Spring 2022 mBIT Advanced Editorial

May 22, 2022

This editorial provides our solutions to each problem as well as accepted programs in each supported language. In some cases, the given programs may use different algorithms than the one described in the editorial. For more complex problems, multiple solutions and programs may be given. Our problems are likely to have solutions which are not covered here and we would be interested to hear about any such solutions the reader may devise.

Contents

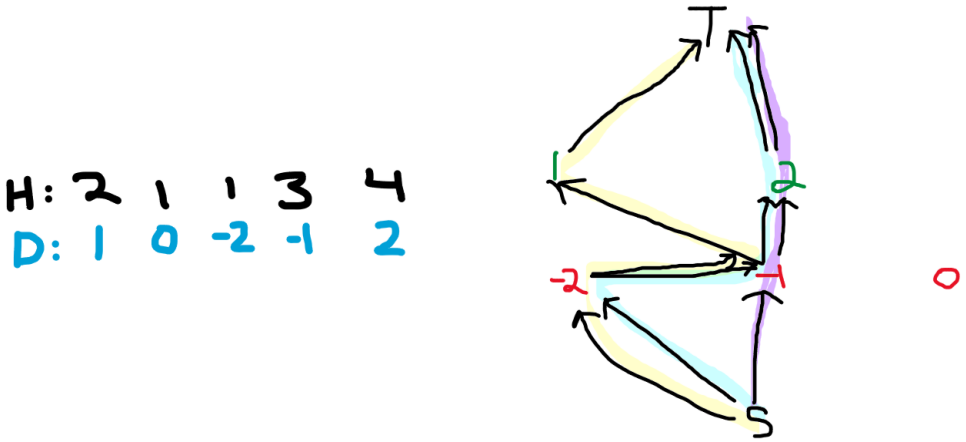
§1 Boardwalk

Consider the differences between consecutive elements (0-indexed): $D_i = H_i - H_{(i+1)\%N}$ $(0 \le i \le N-1)$. The ith operation is equivalent to to incrementing R_i and decrementing $(L_i - 1)\%N$. We want to find the minimum cost of making $D_i = 0$ for all $0 \le i \le N$.

Consider a graph with each D_i being a node, plus a source node S and sink node T. We represent an operation as a directed edge $R_i \to (L_i - 1)\%N$ with cost C_i. Also, let $F = \sum_{D_i>0} D_i = \sum_{D_i<0} -D_i$.

Consider the edges used any solution that equalizes H (not necessarily optimal), with each operation edge appearing the number of times it is used. If we add $-D_i$ edges from a S to every node i with $D_i < 0$, and D_i edges from every node i with $D_i > 0$ to T:

1. The N difference array nodes have indegree = outdegree.
2. S has outdegree F and T has indegree F.



Consider repeatedly DFSing from S, until you end at T, and extracting the path out of the graph. **1** is preserved while the out/in degree of S/T decreases by 1. This lets us prove inductively that the graph can be modelled as F edge disjoint paths from S to T.

If the maximum number of edge-disjoint paths from S to T is less than F, then there can't be a solution. Otherwise, we want to find the minimum sum of edge costs over F edge-disjoint path. We can do this with a (reasonably fast) minimum cost maximum flow algorithm.

Time complexity: $\mathcal{O}(NKlog(N^2 H_{max} C_{max}))$ (or whatever your MCMF complexity is)

Problem: Claire Zhang
Editorial: Claire Zhang
Flavortext: Claire Zhang
Code: C++, Java, Python

§2 Double Deletion

Let's solve a simplified version where you are allowed to perform double deletion on any node and direction.

Solution to simplified problem

Create an undirected graph G with $2N$ nodes: an i_{in} and i_{out} for each original node i ($1 \leq i \leq N$). Convert each input edge $u \to v$ into undirected edge $u_{out} - v_{in}$.

A double deletion translates to deleting any pair of edges from a node. Let's solve each component independently. We can achieve the maximum possible number of double deletions within a component with a greedy DFS algorithm (see this problem for an explanation). The minimum number of un-deleted edges is the number of components with an odd number of edges.

Solution to original problem

Again, create G and assume at the start that we can perform double deletion on every node and direction. Consider all $(u, \text{in/out})$ *not* given in the input as "restricted nodes" in G. Delete all edges connecting two restricted nodes. Then for each restricted node u connected to $v_1, \ldots, v_k$, erase $v_i - u$ and replace it with $v_i - fake_i$, where $fake_i$ is a new fake node (with degree 1). Now we can run the same greedy DFS algorithm on this modified G.

Time complexity: $\mathcal{O}(N)$

Problem: Claire Zhang
Editorial: Claire Zhang
Flavortext: Claire Zhang
Code: C++, Java, Python

§3 Mascot Parade

The mascots will travel in clumps. We will call the furthest right mascot in a clump the "leader" since the entire clump travels at their speed. Initially all mascots are their own clump lead by themselves. Once a mascot catches up with the mascot in front of it, the two mascots merge and become a clump. When the leading mascot in a clump catches up with the trailing mascot of the clump in front of it, the two clumps become one big clump. Now notice that throughout the passage of time, this ends up forming a forest. When a clump lead by mascot x catches up to and merges with a clump lead by mascot y, we say that x's parent is y in the forest. If we can generate this forest, it will give us all necessary information on the leaders of each clump over time.

We can form said forest with a priority queue. The priority queue will store the would-be times of the merge between consecutive clumps if they were in isolation. We ignore the times of merges that never happen in isolation. Initially, we put the times of merge between consecutive mascots into our pq . Until there is nothing left in our pq, we pop out the soonest merge that would happen from our pq. This is the soonest real merge because until a merge happens, everything happens the same as it would in isolation.

Let clump X be the clump merging into clump Y. Let the leader of clump X be mascot x and the leader of clump Y be mascot y. First, we execute the merge by setting the parent of x to be y and storing the time of merge. Then we need to recalculate the new time of merge for the clump behind clump X into clump x (which is now part of clump y). We use the speed and positions of mascot y to calculate this then we add this new time into the pq. We can find the time of merge between two mascots in isolation by taking the amount that the mascot behind needs to catch up (difference in positions minus the difference in indices) divided by the differences in speed. Also note we need to store the trailing mascot in each clump for this to work.

Now that we have the forest, we can answer the queries. For query i, we want to find the leader of the clump that mascot A_i is in at time B_i. This is equivalent to the highest ancestor of A_i that merges before time B_i. We can use binary lifting to find this node, and then a binary advancement up the tree to find this. Let mascot h be this highest ancestor that merges before time B_i. The final answer will be the position of mascot h $(P_h + S_h B_i)$ minus the differences in indices $(h - A_i)$ because they are in the same clump.

Time complexity: $\mathcal{O}((N+Q)\log N)$

Problem: Daniel Wu
Editorial: Daniel Wu
Flavortext: Daniel Wu
Code: C++, Java, Python

§4 Memorable Attractions

Treat the K constraint as just another distance constraint. Let's ignore the $P_i \neq -1$ constraints for now and focus on satisfying the distance constraints.

Create a graph with $N+1$ representing $P_0, \ldots, P_N$ and add edges $P_{A_i} - P_{B_i}$ with weight D_i. If we run Dijkstra from node 0, then $P_i = dist(i)$ satisfies all distance constraints.

Now let's consider the locations we're already given. Our final P_i must be equal to or less than $dist(i)$ (proof: consider the edges on the shortest path from 0 to i). Therefore, if we add the edges $0 - i$ with weight P_i and then run Dijkstra, $dist(i) < P_i$ implies the configuration is impossible. Otherwise if $dist(i) = P_i$ for all i with $P_i \neq -1$, we can use $dist(i)$ $(1 \leq i \leq N)$ as our answer.

Time complexity: $\mathcal{O}(Nlog(N+M))$

Problem: Claire Zhang
Editorial: Claire Zhang
Flavortext: Claire Zhang
Code: C++, Java, Python

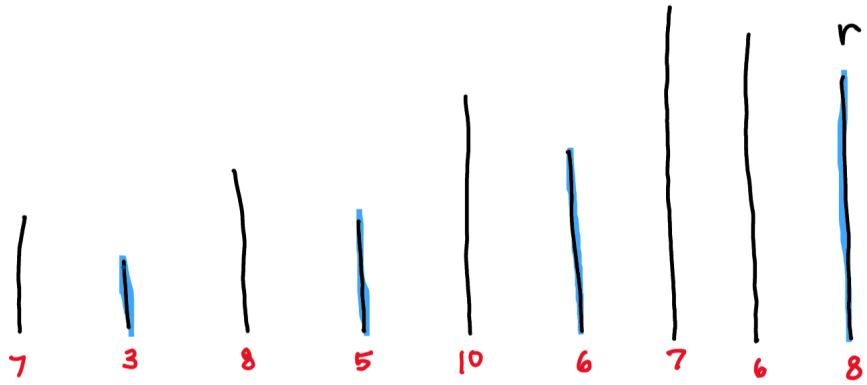
§5 Monkey on a Ladder

We sweep from left to right and process queries at their right endpoint. Consider the path the monkey takes to get from (l, a) to (r, b). For now, pretend $a = 1$ and $b = \infty$.

First, the monkey would never go down (both straight down and diagonally down).

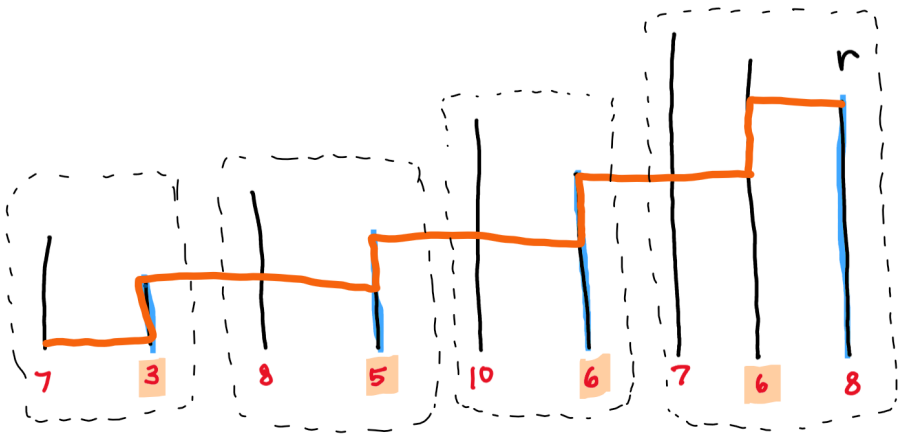
Second, the costs of the ladders on the path must be non-decreasing, with the last ladder being r. Thus we can disregard ladders that aren't suffix minimums by height.

Now consider the set of ladders which are suffix mins by both height and cost; call them *special*. In the diagram below, the costs are labelled in red and the special ladders are highlighted in blue:



When travelling from the top of a special ladder i to the top of the next special ladder, say j, the cost is $(H_j - H_i) \cdot \min_{l<k\leq r} C_k$. This is achievable because all ladders in that range are higher than or equal to H_j.

We can consider the special ladders as the ends of blocks, with the cost of a block being the minimum cost of in-between ladder and the height being the height of the last ladder in the block. In the diagram below, blocks are dotted, the cost of each block is highlighted in orange, and the optimal path (from the ground to ∞) is drawn in orange.



We will use these blocks to answer queries. First, for each query, let's move the left ladder, as needed, to the ladder with minimum height between l and r (if you start higher

than this height, you're just going to be forced downwards until you reach it, then start climbing up.)

As we sweep, we maintain the special ladders in a monotonic (2-way) stack.

We can decompose the path from l to r as (1) going to the nearest special ladder L, (2) travelling between special ladders until you reach the last special ladder R before r, and (3) going to r. We can binary search for L and R. The cost of (2) can be maintained as we sweep using prefix sums. (1) and (3) can be calculated separately, requiring us to find the minimum-cost ladder between $[l, L]$ and $[R, r]$. We must take care to account for the desired starting and ending heights (a and b). See code for this and other details.

Note: There is also a $O((N+Q)\sqrt{N}$ solution which involves Mo's algorithm and combining convex functions. We tried to block these solutions though.

Time complexity: $\mathcal{O}((N+Q)log(N))$

Problem: Claire Zhang
Editorial: Claire Zhang
Flavortext: Claire Zhang
Code: C++, Java, Python
Code (sqrt): C++

§6 Not Nim

The first key observation is that we can take the number of stones in each pile $\bmod(A+B)$. The logic goes as follows: Let the winner of the $\bmod(A+B)$ case be player X and the loser be player Y. Player X's strategy will be as follows to always win: If it is the first turn or if player Y just took from a pile that previously had less than $A+B$ stones, then he moves as if all the piles were $\bmod(A+B)$. Otherwise, if player Y just took from a pile that had $A+B$ stones or more, he mirrors the move and takes from the same pile. This strategy always wins because the mirrored moves on piles that had $A+B$ stones or more are essentially ignored when we look at it through the lens of $\bmod(A+B)$ (because a total of $A+B$ stones are taken from the same pile and the turn doesn't switch), and player X wins when he plays optimally looking through the lens of $\bmod(A+B)$, by definition. This means we can take all piles $\bmod(A+B)$ and determine the winner from there.

The next observation is that the optimal strategy for both players is greedy. Since all piles of less than $A+B$ stones, when one player takes from a pile, it "blocks off" any moves by the other player on that pile. This means when one player takes from a pile they "reserve" it for themselves and should take from it when they can't take from any other pile. Now on a player's turn, he should take from the biggest pile because it will "block off" the most number of "reserved moves" the other player would get and they will "reserve for themselves" the most number of moves to fall back on when all piles have been taken from.

Now the full solution is once we have taken everything $\bmod(A+B)$, we simulate the greedy process using a priority queue.

Time complexity: $\mathcal{O}(N \log N)$

Problem: Claire Zhang
Editorial: Daniel Wu
Flavortext: Claire Zhang
Code: C++, Java, Python

§7 Park Passes

Notice that the process of choosing which packages to buy seems recursive. This motivates dynamic programming. Let DP_{ij} be the minimum cost of tickets to cover days 1 through j while only using the first i ticket packages (packages sorted in increasing order of expiration date) or 10^{18} (an arbitrarily large number) if the first i ticket packages don't suffice. Our transition is as follows:

If $E_i < j$, $DP_{ij} = 10^{18}$
If $E_i \geq j$, $DP_{ij} = \min(DP_{i-1j}, DP_{i-1j-T_i} + C_i)$

The first case is obvious; if the latest-expiring package expires before a day that you need a ticket, the packages don't suffice. The second case is split into two subcases. The first case is if you don't use package i, in which case you simply take the DP value that requires the same number of days but doesn't use package i. The second case is where you do use package i, in which case you take the DP value that asks for T_i less days (because we should use these tickets on the latest days we can since this package expires the latest) and doesn't use package i anymore (you can't use a package twice).

If $DP_{NM} = 10^{18}$, the answer is -1, otherwise the ticket packages are enough to cover all m days so we should print the value stored in DP_{NM}.

Time complexity: $\mathcal{O}(NM)$

Problem: Daniel Wu
Editorial: Daniel Wu
Flavortext: Daniel Wu
Code: C++, Java, Python

§8 Rating Report

The hierarchy forms a tree; supervisors are parents, subordinates are children, and first-hand raters are leaves. We can binary search for answer. Notice that if we are trying to check whether the final result can be greater than or equal to some m, all we care about is whether each leaf gives a score of at least m. This means we can assign every value a 0 or 1, depending on whether they give a rating of at least m. Now we can do a tree dp. Let DP_i be the minimum number of ratings of 1 among the leaves of i's subtree necessary to make reviewer i have a rating of 1. Let cnt_i be the number of decided leaves who gave a rating of 1.

If rater i is a pessimist, DP_i = the sum of DP_x for all children x of i.
If rater i is an optimist, $DP_i = \min(DP_x - cnt_x) + cnt_i$ for all children x of i.

The logic goes as follows: If rater i is a pessimist, then all of reviewer i's children must submit a rating of 1 in order for reviewer i to submit a 1. Since the children are independent, this answer is simply the sum of the DP's of the children. If rater i is an optimist, then only one of reviewer i's children must submit a rating of 1 in order for reviewer i to submit a 1. We want to waste the least number of ratings of 1 on undecided leaves for the child that does submit a rating of 1. This value is the minimum (total leaves that must submit a 1 (DP_x) - predecided leaves that submit a 1 (cnt_x)) out of all subtrees of children. For the other children, we will make undecided leaves in their subtrees be 0 because it doesn't matter. Now, we need to add back all predecided leaves of rating 1 in i's subtree, which is cnt_i by definition.

Now it is possible to achieve a final rating of 1 if and only if $DP_1 \leq K - m + 1$ because there are $K - m + 1$ first-hand raters who will give a rating of 1 in reviewer one's subtree.

Time complexity: $\mathcal{O}(N \log K)$

Problem: Gabriel Wu
Editorial: Daniel Wu
Flavortext: Daniel Wu
Code: C++, Java, Python

§9 Snapshot

Consider the list formed by merging the lists of the people's initial locations and their locations in the first snapshot and then sorting in non-decreasing order. If the first element of this list corresponds to a person's initial location, this person must be travelling in the positive direction (otherwise the snapshot location would be less than the initial location so the initial location wouldn't appear first in the list). If the first element of this list corresponds to a person's location in the snapshot, that person must be travelling in the negative direction (otherwise the snapshot location would be greater than the initial location so the snapshot location wouldn't appear first in the list).

After deducing the direction of this first person, we no longer need to consider them in our process, so we can delete both their initial and snapshot locations from the merged list. Now we can repeat the process to get the direction of another person, and then another person, so on until we have every person's direction. If there is ever a time when the location needed to satisfy the complementary timestamp of the person corresponding with first element of the merged list isn't in the list (if the person associated with the first element doesn't have their corresponding initial location and snapshot location in the list), the snapshot was tampered with.

Now that we have the direction of every person, we can check every other snapshot and see if they are consistent. We use the time of the snapshot to generate what the locations of the people should be, sort the list, then check if this list is the same as the one given. If any snapshot is inconsistent with the directions generated at the beginning, the snapshots were tampered with. Otherwise, they are consistent so we print the directions.

Note: since actually deleting elements from the merged list at the beginning is slow, instead mark them as "used" and only consider "unused" elements after that.

Time complexity: $\mathcal{O}(NK \log N)$

Problem: Claire Zhang
Editorial: Daniel Wu
Flavortext: Daniel Wu
Code: C++, Java, Python

§10 Sugar Rush

At any point your visited stands will be a consecutive interval (subarray) of the stands. This is because you will never go past a cotton candy stand without visiting it. Now, we claim that you will change directions (from travelling right to travelling left or vice versa) at most $\log_2 max(x)$ times.

The logic for this goes as follows: let your current interval of visited stands be from stand L to stand R and let stand m be the stand you started at. If you were to switch directions from right to left, it would mean that $X_{R+1} - X_R < X_R - X_{L-1}$ so the difference between A_m and A_R would at least double when X_{R+1} gets visited which will happen during the next direction change. You can follow the same argument for switching from left to right.

Now we just need a fast way to calculate the stands in which the direction changes take place. This requires some fidgeting with inequalities. Assuming your current interval is from stands L to R and you are travelling right, the next direction change will take place after visiting stand P where P is the first index greater than R in which $X_{P+1} - X_P < X_P - X_{L-1}$. This can be arranged to $X_{P+1} - 2X_P < -X_{L-1}$ which is the same as $2X_P - X_{P+1} > X_{L-1}$.

Knowing this, we can find P quickly by using binary search on range max queries on a sparsetable of values of $2A_i - A_{i+1}$. You can follow the same process for the other direction change to get that the direction change happens on the latest index P less than L in which $2 * X_P - X_{P-1} \geq X_{R+1}$ and this time, do binary search on range max queries on a sparesetable of values $A_{i+1} - 2A_i$.

Now for each starting location, we set the initial interval to be [i,i] then alternate extending the left and right boarders until every stand is visited. We add up the range between boarders at each extension to find the total distance travelled.

Time complexity: $\mathcal{O}(N \log(X_i) \log(N))$

Problem: Claire Zhang
Editorial: Daniel Wu
Flavortext: Daniel Wu
Code: C++, Java, Python