Spring 2022 mBIT Standard Editorial

May 22, 2022

This editorial provides the intended solutions to each problem as well as accepted programs in each supported language. In some cases, the given programs may employ different algorithms than the one described in the editorial. For more complex problems, multiple solutions may be given, in which case there will be programs for each solution. Nevertheless, 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 Ticket Prices 2

2 Whac-a-Mole 3

3 Ferris Wheel 4

4 Behind The Scenes 5

5 Shifting Seats 6

6 Roller Coaster 7

7 Toadstools 8

8 Snapshot 9

9 Park Passes 10

10 Street Performers 11

11 Rating Report 12

12 Sugar Rush 13

13 Not Nim 14

14 Mascot Parade 15

§1 Ticket Prices

To find the cost of someone's ticket, check which age range they fall into and then use the corresponding price. Do this for Alice, Bob, and Charlie then add the prices up to get the total cost.

Time complexity: $\mathcal{O}(1)$

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

§2 Whac-a-Mole

Maintain an array A of length N representing which holes currently have moles in them. Let $A_i = 1$ if there is currently a mole in hole i and $A_i = 0$ if there is not currently a mole in hole i. Initially set A to be all ones.

For each of the Q updates, update the value at the given index in A, then check if the game ended. You can check this by either iterating over the entire array to see if every element is a zero, or by maintaining a counter with the current number of moles out. If the game ended, print the second this is on then break out of the process because the game is over. At the end, if the game never ended print -1.

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

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

§3 Ferris Wheel

Since the wheel spins at a constant rate, pausing after X seconds of spinning results in making $\frac{X}{K}$ rotations. Now notice you only care about the decimal portion of this number because complete rotations don't change final position.

Now we observe that the vertical height of your ending position only depends on how close you are to half of a rotation. Out of all pauses, the one whose equivalent decimal of a rotation is closest to 0.5 will result in the highest vertical height.

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

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

§4 Behind The Scenes

We can greedily go through the lengths in L and at each length, print the lowest lexicographic (earliest in a dictionary) unused string of that length. Generating the strings in this way allows us to systematically go through every string of each length before we have to overlap.

We can execute this process by maintain an array of strings "next" such that next[len] stores the lexicographic lowest unused string of length len. We initialize next[len] to be all 'a's. After using a string next[len] in our process, we need to update the string stored in next[len] to the next lexicographic string of length len.

We find this next lexicographic string by iterating through the letters of the string in reverse order. If the letter we are iterating at is 'z', we set this letter to 'a' and move to the letter to the left. If the letter we are iterating at is not 'z', we increment this letter and break out of the process. You can think of this process as adding 1 to a number in base 26.

Time complexity: $\mathcal{O}(max(L_i)^2 + sum(L_i)^2)$ or $O(max(L_i)^2 + sum(L_i))$

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

§5 Shifting Seats

Consider an array P where $P_i = A_i - i$ (0-indexed). It turns out that the problem of increasing elements of A to make A consecutive is equivalent to increasing elements of P to make all elements of P equal. This is because if $P_i = A_i - i = X$ for all i and a constant X, then $A_i = X + i$ for all i. $A_i = X + i$ means A is consecutive.

Once we generate the array P, our task is to minimize the number of seconds to make all elements of P equal. The ending value of every element of P ends up being the biggest number in the initial array of P. This is because if it weren't the case, every friend would have moved up at least one seat so we could make everyone move up one less seat and have a more optimal answer. Now our process is to increment every element of P that is not equal to max(P) until every element is there. This will take max(P)-min(P) seconds because you will be held back by the smallest value in P.

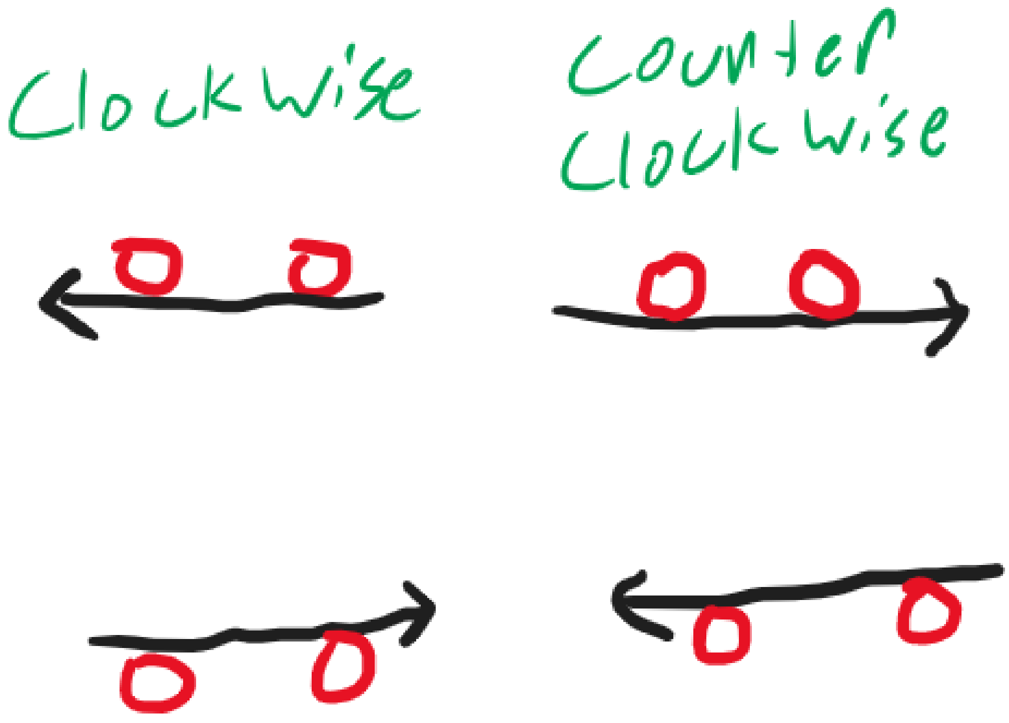
Time complexity: $\mathcal{O}(N)$

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

§6 Roller Coaster

There are two cases: the train's wheels are always clockwise to it's direction or always counterclockwise to it's direction. The orientation is uniquely determined by the direction the train starts travelling in because the problem ensures the train starts on top of the track. This means a train initially facing left is clockwise and a train initially facing right is counterclockwise.

From that, we observe that wheels clockwise of right will be underneath the track and wheels counterclockwise of left will be underneath the track. Now, if we're clockwise we simply check if there is ever a time we are facing right and if we're counterclockwise we simply check if there is ever a time we are facing left.



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

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

§7 Toadstools

We can use a two pointer's approach to iterate over all pairs of toadstools. We will have a first pointer iterate from $L = 1$ to $L = N$ and at each L we will have a second pointer iterate from $R = L + 1$ to $R = N$. Over our iteration of R, we will keep track of the toadstool S with the greatest slope from L that we have iterated over so far.

The toadstool at any R can only be seen if the slope from L to R is greater than the slope L to S. This is because toadstool S obstructs the view of anything under the line of sight from L to S. However if the slope to R is greater than the slope to S, it means the line of sight to R is above the line of sight from L to S. In this case we add one to our answer and replace S with R because it is the new highest sloped toadstool.

Time complexity: $\mathcal{O}(N^2)$

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

§8 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

§9 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

§10 Street Performers

First, we can observe that we should greedily put off moving until we are forced to move. This motivates the following greedy algorithm: repeatedly move to the spot that maximizes the number of performances you can watch, watch all of these performances, and then move again to the place that maximizes the size of the next batch of watchable performances. We can do this until we have watched every performance.

This is clearly very recursive, so we should use dynamic programming. Let DP_i be the answer for the suffix that starts at performance i. Now $DP_i = DP_{X_i+1} + 1$ where X_i is the maximum index such that all performances from i to X_i are within K of each other. Now, our task is to find X_i fast.

Let's iterate backwards from $i = N$ to $i = 1$ (1-indexed). We will keep track of X by maintain a set S of the locations of all performances from i to X_i. When we move to index i, we insert A_i to S. Now While the range of S is greater than K, it means X is too far so we delete A_X from S, and decrement X. We end up with the greatest X that has a corresponding S such that you can see all performances in S from one location. At the end, we simply print out the values of our DP.

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

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

§11 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

§12 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

§13 Not Nim

The first key observation is that we can take the number of stones in each pile $\text{mod}(A+B)$. The logic goes as follows: Let the winner of the $\text{mod}(A+B)$ 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 $\text{mod}(A+B)$. 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 $\text{mod}(A+B)$ (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 $\text{mod}(A+B)$, by definition. This means we can take all piles $\text{mod}(A+B)$ 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 $\text{mod}(A+B)$, we simulate the greedy process using a priority queue.

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

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

§14 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