mBIT Advanced Editorial

June 2020

These are the solutions to the advanced problems. Each answer consists of a brief explanation of the solution followed by a link to our code in each language. Keep in mind that there are multiple ways to do each problem, and the given code may employ a different algorithm than the explanation.

Contents

§1 Zoo Tour

The key idea to this problem is that there are two ways to go around the circular neighborhood to get from one habitat to another. Thus, we can use prefix sums to solve this problem. We will index the habitats from 0 to $N-1$. Before taking the queries, preprocess the data by constructing a prefix array pre such that for $0 \leq i < N$, $pre[i]$ is the distance from habitat 0 to habitat i traveling clockwise and $pre[N]$ is the length of the entire loop, using all the paths.

For each query, arrange u and v such that $u < v$. Then, if the veterinarian travels clockwise, the distance traveled (call this d) is $pre[v] - pre[u]$. If he travels counterclockwise, he will travel through every path except the ones in the clockwise route, so the distance will be $pre[N] - d$. The answer to that query is the minimum of those two distances.

Time Complexity: $O(N + Q)$

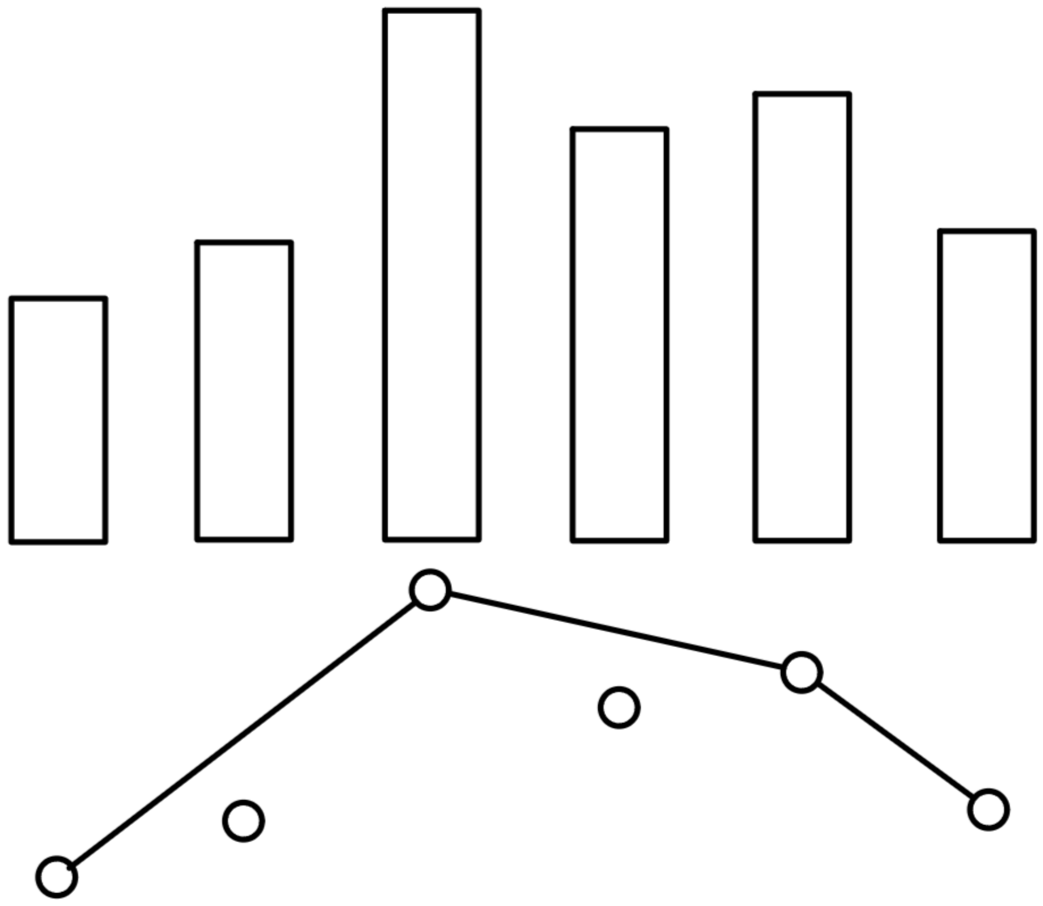
Problem: Aaron Mei
Editorial: Aaron Mei
Solutions: C++, Java, Python

§2 Leaping Lizards

The intended solution of this problem is to reduce it to a graph. We can loop through each pole and determine which poles it may form an edge with. The input size suggests this may not be done naively, so we must maintain the greatest slope we have seen and make sure to only create an edge when the current slope is greater than or equal to the greatest slope we have seen. After we have this graph, we can run a BFS while keeping track of distances to determine the minimal distance to the final pole.

Time Complexity: $O(N^2)$

Note: Using convex hull, the complexity can be further reduced to $O(N \log N)$. The trick here is that the minimum distance can be found by computing the convex hull with the tips of the poles. We can use proof by contradiction to show that the convex hull is indeed the shortest path. Assume that there is a shorter distance than the convex shape. However, in order to achieve this, you must go through an illegal pole (we can see this is true by looking at the slopes of the convex hull). Thus, we can add points below the first and last poles and run convex hull to find the number of poles the lizard need pass through. Then, we subtract out those two points and account for the points with the same slope to get the minimum distance. The diagram below provides intuition on why the convex hull is optimal.



Problem: Ayush Varshney
Editorial: Aaron Mei
Solutions: C++, Java, Python

§3 Raging Rhinos

We see that each interaction between rhinos uses exactly two rhinos. We may choose to look at each interaction from the perspective of either rhino, however it turns out that it is advantageous to only observe these interactions through either the right or left initial rhino for all interactions. For the sake of explanation, we shall look at the right initial rhino. This crucial observation reveals that we are able to simulate these interactions using a stack approach iterating over the given list from left to right.

In this stack, we apply casework before inserting the current rhino. If the rhino on the top of this stack faces right and our current rhino faces left, we have an interaction. While our current rhino can defeat the rhino at the top of the stack, we decrease our rhino's stamina and remove the rhino at the top of the stack. Once the current rhino is unable to defeat more rhinos (the stack is empty/the rhino at the top of the stack is facing left or the rhino at the top of the stack has more stamina than the current rhino) we either insert or do not insert the current rhino with its remaining stamina into the stack. After iterating over the whole list using this process, we can simply output the resultant stack.

Time Complexity: $O(N)$

Problem: Maxwell Zhang
Editorial: Ayush Varshney
Solutions: C++, Java, Python

§4 Raccoon Mischief

Let's examine what happens to an individual raccoon. Say that an individual raccoon is visited by Alice Y times with the values $x_1, x_2, ..., x_Y$, and assume that the raccoon begins with some non-zero amount of candy. On the first visit, Alice will take away the raccoon's candy. On the second visit, Alice will give it x_2 candy. On the third visit, Alice will take away the raccoon's candy again, and on the fourth visit she will give it x_4 candy. This pattern continues, and we can see that if the raccoon is visited an odd number of times, it will have no candy at the end, and if it is visited an even number of times, it will have x_Y candy at the end. The case where the raccoon begins with no candy is the same, except the parities are swapped. If the raccoon is never visited, their candy amount doesn't change.

Hooray, we've solved the problem for an individual raccoon! Now how do we generalize this to the entire array? We can use a sweep line approach. We store two dynamic arrays (`std::vector` in C++, `java.util.ArrayList` in Java, lists in Python) at each of the N positions named *add* and *remove*. add_i represents all queries beginning at position i, and $remove_i$ represents all queries ending at position i. For a given ith query, we insert a marker representing that query into add_{l_i} and $remove_{r_i}$. Each query is stored in two arrays, so $O(Q)$ memory is used overall.

Now we sweep from position 1 to N. Each time we encounter the start of a given query (the query is in *add*), we add it to a set with elements ordered by index of the queries (`std::set` in C++, `java.util.TreeSet` in Java, `heapq` in conjunction with boolean array for Python). Each time we encounter the end of a given query (the query is in *remove*), we remove it from the set. Conceptually, at position i the set holds all queries that affect position i, which are all queries that start before i and end after i. For each position, we look at the parity of the size of the set, and we change the array value to either the query value in the set with the highest index, 0, or leave it the same per the casework done in the first paragraph.

Time Complexity: $O(N + Q \log Q)$ - Each query is inserted and removed from the set once, and ordered set operations take logarithmic time.

Problem: Colin Galen
Editorial: Maxwell Zhang
Solutions: C++, Java, Python

§5 Turtle Tribulation

Note that we are given a list of coordinates on a plane. We know that every y value has at most one corresponding x value, but every x value does not necessarily have a single corresponding y value. This means that if we make a list of size Z for every y value (which is an N by Z grid), we are able to store all the possible positions for all points. At each position in this grid, we can store the minimum number of shifts necessary to arrive at that point from the given bottom-most point. The idea of having all path possibilities with a fixed starting position in a grid and picking one optimally with the least cost lends itself to dynamic programming.

However, applying a dynamic programming approach here is not enough. In the transition between states, we must add the least cost (the least number of shifts necessary to get to any point in the previous row, which satisfies the slope condition with respect to the current point, from the bottom-most point) to the cost of our current point. Without any optimizations, we would have a slow $O(NZ^2)$ solution since we would need to check the slope condition for every single point in the previous row (of which there are Z) for every single current point (of which there are approximately NZ).

We must realize that the absolute slope between two points, given a fixed change in y and a variable change in x, decreases as the change in x increases. So, forcing the absolute slope to be greater than or equal to K means that all points in the previous row of points, which satisfy the slope condition, have to lie in a range. Through some algebra we can determine the endpoints of this range and then determine the minimum cost within that range. One method to determine the minimum cost within that range is by creating a segment tree from all the costs of points in the previous row. This would allow us to apply a range minimum query on the range of viable previous points for the current point and obtain the minimum cost. After filling out our dynamic programming table, the final answer is simply the cost corresponding to the point at the top-most position since that point must remain fixed.

Time Complexity: $O(NZ \log Z)$

Note: It was also possible to obtain an $O(NZ)$ solution by applying a sliding minimum window using a deque instead of a range minimum query using a segment tree since the range of viable points in the previous row would shift by at most one point after incrementing the current point in the dynamic programming. You can read about it here.

Problem: Ayush Varshney
Editorial: Ayush Varshney
Solutions: C++, Java, Python

§6 Gorilla Grouping

If you think of each gorilla as a node of a graph, and you connect each pair of incompatible gorrillas with an edge, then the graph becomes a series of disjoint chains. This is because every node is connected to a maximum of two other nodes and there cannot be any cycles. The problem has now been reduced to counting the number of independent sets (subsets of disjoint nodes) in the graph. We can find the lengths of the chains by storing the IDs in a set or a map, then adding K to each ID over and over again until we get an invalid ID.

Since the chains of nodes in the graph are disjoint, we can count the number of independent sets in each one separately, then multiply the totals together. In a chain of length c, there are fib_c independent sets, where fib_i represents the ith Fibonacci number. This can be shown with induction: if the bottom node in the chain is included in the set then there are fib_{c-2} ways, otherwise there are fib_{c-1} ways.

By precomputing the first N Fibonacci numbers mod $10^9 + 7$, we can find the solution in $O(N)$ time. Remember to subtract 1 from the final product because the problem asks for non-empty subsets.

Time Complexity: $O(N)$

Problem: Gabriel Wu
Editorial: Gabriel Wu
Solutions: C++, Java, Python

§7 Hen Hackers

Since we are given the information that all characters that appear in the string only appear once, we may simply query each of the 62 possible characters one at a time to determine the characters the password consists of.

All we need now is the order of these characters within the password. Though this may seem strange at first, we can simply apply a sort operation on the list of known characters within the string. This requires a custom comparator: a function that determines whether one character should go in front of or behind another. In this case, the comparator would be a function that sends queries of two characters to the grader. Depending on the output from the grader, the comparator returns the respective value for the sorting algorithm. For example, if you want to know whether 'a' comes before or after 'b' in the password, you simply query 'ab'. After all the characters are sorted, we query the correctly ordered string of characters and end the program.

Additionally, it is important to remember that the password size could be 1 or 2 characters long, so any time we are sorting with the comparator and receive a response of 'C' instead of 'Y' or 'N', we must immediately end the program.

Query Complexity: $O(N \log N)$, where N is the password length.

Problem: Gabriel Wu
Editorial: Ayush Varshney
Solutions: C++, Java, Python

§8 Platypus Puddles

Let the water level be the amount of water on a cell, plus the height of the ground in the cell. Let the *weight* of a contiguous path of cells in the habitat (connected by shared edges, not shared corners) be the maximum ground height of any cell on the path. The key idea here is the following: if there is a contiguous path of cells with weight h going from cell c to the edge of the habitat, then the water level at cell c can be no more than h. If this condition is violated, it means there is a way for water from cell c to flow out of the habitat, which is impossible after an equilibrium is reached. Now, let H_c be the minimum value of h for all paths connecting cell c to the edge. The water level at cell c must be exactly H_c (so if H_c is equal to the height of the ground at c, then c has no water).

Now, let H_c represent the minimum weight of any path from the edge of the habitat to cell c. It turns out that H_c is exactly the water level at cell c because it is guaranteed that enough rain fell to fully saturate all puddles. Our task is now to find H_c for each cell c. In other words, we want to find the lowest weight path from the edge of the habitat to each cell. Think: what algorithm could help us with that? Since the our definition of *weight* is non-decreasing (meaning that a path can never lose weight by adding more nodes), we simply apply Dijkstra's algorithm. Think of the exterior of the habitat as a starting node that is connected to all cells on the border of the habitat. Then, use a priority queue and successively visit cells with the next lowest H_c values. Once you have all H_c values, add them up and subtract the sum of the ground heights.

Time complexity: $O(N^2 \log N)$

Note: We couldn't get our Python solution to run in much less than 10 seconds, which is why we recommended using a different language.

Problem: Gabriel Wu
Editorial: Gabriel Wu
Solutions: C++, Java, Python

§9 Playlist Shuffle

First note that at any location, there is an optimal move that you should take regardless of any of your previous moves. Now we characterize these types of optimal moves. Note that there is a radius R such that if you are at most R songs away from B, you should just press *next* or *previous* until you arrive at B. Otherwise, you hit *randomize*. Our goal is to compute this R in sublinear time. Let $f(R)$ be the expected amount of time you would take with to get to B with a radius of R. Note that if R is very large, then $f(R)$ is also large. This makes sense since you're not using the random function smartly. Note that if R is very small, then $f(R)$ is also large, since you're using the random function way too often, and thus the expected value goes up. Thus, if we graph $f(R)$ on a plane, it looks like a "U". The formal proof of this is left as an exercise to the reader. But now we can ternary search the optimal value of R.

It suffices to be able to compute $f(R)$ for a given R. For all locations L such that L is within R of B ($|L-B| \leq R$), we have that the expected amount of additional time to get to B if you're at L is simply $|L-B| \cdot X$; you just go straight to B. For all other L, you click *randomize* immediately. Thus for all other L, we have that the expected amount of additional time to get to B is a constant, which we will call E. Let the sum of $|L-B| \cdot X$ for all L satisfying $|L-B| \leq R$ be S, and the number of such L be M. Then we have that $E = \frac{N-M}{N} \cdot E + \frac{M}{N} \cdot S + Y$. This is because you gain an additional Y seconds, and then you do casework onto which type of song you randomize onto. Thus, you can solve for E. It's thus easy to find the average value of all the expected values for each location L from $1, \ldots, N$, which is simply $f(R)$. This can be done in $O(1)$ time.

Time Complexity: $O(\log N)$ due to the ternary search.[1]

Problem: Timothy Qian
Editorial: Timothy Qian
Solutions: C++, Java, Python

[1]It is probably possible to solve this in $O(1)$, but it's just more math. And this is a programming competition!

§10 Penguin Mayhem

Since we can treat penguins as passing straight through each other, we can just compute the number of collisions between all pairs of penguins at time less than or equal to T. We want to compute that pretty quickly, since we need to do this N^2 times.

Consider two penguins whose coordinates are determined by $(x_1 + p_1t, y_1 + q_1t), (x_2 + p_2t, y_2 + q_2t)$ at time t. We determine when their x-coordinates are the same first; their y-coordinates can be done analogously. This is precisely when we have $x_1 + p_1t = x_2 + p_2t + kW$, where k is an integer. The extra kW accounts for the wrap around effect. Thus we have that $t = \frac{x_2-x_2}{p_1-p_2} + \frac{W}{p_1-p_2} \cdot k$ for some integer k. The case of $p_1 = p_2$ is addressed by noticing this just means they have the same x-speed; thus they either always have the same x-coordinate or never do. We can easily take care of this case separately, so assume $p_1 \neq p_2$. Note that we can simplify this to $t = a + bk$ for some integer k and rational numbers a, b. And similarly, when their y-coordinates are the same, we must have $t = c + dl$ for an integer l and rational numbers c, d. (We similarly assume here $q_1 \neq q_2$, as this can be taken care of separately). Thus we have $a + bk = c + dl$, or $bk - dl = (c - a)$. Taking advantage of the fact that a, b, c, d are rational numbers, we clear denominators to make this equation of the from $gk + hl = i$, for integers g, h, i. Note we can assume $\gcd(g, h, i) = 1$ by dividing through by the gcd. Taking $\pmod g$, we get $l = i/h \pmod g$, and taking $\pmod h$, we get $k = i/g \pmod h$, where division is modular inverse (g, h could share factors making division impossible, but if that happens, then i must share this factor with g, h for there to be a solution, which contradicts us saying the three are have a gcd of 1). This is solvable via the Chinese Remainder Theorem. We get an equivalence of the form $k \equiv u \pmod v$ for some integers u, v.

And thus we can characterize all t where the penguins intersect. It's easy to compute the number of integers k such that $k \equiv u \pmod v$ and $0 \leq a + bk \leq T$ in $O(1)$. We leave that as an exercise for the reader. So we can count everything by doing some math. The complexity comes from clearing denominators, doing the Chinese Remainder Theorem, and maybe applying LCM/GCD (if you're using fractions to store the rational numbers). This comes out to $O(\log(\max(W, H)))$. One must be careful when doing all these operations because handling these numbers wrong will overflow even long long sometimes.

Time Complexity: $O(N^2 \log(\max(W, H)))$.

Problem: Timothy Qian, Ayush Varshney
Editorial: Timothy Qian
Solutions: C++, Java, Python

§11 Sharing Seals

Think of the array being represented as a matrix. We will consider the case of $N = 4$ as an example. The inital array would be represented by

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix}$$

If we multiply this out, we just get the right matrix. To do one change for $K = 1$, we left multiply by the matrix

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

And if we do this change twice, we would simply multiply by the square of this matrix, which is

$$\begin{bmatrix} 2 & 1 & 0 & 1 \\ 1 & 2 & 1 & 0 \\ 0 & 1 & 2 & 1 \\ 1 & 0 & 1 & 2 \end{bmatrix}$$

Each the jth weight from the top of the ith column is like how much each A_j contributes to A_i. At this point, you could try to do matrix exponentiation with matrix multiplication at get an $O(N^3 \log T + N^2 Q)$ solution. We made the time limit one second to try to discourage these solutions, but it is possible that some of these got through.

However, note that this columns are just shifts of each other. This should be intuitive, as our update is circularly symmetric. Thus, we can precompute the 1st column of this matrix for an update for $K = 1$ for any T by just applying the transition over and over to an array with only one 1 and the rest 0's. This would take $O(NT)$ times, because it takes $O(N)$ time to transition for each of the T arrays we want to keep track of.

But note that there are only Q of these arrays that we care about, because there are only at most Q different update numbers. So we can store this in $O(NQ)$ space. Thus, for all updates with $K = 1$, we can just compute this matrix, and update the values by multiplying the matrix with the current array. This would take $O(N^2)$ time for each query, for an overall $O(N^2 Q + NT)$ solution if all $K = 1$.

For $K \neq 1$, we can reduce this down to a case where $K = 1$. Note that If we take the elements $A_i, A_{i-K}, A_{i-2K}, \ldots$, where the indices are $\pmod N$, we form cycles. we can treat this cycle as equivalently the $K = 1$ case. However, the length of these cycles might not be N, they will be d, where $d|N$. One approach is to just compute the tables for all divisors of N, which would take $O(N^2 Q + N^{\frac{4}{3}} T)$ time and $O(N^{\frac{4}{3}} T)$ space (the extra $N^{\frac{1}{3}}$ comes from the number of divisors of N). We tried (and maybe unsuccessfully) to make these solutions exceed the time limit.

The other solution is to notice that if $d|N$, we can simply take our precomputed weights for N, and collapse it to an array for d by adding it over. For if our precompute weights for $N = 4$ where $[2, 1, 0, 1]$, to get the weights for $N = 2$, we can split the arrays into $[2, 1], [0, 1]$, and add them to each other to get $[2, 2]$. I'll leave if to the reader to figure out why that works. Thus, we actually don't need any extra computation. The solution will still take $O(N^2Q + NT)$ time. This runs very quickly in around 200 ms for all test cases.

A fun fact is that the weights for $K = 1$ are simply the rows of Pascal's triangle. You could also try roots of unity filter once you know that, but that seems really messy.

Time Complexities: $O(N^2Q + NT), O(N^2Q + N^{\frac{4}{3}}T), O(N^2Q + N^3 \log T)$

Problem: Timothy Qian
Editorial: Timothy Qian
Solutions: C++, Java, Python

§12 Zookeepers' Gathering

Let's make two observations.

1. The timeline can be represented as an array of $N + 1$ integers. Adding an event is akin to adding 1 to all elements in array positions L to R, and removing an event is subtracting 1 from L to R. The longest contiguous block of available time is just the longest subarray of 0s.

2. The peculiar stipulation of specifying four different people, each with non-overlapping events, means the array values can only ever range from 0 to 4 inclusive.

Adding and subtracting from a subarray can easily be done with a segment tree, but how do we find the longest subarray of 0s? Instead of thinking of an update as adding or subtracting, let's think of it as shifting. If we maintain the longest subarray of 0s, 1s, 2s, 3s, and 4s for a given interval, then adding 1 to the entire interval means the new answer for longest subarray of 4s is now the old answer for the longest subarray of 3s, the new answer for longest subarray of 3s is now the old answer for the longest subarray of 2s, and so on. Subtracting is the same idea, except it's a downward shift.

The only question left is how to compute a non-leaf node in the segment tree based on its children. In addition to the longest subarrays for each of the five values, we'll also maintain the longest contiguous prefix and suffix for each of the five values as well as the size of the interval in each of the nodes. Say we are computing node C based on its children A and B.

For i = 0, 1, 2, 3, 4:

$$C.prefix[i] = \begin{cases} A.prefix[i] & A.prefix[i] < A.size \\ A.prefix[i] + B.prefix[i] & A.prefix[i] = A.size \end{cases}$$

$$C.suffix[i] = \begin{cases} B.suffix[i] & B.suffix[i] < B.size \\ B.suffix[i] + A.suffix[i] & B.suffix[i] = B.size \end{cases}$$

$$C.answer[i] = max(A.answer[i], B.answer[i], A.suffix[i] + B.prefix[i])$$

So the complete solution is to use a segment tree with lazy propagation that maintains an array of size 5 of prefix, suffix, and answer at each of the nodes. Merge nodes using the merge function described above.

Time Complexity: $O(Q \log N)$

Bonus: Can you solve the problem under the same bounds of Q and N if there was no limit on the number of people? [2]

Problem: Maxwell Zhang
Editorial: Maxwell Zhang
Solutions: C++, Java, Python

[2]Hint: square root. We didn't put this on the contest because it would be hard to distinguish between this solution and just an $O(N^2)$ bash.