Problem A. Advertisement Matching

Problem author : Jaehyun Koo (koosaga)
First solver: japan02 : Shigemura, Kawasaki, Yui Hosaka (38 minutes)
Total solved team: 51

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 $0 \leq k \leq n$:
$\sum_{i=k+1}^{n} a_i + \sum_{i=1}^{m} \min(b_j, k) \geq \sum_{i=1}^{n} a_i$
$\sum_{i=1}^{m} \min(b_j, k) \geq \sum_{i=1}^{k} a_i$ $\square$

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. Cactus Competition

Problem author : Sunghyeon Jo (ainta)
First solver: USA1 : Kevin Sun, Scott Wu, Andrew He (140 minutes)
Total solved team: 5

Let's consider a simpler problem where the start point is set as $(1, 1)$, and the end point is set as (N, M). There are four possible cases where no path exists:

- Case 1. $A_i + \max B_j < 0$ for some row $i \in [n]$ (Row is blocked)
- Case 2. $\max A_i + B_j < 0$ for some column $j \in [m]$ (Column is blocked)
- Case 3. There exists $x \in [n], y \in [m]$ such that $A_i + B_y < 0$ for $1 \leq i \leq x$, $A_x + B_j < 0$ for $1 \leq j \leq y$ (Start point is blocked)
- Case 4. There exists $x \in [n], y \in [m]$ such that $A_i + B_y < 0$ for $x \leq i \leq n$, $A_x + B_j < 0$ for $y \leq j \leq m$ (End point is blocked)

It is clear that no path exists when one of the following holds. Now we will show this is also sufficient.

By Case 1 and 2, $\min A_i + \max B_j \geq 0, \max A_i + \min B_j \geq 0$. Thus, in the row with maximum A_i, and in the column with maximum B_j, all cells are reachable. Thus, we can assume that all paths have to pass a cell with maximum A_i and B_j, which means we should be able to reach the blue cells from start, and we should be able to reach the end from some yellow cells in the figure.

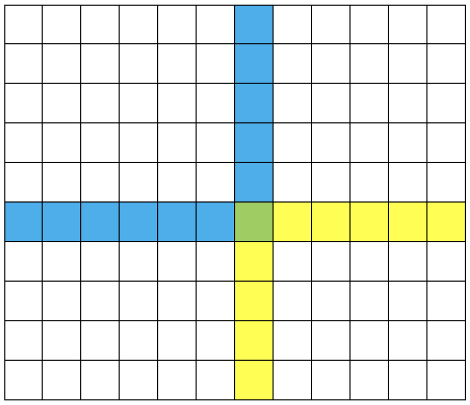

Figure 1.

We will now focus on reaching blue cells from the start (other case can be done identical). Let (p, q) be the cells with maximum A_i and B_j. If the $(p-1) \times (q-1)$ grid contains a row and a column that can't be passed, then we can not reach the blue cells (Figure 2). Otherwise, we have a row or column such that all cells are reachable (Figure 3). You can show this by the argument we used to find (p, q).

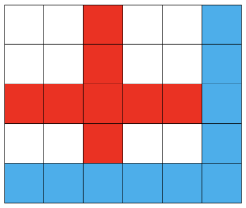

Figure 2.

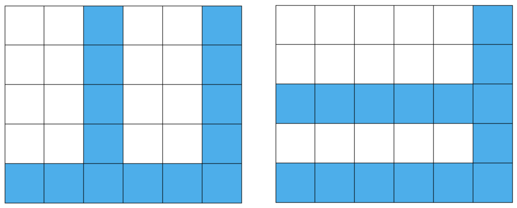

Figure 3.

The case in Figure 2 contains a pattern described in Case 3. So, if such pattern does not exist, We can recursively reduce the grid like Figure 3, until we cover the cell $(1, 1)$ or find the forbidden pattern, thus finishing the proof.

As the condition of reachability is reduced to much simpler ones, let's try to convert each cases into a condition for a valid pairs.

- Case 1 is obvious: if $A_i + \max B_j < 0$, all pair with $1 \le s \le i \le e \le N$ is impossible.
- For case 2, we only have to consider j with minimum B_j. Compute the list of consecutive intervals $[l_1, r_1], [l_2, r_2], \ldots, [l_k, r_k]$ for i that satisfies $A_i + \min B_j \le 0$. If both start and end points belongs to a single interval, then the grid induced by it have a blocked column.

- For case 3, Fix the x and extend rightward until you reach $A_i + B_j \geq 0$. Now, we only have to care about the minimum B_k where $k < j$. Find this k with prefix minimums, then extend upward until you reach $A_i + B_k \geq 0$ (using appropriate data structures). Then we can see that no starting point can exist in that interval.
- Case 4 can be done similarly with Case 3.

Now, if you plot (s, e) into a 2D grid, you can see all restrictions form a rectangle. So, the number of impossible (s, e) corresponds to an area of the rectangle, which can be computed with sweep lines and segment tree. Note that lots of small optimization is possible: Our shortest solution uses no data structures at all. Details are left as an exercise for interested readers.

Shortest solution: 1190 bytes

Problem C. Economic One-way Roads

Problem author : Jaemin Choi (jh05013)
First solver: Waterloo U : Gainullin (79 minutes)
Total solved team: 12

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. Just Meeting

Problem author : Jeyeon Si (tlwpdus)
First solver: Almost Retired Dandelion : Danilyuk, Kalinin, Merkurev (22 minutes)
Total solved team: 79

As the structure implies, let's try to formulate the condition as a graph-theoretical way. A meeting is just if $C(i, j) \geq min(C(i, k), C(k, j))$. If we consider a complete graph with weight $C(i, j)$, we can see that $C(i, j)$ should be at least the minimum edge value in a path $\{i, k, j\}$. By putting few more terms (like replacing $C(i, k) = min(C(i, l), C(l, k))$, we can see that $C(i, j)$ should be at least the minimum edge value in **all simple paths** between i, j. This minimum edge value can be computed by taking a maximum spanning tree with edge cost $C(i, j)$.

This hints the solution based on a maximum spanning trees, so let's try something related. Consider the input as a weighted graph, take the maximum spanning forest. Clearly, if there exists an edge (A_i, B_i) such that the minimum edge value in the path $A_i - B_i$ in spanning forest is greater than D_i, then it induces an unjust meeting. In this case, we report that it is impossible. This condition can be computed while computing the MSF itself. Group all edges with the same value, and check if there exist an edge that connects the same component, given that the edges with strictly larger weight are already added into the DSU.

Now, we claim that we can assign a valid $C(i, j)$ if such case does not exist. Connect all the components of spanning forest by edges of arbitrary weight, and let $C(i, j)$ be the minimum weight value for the path $i - j$ in tree. Then it's easy to see that all the meetings are just, and no condition in the input is violated. Since we have just set $C(i, j)$ to its lower bound, the answer is already the minimum possible, and as the dummy edges can take arbitrary weight, we don't have any reason to not set it 1.

Finally, it remains to find the value $\sum_{i=1}^{N} \sum_{j=i+1}^{N} C(i, j)$ efficiently. This can be also done while computing the MST, because if an edge connects the two component, any path between those two component have $C(i, j)$ as an added edge value. So, maintaining a size of component in DSU is sufficient for computing this.

Shortest solution: 1136 bytes

Problem E. Chemistry

Problem author : Jaehyun Koo (koosaga)
First solver: ext71 : Mingyang DENG, Yuhao DU (29 minutes)
Total solved team: 18

A chain is a tree that have no vertices with degree greater than 2. A tree is a graph that satisfies more than two conditions below:

- Connected.
- Acyclic.
- $V - E = 1$.

Let's take a condition that seems easier than others. But since we are not sure, let's just consider the underlying graph is a tree, which automatically satisfies the second condition.

We want to calculate the number of interval $[L, R]$, which induces a graph which

- Every vertices have degree at most 2.
- $V - E = 1$.

For the first condition, we can use two pointers. Let $f(i)$ be the maximum j such that there exists a vertices with degree at least 3 in interval $[i, j]$. If such doesn't exist, let $f(i) = N+1$. It's obvious to see $f(i) \leq f(i+1)$, so we can maintain the degree of vertices $[i, f(i)]$ in array and increase the right endpoint until degree 3 vertices are found. Thus $f(i)$ could be computed in linear time.

For the second condition, some ideas can result in segment tree solution. Let $EC[i][j] = (j - i + 1) -$ (the number of edges completely inside interval $[i, j]$). We want to compute the number of $i \leq j$ such that $EC[i][j] = 1$. As the edge (x, y) adds -1 to the rectangular range $[1, x] \times [y, n]$, The difference of $EC[i][*]$ and $EC[i + 1][*]$ can be described as some number of range updates, which means it's tempting to use segment tree as an underlying data structure to maintain EC.

The challenge here is to count the number of entries that have exactly some value. However, note that $V - E \geq 1$ for all acyclic graphs: Thus, $EC[i][j]$ can never go below one, and hence it is sufficient to maintain the minimum and it's occurence to compute the number of entries with $EC[i][j] = 1$, which is a standard problem solvable with lazy propagation.

The first condition is very simple, so it's very easy to combine the both condition: You can simply query in range $[i, f(i) - 1]$ instead of $[i, N]$, which gives an $O((n+m) \log n)$ solution if the underlying graph is a tree.

In a general case, we have to add the *acyclic* or *connected* condition into the problem. Here we will consider the *acyclic* condition. Let $g(i)$ be the minimum j such that $[i, j]$ induces a cycle. If such doesn't exist, let $g(i) = N + 1$. If we can compute the $g(i)$, we can simply take the query interval as a $[i, min(f(i), g(i)) - 1]$: By construction, the condition $EC[i][j] \geq 1$ will hold for every position that we will query.

Now our final challenge is to compute $g(i)$, where we are aware of two kinds of solution. Note that both utilize the condition $g(i) \geq g(i + 1)$ which is trivial.

- **Knowledge oriented solution.** We use two pointers just like how we computed the value $f(i)$, but our underlying data structure should be able to detect cycles, add or remove edges. Here, you can simply use the Link Cut Trees, a Swiss army knife for maintaining dynamic graphs. This solution is easy to both come up and implement, but requires quite a complicated data structure: Good prewritten codes or libraries will help. The time complexity is $O((n + m) \log n)$.
- **Ad-hoc solution.** Use divide-and-conquer as per the context of *Divide and conquer optimization*, which I assume the reader is familiar with. Let $f(L, R, L_{\text{opt}}, R_{\text{opt}})$ be a recursive function that computes $g(L) \ldots g(R)$ given that $g(L), \ldots, g(R)$ lies in $[L_{\text{opt}}, R_{\text{opt}}]$. Here, we have an invariant that the edges in interval $[R, L_{\text{opt}}]$ are stored in an appropriate data structure. Let $M = (L + R)/2$, and brute force to find $g(M)$ by exploiting the fact that $[R, L_{\text{opt}}]$ are in the data structure. After finding $g(M)$, undo the changes done in the DS, and recursively call $f(L, M - 1, L_{\text{opt}}, g(M))$, $f(M + 1, R, g(M), R_{\text{opt}})$. If you take the underlying data structure as undoable union-find (where path compression is not used), and carefully brute force to find $g(M)$, you can solve this problem with $O((n + m) \log^2 n)$. There are surprisingly many technical details: For example, naively iterating through adjacency list will mess up the complexity analysis. Good luck!

Shortest solution: 4012 bytes

Problem F. Interval Graph

Problem author : Jaehyun Koo (koosaga)
First solver: japan06 : Kobayashi, Imanishi, Kamiya (20 minutes)
Total solved team: 30

The set of interval is acyclic if and only if each point is covered by at most 2 intervals. The only if part is easy, because if there exists a point that is covered by 3 intervals, then they form a cycle. For the if part, consider a set of intervals that forms a cycle. Take the interval that have the least endpoint: Two different intervals that are adjacent in the cycle, meets at this endpoint, thus finding a point covered by at least 3 intervals.

Thus, we have to find the maximum weight interval set that covers each point at most twice. If this number is one, then we have to find maximum disjoint interval set. This problem can be solved by dynamic programming, where $dp[i]$ be the maximum weight when we consider intervals with endpoint at most i.

Equivalently, we can consider this as the minimum weight path in the DAG, where:

- There exists an arc from $i-1$ to i with weight 0.
- There exists an arc from $s_i - 1$ to e_i with weight $-t_i$.

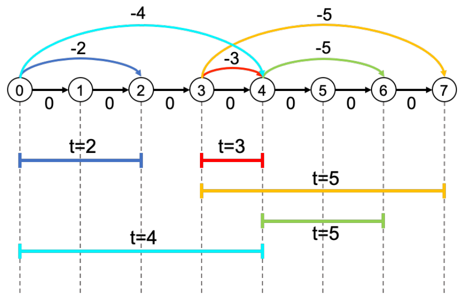

Figure 1. The DAG induced by the set of intervals.

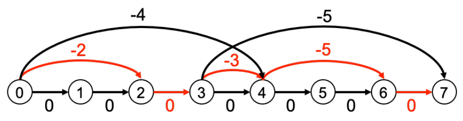

Figure 2. Shortest path in the DAG.

Let's go back to the original problem. If we have to cover each point at most twice, we should pick two **edge disjoint paths** in the DAG that have minimum sum of weights. A set of interval covers each point at most K times if and only if it can be partitioned into K disjoint set of intervals. You can prove this by greedy algorithm or using the fact that interval graphs are perfect. As each path corresponds to a disjoint set of intervals, we can see that the problem is about finding two edge disjoint paths. Finding two edge disjoint paths are the standard min-cost flow problem. Typical implementation of MCMF usually involves Bellman-Ford or SPFA to find a shortest path in a graph with negative edge weights, which results in $O(N^2)$ time algorithm.

To optimize this, we use the *potential method* technique. Let $P(i)$ be the length of shortest path from source, in the original DAG. This can be computed with the DP we have discussed earlier. For an arc $u \to v$ with cost c, let's assign a new cost $c' = c + P(u) - P(v)$. As $P(v) \le P(u) + c$ by the definition of shortest paths, for all non-negated arcs $c' \ge 0$. For negated arcs, they lie in the shortest path, so $P(v) = P(u) + c$ also by the definition, which means $c' = 0$. This new cost function is always nonnegative, so by using Dijkstra's algorithm, we can find a path that minimizes $D(source, sink) + P(source) - P(sink)$. We can see that the distance between source and sink is also minimized and can be easily traced.

This yields the time complexity to $O(n \log n)$. Note that the potential method is a rather generic technique, and it can be used to speed up any kind of MCMF instance by reducing the required number of Bellman-Ford computation into one. In other words, if you have good MCMF library, you can get easy AC by simply replacing the initial SPFA to DP.

Shortest solution: 1196 bytes

Problem G. LCS 8

Problem author : Jaehyun Koo (koosaga)
First solver: XZ Team (58 minutes)
Total solved team: 27

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 $K \leq 3$. 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 $|i - j| \leq K$. 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 $dp[i][i] \geq i - K$ 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 H. Alchemy

Problem author : Sunghyeon Jo (ainta)
First solver: ETH Zurich Daniel (21 minutes)
Total solved team: 94

Let's think about a decision problem, where we decide whether we can obtain a single element with value k. We will think the process in reverse: We will replace the value k into $\{0, \ldots, k-1\}$ or such, and obtain the multiset C that is given as input. The two possible operations are:

- Type 1. $\{i\} \rightarrow \{$one or more elements for each value less than i, zero or more elements greater than $i\}$.

- Type 2.$\{0\} \to \{$zero or more elements greater than 0$\}$.

Intuitively, it seems wasteful to make unnecessary elements to fit into C. Indeed, you can transform the problem into creating a *subset* of multiset C, and then you don't have to create unnecessary elements except some special cases. If $k > 0$ and you did at least one operation, then the first operation should be type 1. We can create dummy 0 at that time, and expand this by type 2 operation.

So now the operations can be described in simpler way.

- $\{i\} \to \{0, 1, \ldots, i-1\}$ $(i > 0)$
- $\{0\} \to \{i\}$ $(i > 0)$

You can prove by the exchange argument, that all type 1 operation preceeds type 2 operation in some solution. Now, we are ready to design an actual algorithm.

We will maintain a multiset D which is supposed to be a subset of C. Given the single element k, decide whether this can fit in D. If it does not, replace it to $\{0, 1, 2, \ldots, k-1\}$. Then now decide this for $k-1$, and if it fails then replace it. If you do it for $k-2, k-3, \ldots$, you will have several number of zeros, which you can then compare with the cardinality $|C| - |D|$. The current set can be maintained by a simple counter, since all elements have same occurences by construction.

Finally, it can be proven that the decision problem is monotone (if answer exists for $k+1$ then it exists for k), and $k \le n + 100$. Thus, a binary search suffices for maximizing the k. You should be careful of the corner case mentioned earlier, but hopefully they are all described in a sample.

Shortest solution: 881 bytes

Problem I. Query On A Tree 17

Problem author : Jaehyun Koo (koosaga)
First solver: hanoi20 (85 minutes)
Total solved team: 13

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 J. Remote Control

Problem author : Jaemin Choi (jh05013)
First solver: Polish Mafia : Nadara, Sokolowski, Radecki (28 minutes)
Total solved team: 95

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 K. Sewing Graph

Problem author: Donghyun Kim (kdh9949)
First solver: NRU HSE : Romanov, Safonov, Anoprenko (9 minutes)
Total solved team: 100

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).

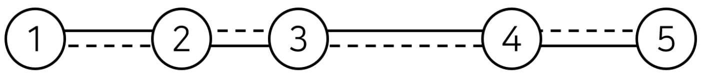

Figure 1. Strategy for dots on a straight line.

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).

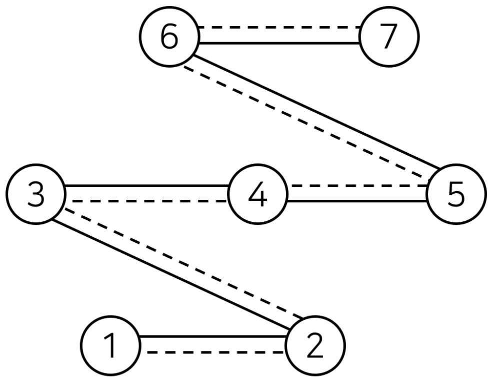

Figure 2. Strategy for general case.

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 L. Steel Slicing 2

Problem author: Jaehyun Koo (koosaga)
First solver: ext71 : Mingyang DENG, Yuhao DU (142 minutes)
Total solved team: 9

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