches the bracket on top of the stack, we remove the top, otherwise put the new bracket on top.

Observation

By adding brackets in the beginning we can only fix several first unmatched closing brackets until an unmatched opening bracket is met.
Same goes for adding brackets in the end. This implies that the sequence can be fixed only if unmatched brackets are first some closing brackets, and then some opening brackets.

F. Pen

Example

Consider a bracket sequence `()][]])(()[`.
Find all the unmatched brackets (marked in red): `()][]])(()[`
Add new brackets to fix the unmatched brackets (marked in blue):
`([[()][]])(()[])`

Complexity is $O(n)$.

G. Board game

We’re given a set of points on the plane with color assigned to every point. Find the maximal distance between all pairs of points which are of different color.

G. Board game

Consider the case when only two colors are present, that is, given two sets of points, we have to find the maximal distance between points from different sets.

Observation

We can add the interior of the convex hull to every set without changing the answer.
Short reasoning: let a and b be the farthest points from the convex hulls' interiors of sets A and B. Clearly, b is the farthest point of B in the direction of vector $b - a$ (that is, scalar product $(x, b - a)$ is maximized for $x = b$), similarily, a is the farthest point of A in the direction of vector $a - b$. That implies that both points are on convex hulls' borders, and therefore belong to initial sets (they can not lie on the sides, since otherwise the distance may be improved).

After the observation, we have reduced the problem to find the farthest pair of points in two convex polygons.

G. Board game

Definition

Minkowski sum of vector sets A and B is the set $C = \{x + y | x \in A, y \in B\}$.

Fact

Minkowski sum of two convex polygons is a convex polygon. Moreover, it can be found in $O(n)$ time.

Consider C — Minkowski sum of polygons A and $-B$ (which is obtained from B by changing every point's coordinates to their negatives). By definition, its points are exactly all vectors starting somewhere in B and ending somewhere in A. To find maximal distance, choose a vertex of C farthest from the origin. Therefore, the problem with two colors can be solved in $O((n+m)\log(n+m))$ (for building convex hulls of the sets).

G. Board game

How to deal with the multi-color problem?
Divide-and-conquer approach helps.
Consider the problem for colors with numbers from $[L; R)$.

- if $R - L = 1$, then there is only one color, so nothing to do.
- Otherwise, choose $M = \lfloor (L + R)/2 \rfloor$. Find answers for segments $[L; M)$ and $[M; R)$ independently. Then, find the farthest pair of points with one point with color from $[L; M)$ and another with color from $[M; R)$; that is a two-color problem considered before.

On every depth level of recursion total number of operations does not exceed $O(n \log n)$, therefore total complexity is $O(n \log^2 n)$. This is enough to get OK.
The solution can be optimized to $O(n \log n)$ if we merge convex hulls of two halves in linear time instead of building new one from scratch.

H. Scouts

Given an array of n integers, build a binary tree as follows: choose an element as the root, then recursively build trees on left and right halves (if they're not empty). Build such a tree that minimizes maximal sum from path to leaves.

H. Scouts

Let's write subsegment DP formula (for the set $[l; r)$):

$$dp_{l,r} = \left(\min_{i=l}^{r-1} a_i + \max(dp_{l,i}, dp_{i+1,r})\right)$$

It is evident that $dp_{l,r}$ increasing over increasing r (or decreasing l).
We could try to use some clever tricks like Knuth's optimization, but a_i summand screws up monotonicity of optimal i.

H. Scouts

Still, from monotonicity over segment extending it follows that for any $l < r$ there exists k such that

- $dp_{l,j} \leqslant dp_{j+1,r}$ for $l \leqslant j \leqslant k$
- $dp_{l,j} > dp_{j+1,r}$ for $j < k \leqslant r$

Given values of all $dp_{l,j}$ and $dp_{j+1,r}$ for $l \leqslant j < r$ we can find k with binary search.
After that, we can rewrite the formula as:

$$dp_{l,r} = \min\left(\min_{l \leqslant j \leqslant k} a_j + dp_{j+1,r}, \min_{j < k \leqslant r} a_j + dp_{l,j}\right)$$

H. Scouts

It suffices to compute expressions of sort $\min_{l \leqslant j \leqslant k} a_j + dp_{j+1,r}$ and $\min_{j < k \leqslant r} a_j + dp_{l,j}$.
To do this, build segment trees for all l's storing numbers $a_j + dp_{l,j}$ for all $j \geqslant l$, and for all r's storing numbers $a_j + dp_{j+1,r}$ for all $j < r$. Change the trees' elements as $dp_{l,r}$ are computed.
To sum up, for every segment $[l; r)$ we perform $O(\log n)$ operations:

- Binary search to find k (in $O(\log n)$)
- Make queries to segment trees to compute $dp_{l,r}$ (in $O(\log n)$)
- Change segment trees to account for just computed value of $dp_{l,r}$ (in $O(\log n)$)

That makes for an $O(n^2 \log n)$ solution.

I. Insects

We have several types of bugs with parameters x_i, y_i, z_i. We can buy an antidote for any particular x_i for A coins, for any y_i for B coins, and for any z_i for C coins. After that, all bugs, for which x_i, y_i and z_i are countered with an antidote, are killed, and we earn p coins for each bug. Find the maximum profit.

I. Insects

Suppose that we can choose which antidotes we buy and which bugs we kill arbitrarily. To get a problem which is equivalent to the original we introduce the following penalties:

- penalize for A, B, or C for buying corresponding type of antidote
- penalize for p for *not* killing a bug
- penalize for ∞ for killing a bug and not buying an antidote (of any kind) that kills him

The problem has now become a straightforward min-cut application.

I. Insects

Build a network as follows:

- Create source S and sink T
- Create vertices for all bugs b_i and for all antidotes x_i, y_i, z_i
- Add edges from S to all b_i with capacity p
- Add edges from all x_i (y_i, z_i) to T with capacity A (B, C)
- Add edges from all b_i to corresponding x_i, y_i, z_i with capacity ∞

Consider an $S - T$ cut in the resulting graph. Its capacity (total capacity of edges from S's half to T's half) coincides with total penalty for killing all bugs and buying all antidotes in the same half with S. This means that finding minimal penalty (and, therefore, maximal profit) reduces to finding minimal $S - T$ cut in the network.
By Ford-Falkerson theorem, this coincides with the maximal flow in the network. Any sufficiently efficient algorithm (like Dinic's algorithm or scaling) is fast enough.

J. Caves

We are given an undirected unweighted graph (possibly disconnected). We are standing at the vertex 1. Each other vertex has a certain chance to contain treasure. On each turn we can choose one of two possible moves:

- follow an adjacent edge to another vertex, spend 1 time
- jump to a random vertex chosen equiprobably, spend t time

Find the minimal average time needed to find the treasure (under optimal strategy).

J. Caves

Suppose that we have already visited a subset S of vertices without finding the treasure, and currently are standing in the vertex $v \in S$. Denote $d_{S,v}$ the minimal average time to win under these conditions.

Observation

There are two possible strategies to follow:

- Follow a path to a yet unvisited vertex
- Immediately jump to a random vertex

Indeed, it doesn't make sense to make steps and then make a jump before visiting new vertex, since we could have jumped from the start.

J. Caves

Choose a subset S, and assume that we know the answers for all its proper supersets. Compute

$$d'_{S,v} = \min_{u \notin S} \left((1 - p_u / \sum_{w \notin S} p_w) d_{S \cup \{u\}, u} + \rho(u, v) \right)$$

— the answer for starting vertex v without considering random jumping (here $\rho(u, v)$ is the length of the shortest path between u and v).

Let E_S be the average time to win after a random jump if the visited subset is S (clearly, the time doesn't depend on the vertex). The following equation should be satisfied:

$$E_S = t + \frac{1}{n} \left(\sum_{u \notin S} dp_{S \cup \{u\}, u} + \sum_{v \in S} \min(E_S, dp'_{S,v}) \right)$$

J. Caves

It suffices to find E_S. Denote $R = \sum_{u \notin S} dp_{S \cup \{u\}, u}$. Transform:

$$\sum_{v \in S} \min(E_S, dp'_{S,v}) = n(E_s - t) - R$$

Sort all numbers $dp'_{S,v}$ to obtain the sequence $d_1, \ldots, d_k$, also put $d_0 = 0$, $d_{k+1} = \infty$. Suppose that $d_j \leqslant E_S < d_{j+1}$, then

$$\sum_{i=1}^{j} d_i + (k - j)E_S = n(E_S - t) - R,$$

or $E_S = \frac{R + nt + \sum_{i=1}^{j} d_i}{n - k + j}$
For every j such that $0 \leqslant j \leqslant k$ compute possible value of E_S and check that it falls in the interval $[d_j; d_{j+1})$, from these choose minimal. Since E_S should be well-defined, it will be found valid in at least one case. After that put $dp_{S,v} = \min(dp'_{S,v}, E_S)$.

J. Caves

Finally, we use subset DP to compute all $dp_{S,v}$ by decreasing of S. The answer is $dp_{1,0}$. The complexity is $O(2^n n \log n)$.

K. Blocks

For a permutation of $1, \ldots, n$ we can count numbers a — the number of elements which are greater then all to its left, and b — the number of elements which are greater then all to its right. Count the number of permutations for many values of n, a and b.

K. Blocks

Denote $f(n, a, b)$ the answer to the problem. Consider erasing the smallest number (1) from a permutation with $n > 1$ elements and renumbering all the rest elements (that is, decreasing them by 1).

- If 1 was the leftmost element, then a decreases by 1, b stays the same
- If 1 was the rightmost element, then b decreases by 1, a stays the same
- Otherwise, both a and b stay the same

That leads to the formula
$f(n, a, b) = f(n-1, a-1, b) + f(n-1, a, b-1) + (n-1)f(n-1, a, b)$ if $n > 1$.
Of course, $f(1, a, b) = 1$ if $a = b = 1$, and 0 in all other cases.
If a and b do not exceed k, straightforward DP solution we obtained requires $O(nk^2)$ time and memory, which does not pass.

K. Blocks

For a given permutation construct a sequence as follows: consider all numbers from $n-1$ to 1; if the number x is to the left (to the right) of all greater number, append L (R) to the sequence, otherwise append $n-1-x$ — the number of ways to place x, but not on the either end.

Example

For example, it's easy to see that for the permutation $(3, 5, 1, 4, 2)$ we obtain the sequence $R, L, R, 3$.

The number of permutations with the chosen sequence is exactly the product of all numbers in the sequence (that is, in the example other two permutations $(3, 1, 5, 4, 2)$ and $(3, 5, 4, 1, 2)$).

K. Blocks

Observation

The answer $f(n, a, b)$ is the sum of products of numbers over all possible sequences of $n-1$ elements with $a-1$ L's and $b-1$ R's.

Observation

It doesn't matter for the product if we put L's or R's into the sequence, we can change them all to X and multiply the answer by $\binom{a+b-2}{a-1}$.

Denote $g(n, k)$ the sum of products of numbers over all sequences of $n-1$ elements with k X's.

Observation

$g(n, k) = g(n-1, k-1) + (n-1)g(n-1, k)$

This recurrence can be precomputed in $O(nk)$ time and memory. The answer is $f(n, a, b) = g(n, a+b-2)\binom{a+b-2}{a-1}$.

L. Postman

We are given an undirected unweighted tree. We can choose a starting vertex and walk the tree so that to visit all the vertices. Minimize the sum of times of first visit for all vertices.

L. Postman

Let's solve the problem if the starting vertex is fixed.
It's evident that we should choose the order of subtrees to traverse and visit all vertices from the first subtree first, the from the second subtree, and so on. In each of the subtrees we choose the order of its root's subtrees, and so on.

Observation

The answer (total time) doesn't depend on the order of subtrees.

L. Postman

Proof

Try to change the order of two adjacent subtrees of a vertex. Let first subtree contain k_1 vertices and have answer t_1 (that is, the minimum total first visit time to start from the root and visit all vertices in the subtree), similarily, for the second subtree the numbers are k_2 and t_2.
If the first subtree comes first, the answer is k_1 (additional delay 1 to step into the first subtree) $+t_1$ (traverse the first subtree) $+k_2(2k_1 + 1)$ (additional delay to enter the second subtree) $+t_2 = t_1 + t_2 + k_1 + k_2 + 2k_1k_2$. From the symmetry it follows that the answer is independent of the order of subtrees.

It's fairly easy to use subtree DP to compute answer for the fixed root.

L. Postman

Now to account for all possible roots. To do that, consider a vertex v which is not a root chosen before. In addition to its leaves' answers and sizes we have to know the answer and size of the subtree over the edge to its parent p.
To compute the answer for the "parent" subtree we can use the answer for the case when p is root and "remove" the subtree containing v. To do that, note that the order is irrelevant, so we may assume that v's subtree comes last in the order and subtract its contribution from p's answer.
Implement (yet another) DFS that computes the answer for every root using parent's results. This makes for a linear solution.