Problem A. Advertisement Matching

Problem author: Jaehyun Koo (koosaga)
First solver: 윤형신 외 2명 (Onsite, 107 minutes), dotorya (Open, 71 minutes)
Total solved team: 6 (Onsite), 4 (Open)

Maximum flow modeling

The easiest way to model this problem is to use maximum flow. Make n nodes for advertisers and m nodes for recipients. Attach them to source and sink with capacities of a_i and b_j, respectively. For each pair of advertiser and recipient, connect them with an edge with capacity 1: No recipient should receive more than one ad from a specific advertiser. The delivery is possible if and only if the maximum flow of such graph is equal to $\sum_i a_i$. By using any standard maximum flow algorithm, we obtain a polynomial time algorithm.

Optimizing maximum flow

Interestingly, we don't need any maximum flow algorithm to compute the maximum flow of the above graph.

Theorem. WLOG assume a is sorted in decreasing order. Delivery is possible if and only if $\sum_{i=1}^{m} min(b_i, k) \geq \sum_{i=1}^{k} a_i$ holds for all $0 \leq k \leq n$.

This can be proven with a fancy construction, but here we will only rely on flow-type arguments.

Proof. We'll try to compute the minimum cut for the above graph and leverage the max-flow min-cut theorem. Let $S \subseteq [n], T \subseteq [m]$ be the set of vertices that is connected to the source. Then the value of the cut is: $\sum_{i \notin S} a_i + \sum_{i \in T} b_j + |S| \times (m - |T|)$.

We want to minimize this value for all $S \subseteq [n], T \subseteq [m]$. Fix $|S| = k$. Then:
$\sum_{i=k+1}^{n} a_i + \sum_{i \in T} b_j + k(m - |T|)$
which is
$\sum_{i=k+1}^{n} a_i + \sum_{i \in T} b_j + \sum_{i \notin T} k$

We should therefore choose T that minimizes $\sum_{i \in T} b_j + \sum_{i \notin T} k$. If we take all j with $b_j \leq k$, then:
$\sum_{i=k+1}^{n} a_i + \sum_{i=1}^{m} \min(b_j, k)$
The minimum of this should be at least $\sum_{i=1}^{n} a_i$. Therefore, for each $0 \leq k \leq n$:
$\sum_{i=k+1}^{n} a_i + \sum_{i=1}^{m} \min(b_j, k) \geq \sum_{i=1}^{n} a_i$
$\sum_{i=1}^{m} \min(b_j, k) \geq \sum_{i=1}^{k} a_i$ □

Optimizing the algorithm

Imagine a histogram of b where elements are ordered in decreasing order of b_i. We see that $\sum_{i=1}^{m} min(b_i, k)$ is the number of blocks in the lowest k rows. If we *transpose* b, then we can compute the desired sum for a given value of k with prefix sums.

Since the value of b_i will remain small after each query (even though it's not necessarily true, we'll assume for simplicity that $b_i \leq n$), we can afford to use this technique. Let c_k be the number of b_i such that $b_i \geq k$. Then, $\sum_{i=1}^{m} \min(b_i, k) = \sum_{i=1}^{k} c_i$. Therefore, we can implement the previous logic in linear time per query.

Finally, it turns out we can explicitly maintain a sorted array a_i and an array c_i even with the update queries.

- If we have to increment or decrement some a_i, then we can take the first/last occurrence of a_i and decrement or increment it, which doesn't break the sorted condition. We can find the first/last occurrence with binary search.
- If you change the value of b_i by one, only c_{b_i} or c_{b_i-1} can possibly change, which can be easily tracked.

We have to check if $\sum_{i=1}^{k}(c_i - a_i) \geq 0$. This is a standard segment tree problem, where for each node, we maintain the minimum prefix sum of $c_i - a_i$ inside the interval, along with the sum of $c_i - a_i$.

Shortest solution: 1634 bytes

Problem B. Bombs In My Deck

Problem author: Donghyun Kim (kdh9949)
First solver: RUN to GoN (Onsite, 7 minutes), dotorya (Open, 4 minutes)
Total solved team: 12 (Onsite), 27 (Open)

Let's calculate the probability that Donghyun **loses** next turn. We can then subtract it from 1 to get the answer.

Donghyun loses if and only if Donghyun draws $D = \lceil \frac{C}{5} \rceil$ bombs in a row. So, we have to calculate the probability that the D topmost cards in his deck are all bombs. There are two cases:

- If $D > B$, the probability is 0. (he never loses next turn.)
- Otherwise, the probability is $\frac{B}{A} \times \frac{B-1}{A-1} \times \cdots \times \frac{B-D+1}{A-D+1}$.

Since the input values are very small, we don't need to worry about precision issues when doing floating point computation.

Shortest solution: 188 bytes

Problem C. Economic One-way Roads

Problem author: Jaemin Choi (jh05013)
First solver: N/A (Onsite), N/A (Open)
Total solved team: 0 (Onsite), 0 (Open)

In this problem, you are asked to compute the minimum cost needd to orient each edge of the undirected graph such that the graph is strongly connected.

The main idea is to represent a strongly connected graph as something we can gradually construct.

Theorem. A graph G is strongly connected if and only if it can be constructed using the following procedure:

1. Start with N isolated vertices. Pick any vertex v, and let $S = \{v\}$.
2. Repeat the following until $S = V(G)$:
   (a) Pick two vertices v and $u \in S$. The two vertices can be the same.
   (b) Pick zero or more distinct vertices $w_1, \cdots, w_k \notin S$.
   (c) Connect $v \to w_1 \to \cdots \to w_k \to u$, and put $w_1 \cdots w_k$ into S.

This process is known as "ear decomposition", and the trail $v \to w_1 \to \cdots \to w_k \to u$ is called an "ear".

Let's start with an $O(N^2 3^N)$ solution, even though it's not fast enough to solve the problem. We compute $dp[S]$, the minimum cost to make S equal to the given set, where S is as described in the above theorem. To construct the next ear, there are $O(|S|^2 2^{N-|S|})$ possible choices of the starting and ending vertices in S, and the intermediate vertices not in S. Details are skipped since it's not required for the faster solution.

Before moving on, there is one problem: when you add zero intermediate vertices, there is a cyclic relation in this DP relation. To resolve this problem, reduce the cost of orienting each edge so that one of the directions doesn't cost anything. Specifically, if orienting an edge costs x or y depending on its direction, subtract $\min(x, y)$ from each cost. Now we can always assume that all edges contained within S are added. Don't forget to add the sum of $\min(x, y)$ back to the final answer.

To optimize the algorithm, we memoize the process of constructing an edge. We compute $dp[S][u][w][b]$, where S is the set of connected vertices including the current (incomplete) ear, u is the ending vertex of the ear, and w is the current intermediate vertex (or u if the ear is complete). Also, b is a boolean variable, which is true if we are allowed to connect w to u and complete the ear, and false otherwise. We use this variable because you cannot orient an edge in both directions. Additionally, note that if the ear is complete, then u, w, and b are irrelevant, so you just need to compute $dp[S][\cdot][\cdot][\cdot]$ once for each S in that case. This gives an $O(N^3 2^N)$ algorithm, which works in time.

Shortest solution: 1624 bytes

Problem D. Fix Wiring

Problem author: Younghun Roh (Diuven)
First solver: TunaMilk (Onsite, 17 minutes), contest_account (Open, 16 minutes)
Total solved team: 12 (Onsite), 19 (Open)

Let's denote the list of tag values sorted in non-decreasing order as $A_1, A_2, \cdots A_M$.

To minimize the installation cost, greedily use the minimum values from A_i, by assigning A_i to the wire between node 1 and node $i+1$. The cost is $\sum_{i=1}^{n-1} A_i$.

To maximize the installation cost, we should 'waste' as many small values as we can. We will repeatedly form a complete graph of increasing size using the smallest available values. For $2 \le k \le N$, assign $\binom{k}{2}$ smallest tags to form a complete graph of nodes $1, 2, \cdots k$. Then, the smallest value connecting k and $1, 2, \cdots k-1$ is $A_{\binom{k}{2}+1}$. The cost is $\sum_{i=1}^{n-1} A_{\binom{i}{2}+1}$.

Shortest solution: 117 bytes

Problem E. LCS 8

Problem author : Jaehyun Koo (koosaga)
First solver: 윤형신 외 2명 (Onsite, 235 minutes), N/A (Open)
Total solved team: 1 (Onsite), 0 (Open)

Consider the typical DP approach for computing the LCS. Since we want to compute the number of strings that have a certain value on the entry $dp_{n,n}$, we can try to use dynamic programming where our state is the **dynamic programming table for computing the LCS**. In other words, our state is a tuple of $\{i, dp[i][0], dp[i][1], dp[i][2], \ldots, dp[i][n]\}$, where we have completed the first i states, and the resulting DP table (for LCS) looks as such. To generate another row, it is sufficient to supply the character for t_i (A to Z) and the DP table for previous rows. This gives a solution with about $O(N^{N+2})$ states.

Observe that the above definition actually has only $O(N \times 2^N)$ states. This is because the value $dp[i][j+1] - dp[i][j]$ in the LCS DP table is either 0 or 1. Now, instead of taking the whole array, we can instead save a bitmask of the differences of the $dp[i]$ table.

Let's optimize further by exploiting the fact that $K \le 3$. For the LCS to have length at least $N-K$, the path denoting the LCS in the DP table should be almost completely diagonal. In fact, any path that gives an LCS of length at least $N-K$ only passes the cells with $|i-j| \le K$. Thus, we only have to save the value $dp[i][i-K], dp[i][i-K+1], \ldots, dp[i][i+K]$, which can be represented with $2K$ differences and the value of $dp[i][i]$ itself. Here, you can again notice that $dp[i][i] \ge i-K$ for a valid LCS table: Therefore there exists only $O(2^{2K} \times (K+1))$ valid states for each row, which is sufficient for us to store.

Our final concern is that it's quite heavy to recompute the states every time. Here, note that every state transition only depends on the previous state and the predicates whether $S_{i-k}, S_{i-k+1}, \ldots, S_{i+k}$ matches with potential T_i or not. We can simply precompute all state transitions in $O(2^{4K} \times 26)$ time, and then proceed with the DP afterwards. (However, since the TL was very lenient, you can pass this problem by recomputing the states every time, if well implemented).

Shortest solution: 1110 bytes

Problem F. Min-hashing

Problem author: Jihoon Ko (jihoon)
First solver: 윤형신 외 2명 (Onsite, 13 minutes), dotorya (Open, 35 minutes)
Total solved team: 12 (Onsite), 9 (Open)

Let A be the adjacency matrix of the input graph G. Then, the (i, j)-th entry of the matrix A^k becomes the number of paths of length k from node i to j.

Theorem. $h_v^{(k)} = \min_{A^k_{u,v}>0} l_u$.

Proof. For $k = 1$, the statement is trivial by definition of shingle value. Assume that the statement holds for $k = k_0$. Then, $h_v^{(k_0+1)} = \min_{\exists w: A^{(k_0)}_{u,w}>0 \wedge A_{w,v}=1} l_u = \min_{A^{(k_0+1)}_{u,v}>0} l_u$, which implies the statement holds for $k = k_0 + 1$. Therefore, by induction, the statement holds.

Also, since $A^{(k)}_{u,v} > 0 \Rightarrow A^{(k+2)}_{u,v} > 0$ holds for every $u, v \in V$ and every positive integer k, the answer of the problem is c_k when k is sufficiently large.

For sufficiently large integer k and nodes $u, v \in V$, there are three possible cases:

1. If u and v are in different components.
2. If u and v are in the same non-bipartite component C.
3. If u and v are in the same bipartite component C whose bipartition has the parts C_0 and C_1.

It is trivial that $h_u^{(k)} \neq h_v^{(k)}$ for case 1, and $h_u^{(k)} = h_v^{(k)}$ for case 2. For case 3, $h_u^{(k)} = h_v^{(k)}$ if and only if u and v are in the same part. So, you can solve the problem by checking whether a given component is bipartite or not for every connected component, and the whole algorithm takes $O(n + m)$ time.

Shortest solution: 754 bytes

Problem G. Mock Competition Marketing

Problem author: Taehyun Lee (etaehyun4)
First solver: 윤형신 외 2명 (Onsite, 10 minutes), contest_account (Open, 7 minutes)
Total solved team: 12 (Onsite), 16 (Open)

Try all ($2^6 = 64$) possibilities of bidding strategies, where you decide whether to bid or not for each ad-type.

If you fix the set of ad types to bid, then the number of successful bids can be found in a single loop, where you simulate the auction by maintaining the current available balance.

Shortest solution: 335 bytes

Problem H. Query On A Tree 17

Problem author: Jaehyun Koo (koosaga)
First solver: N/A (Onsite), contest_account (Open, 166 minutes)
Total solved team: 0 (Onsite), 2 (Open)

Let x be the optimal vertex that minimizes the total distance. Root the tree at vertex x. If there exists a subtree that contains strictly more than half of the people, moving x to that position strictly decreases the total distance. Hence, the optimal vertex should contain no subtree that contains more than half of the people: this concept is usually known as the *centroid*.

x may not be unique: There may exist a subtree that contains exactly half of the people, and moving towards that subtree does not change the distance. Indeed, the set of possible x forms a path in the tree. In this problem, we have to find the vertex closest to the root, which is the LCA of the two endpoints. Here, we will assume $S = \sum_i A[i]$ is odd, which means the answer is unique. The case of even S will be trivial after we can handle odd cases.

Root the tree from an arbitrary vertex. Except for the root vertex, each vertex v has a subtree directed towards the root. In the optimal solution, this subtree should contain at most $\frac{S}{2}$ people, which means the rooted subtree v should contain at least $\frac{S}{2}$ people. If we sort the people by the DFS preorder value of their respective vertices, each subtree contains a people that forms a contiguous range over that sorted order. The optimal solution's interval contains more than half of them: Thus, it should always contain the median over the DFS preorder sequence. So, if we know the median, and can find how many people that subtree v contains, then we can find x by doing a binary search with jump pointers (sparse table).

Now we have to maintain $A[i]$ over path updates and subtree updates. The operation we need is range addition or range sum queries over Euler tour orders, so you can use segment trees (or even Fenwick trees) as a data structure. Path update can be tricky, but the heavy-light decomposition can be maintained in a way such that it is compatible with Euler tours: See `https://codeforces.com/blog/entry/53170` on how.

Shortest solution: 2062 bytes

Problem I. Remote Control

Problem author: Jaemin Choi (jh05013)
First solver: 윤형신 외 2명 (Onsite, 32 minutes), contest_account (Open, 84 minutes)
Total solved team: 7 (Onsite), 7 (Open)

Let's receive all questions before answering any of them. It would be good if there is an efficient way to simulate the toy car for Q starting positions at once.

You can't move all the toy cars. However, motion is relative, so let's move something else - specifically, the

wall. Whenever the toy cars have to move in one direction, move the wall in the opposite direction instead. After all N movements, reassign the grid's coordinates so that the wall is placed at $(0, 0)$, and use it to answer all the questions.

How do we deal with collisions? Whenever some cars are going to collide with the wall, push them in the direction the wall is moving to. However, there may be up to Q cars to push, since multiple cars may be in the same cell. To resolve this, note that the cars in the same cell will never be separated; they will always be in the same cell. Group the cars in the same cell into one group, and place the group in the grid instead of the cars. (A group may have only one car). Now, you will push one group at a time.

Sometimes, two groups may have to be merged into one. A typical method is to label each group with an integer and use a disjoint set data structure, which is enough to solve the problem in $O((N+Q)\log Q)$. Here we introduce another useful method called the "smaller to larger technique": always merge the smaller group into the larger group. Then each car migrates to another group at most $O(\log Q)$ times, since whenever the car migrates, the size of the group increases to at least twice the previous size. The whole algorithm takes $O((N+Q)\log Q)$ time.

Shortest solution: 905 bytes

Problem J. Sewing Graph

Problem author: Donghyun Kim (kdh9949)
First solver: 윤형신 외 2명 (Onsite, 46 minutes), songc (Open, 29 minutes)
Total solved team: 11 (Onsite), 10 (Open)

To connect the N dots for the both sides of the cloth, we need at least $2(N-1) = 2N-2$ edges. So, the length of any valid sewing sequence should be at least $2N-1$. Then, would it be possible to make a beautiful pattern with $2N-1$ numbers?

Let's assume that all N dots are on a straight line, ordered by their indices. We can see that the sequence $\{1, 2, \cdots, N-1, N, N-1, \cdots, 2, 1\}$ generates a beautiful pattern (see Figure 1).

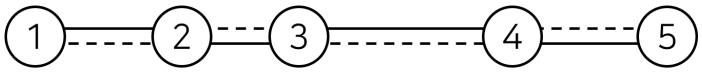

Figure 1. Strategy for dots on a straight line.

We can apply this approach more generally. Suppose that we've sorted the dots $D_1, D_2, ..., D_N$ ($D_i = (x_i, y_i)$) with the following comparison criteria:

- If $x_i \neq x_j$, $D_i < D_j$ when $x_i < x_j$. Otherwise, $D_i < D_j$ when $y_i < y_j$.

Now, we can see that segment $\overline{D_iD_{i+1}}$ and $\overline{D_jD_{j+1}}$ do not intersect for all $1 \leq i < j \leq N-1$. Therefore, we can use the same strategy again (see figure 2).

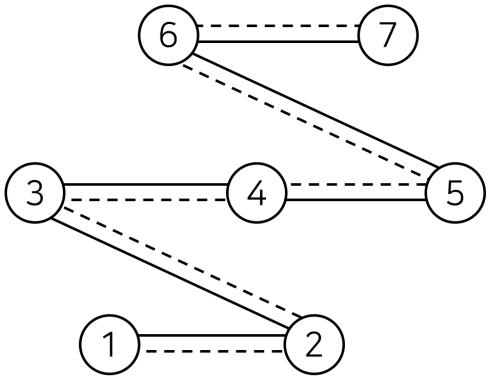

Figure 2. Strategy for general case.

Therefore, we can always find a sewing sequence of length $2N - 1$ which generates a beautiful pattern.

The above solution was the simplest one, and there can be some different (and more complex) approaches. For example, we can build a Euclidean MST on the both sides of the cloth.

Shortest solution: 173 bytes

Problem K. Square, not Rectangle

Problem author: Jaemin Choi (jh05013)
First solver: 카이마루 (Onsite, 7 minutes), songc (Open, 9 minutes)
Total solved team: 12 (Onsite), 21 (Open)

Instead of asking "what is the answer?", we can ask "is the answer at least x?" If we know that the answer is at least L and at most R, we let $x = \frac{L+R}{2}$ (rounded down) and ask the question. Then, depending on the answer, we either know that the answer is at most $x - 1$, or know that the answer is at least x, thereby reducing the range of possible answers by half. This technique is known as **binary search**.

How can we determine whether the answer is at least x in this problem? We have to check whether the histogram contains a square whose side length is x. The largest square could be larger than x, but if so, it clearly contains a smaller square as well.

Now, we have to check if there are x consecutive numbers whose values are at least x. Maintain a counter and look over the numbers one by one. If the number is at least x, increment the counter. Otherwise, reset it to 0. If the counter reaches x at any point, then yes, the histogram contains a square whose side length is x. This process takes $O(N)$, and since we are applying this process $O(\log N)$ times due to the binary search, the whole algorithm runs in $O(N \log N)$.

There is another, more difficult solution. If you know how to find the largest rectangle in the histogram in $O(N)$ using a stack, then you may realize that you can actually apply the exact same strategy to find the

largest square. The proof is left to the reader for exercise.

Shortest solution: 277 bytes

Problem L. Steel Slicing 2

Problem author: Jaehyun Koo (koosaga)
First solver: TunaMilk (Onsite, 129 minutes), contest_account (Open, 252 minutes)
Total solved team: 1 (Onsite), 1 (Open)

Let's forget about the histogon and consider this as a general simple rectilinear polygon (polygons with axis-parallel sides). Each vertex of the polygon is either *convex* or *concave*, and a non-empty rectilinear polygon is a rectangle if and only if it is composed only with convex sides. Our goal is to use the laser cutter appropriately, to remove all concave vertices over all polygons.

Lemma. In the optimal solution, for each cut made by a laser cutter, one of its endpoint is a concave vertex.
Proof. An endpoint cannot be convex by definition. Suppose that there exists an operation such that both endpoints are in the middle of the segment. We can move both endpoints to the left or right until it touches a vertex. Any other operation which becomes impossible after this movement, can be replaced by this exact operation, so optimality is preserved.

Thus, all operations have to touch a concave vertex. An operation will split a concave vertex into a segment and a convex vertex, essentially removing a concave vertex. So, each operation removes one or two concave vertices at its endpoints. Now the problem is equivalent to finding the maximum number of operations, where both endpoints lie on concave vertices.

Let's find a list of segments which have both endpoints as concave vertices. For vertical segments, this is trivial. For horizontal segments, we can consider each side of the histogram separately. For each side, we can maintain a stack which stores an increasing sequence of heights, and whenever we have a new block r, we can find a matching block l where $H_l = H_r$ and the right corner of l and left corner of r form a valid segment. Alternatively, you can check all pairs of adjacent elements of equal heights, and use range maximum query to check if the pair forms a valid segment. In total, this operation takes $O(n)$ or similar time.

Let's denote the operation that removes two concave vertex as a *2-operation*. Obviously, any 2-operation can be represented as one of the segments we have found.

Theorem. The maximum possible number of 2-operations is equal to the size of the maximum independent set over the segments, where two segments intersect if they share a point (including its endpoints).
Proof. Since we don't cut the same place twice, any independent set of 2-operations can be fully performed. If a 2-operation is performed, it creates no concave vertices in between, and no 2-operation can reach the segment where 2-operation was previously done.

At this point, we have a polynomial-time algorithm for the problem. Compute the number of concave vertices by simple iteration. Since all vertical segments don't intersect each other, and all horizontal segments also don't, we can build a bipartite graph over a set of segments and find a maximum independent set over a graph. Maximum independent set is a complement of minimum vertex cover, so we can use König's theorem. Standard bipartite matching algorithms will give about $O(n^3)$ time complexity.

To optimize this algorithm we have to use a geometric property of histogons. Observe that you can extend the vertical segment infinitely. The extended part is either in the border or outside the polygon, while the horizontal segment lies (almost) completely inside the polygon. This implies that, we can consider a vertical segment as a point in an x-axis, and a horizontal segment as an interval in an x-axis. We have simplified our problem into finding a maximum matching between points and intervals.

Now, a greedy algorithm suffices for a maximum matching. Sort the points by their x-coordinates. For each point, you can try to match any unmatched interval with the smallest possible endpoint. By sweeping through the x-axis, and maintaining the set of matchable intervals in a priority queue, you can obtain an $O(n \log n)$ time algorithm.

Shortest solution: 1022 bytes