mBIT Varsity Solutions

November 2019

These are the solutions to the rookie 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 Baking Pan

Let's find the answer for a single circle. For any set of points, the bounding rectangle of minimal area has its top at the highest y-coordinate of the points, its bottom at the lowest y-coordinate, its left at the lowest x-coordinate, and its right at the highest x-coordinate.

This means that for any circle, the bounding x-coordinates are $c_x - r$ and $c_x + r$, and the bounding y-coordinates are $c_y - r$ and $c_y + r$, where (c_x, c_y) is its center and r is its radius. We can therefore only focus on those points for each circle and ignore everything else. Our bounding rectangle is determined by the minimum/maximum x and y coordinates of the points for all of the circles, and the area can be easily calculated.

Each circle contributes 4 ($O(1)$) points of interest, so our total complexity is $O(N)$.

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

§2 Frosting Patterns

What would the string look like if it was periodic (repeating) with period K? The characters $1 \dots K$ would be equal to $K+1 \dots 2K$, which would be equal to $2K+1 \dots 3K$, and so on. This equality must hold for every block of K characters that starts at the end of the last block, up until the last block in the string. If N is not divisible by K, then only the first characters of the last block up to N must be equal to the first characters of every other block, otherwise the whole block must be equal to every other block. If we can find any K that satisfies this, then we can construct our answer. Checking $\lceil \frac{N}{K} \rceil$ blocks of size K will take up to N operations, and we have up to $N/2$ (because the string must repeat at least twice) values of K to check. This process of checking is $O(N^2)$.

Once we find a K, we can construct a string of length $N+M$ with that period and print the substring $N+1 \dots N+M$. Just keep appending the period string until the answer string's length is greater than or equal to $N+M$. Our final complexity is dominated by finding the length of the period, so it is $O(N^2)$ in total.

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

§3 Mountain Mileage

We can just check each house and find the total gas used to reach each house. This can be done in just $O(N)$ after sorting the array of houses in $O(N \log N)$. We loop through the array and maintain the left and right sums. The left sum begins at zero, and we add each house's height to the left sum after processing it. Similarly, the right sum begins at the total sum, and we subtract each house's height from the right sum after processing it. When processing any given house H_i, the total gas used is

$$\begin{aligned} & A \cdot ((H_i - H_1) + (H_i - H_2) + \cdots + (H_i - H_{i-1})) \\ & +B \cdot ((H_{i+1} - H_i) + (H_{i+2} - H_i) + \cdots + (H_n - H_i)) \\ & = A \cdot ((i-1)H_i - (H_1 + H_2 + \cdots + H_{i-1})) \\ & +B \cdot ((H_{i+1} + H_{i+2} + \cdots + H_n) - (n-i)H_i) \\ & = A \cdot ((i-1)H_i + leftSum_i) + B \cdot (rightSum_i - (n-i)H_i) \end{aligned}$$

The answer is just the minimum of this value for all H_i.

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

Note: It turns out the optimal house is always located at a weighted median (assuming 0-based indexing, the optimal index of the sorted array is $\lfloor n \cdot b/(a+b) \rfloor$). See if you can prove this fact!

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

§4 Milky Way

Run a BFS, keeping track of the planet we're at. We also need to keep track of how much fuel we have, so our state will not just be the planet we're at, but rather (current planet, current fuel). Because of the nature of BFS, whenever we're at a state we've already visited, we'll have taken more steps to get here than the first time we visited this state, so we shouldn't continue on this path. This means we only actually visit each state 1 time.

For each state, we can use the edges adjacent to the planet we're at to transition to new states. To consider refueling at a planet, we can add another edge from each planet to itself with negative weight (so traversing it gives us fuel). If refueling gives us more than C fuel, we just set our fuel to C. If we have less than 0 fuel after taking an edge, we cannot continue on that path. Start at the state (planet $= A$, fuel $= 0$). The first time we reach B will be the minimum answer (also because of the nature of BFS).

For each fuel level c ($0 \leq c \leq C$), there will be N states (one for each vertex), and $N + M$ transitions to other states. Thus, we perform $O(C(N + M))$ operations in total.

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

§5 Coffee Swapping

First, let's characterize what a maximal beauty configuration is, and then we'll figure out the minimum number of swaps. Determining the maximal beauty configuration is equivalent to finding the maximum possible number by pairing up coffees, summing the temperatures in each pair, and third taking the product of these pair values. Say we have four coffees with temperatures $a < b < c < d$. The possible pairings we can have yield beauty products of $(a+b)(c+d), (a+c)(b+d), (a+d)(b+c)$. Subtracting the third from the second and factoring, we get $(a-b)(d-c)$. Note this is 0 if and only if $a = b$ and $c = d$, which is impossible for distinct temperatures. This is always nonpositive, therefore the third is larger than the second. Subtracting the third from the first, we get $(a-c)(b-d)$, which is always nonnegative. Note that this is equal to 0 if and only if we have $a = c$ or $b = d$, meaning $a = b = c$ or $b = c = d$, again impossible for distinct temperatures. Thus the third configuration is the largest.

So say that we have coffees a, b, c, d such that currently two are paired and the other two are paired, we want to pair the largest with the smallest. So now we will prove the largest and the smallest temperature coffees must be paired together. Say for contradiction they are not. Let the smallest have temperature a and largest have temperature d. Say a is paired with b and d is paired with c. We have that $a \leq b, c \leq d$. But then we can make the temperature higher by pairing a with d by what we proved before. Thus we proved we have to pair the smallest and largest coffees. Similarly, the next smallest and next largest temperature coffees must be paired, and so on.

Now we must figure out the minimum number of swaps to achieve this maximal beauty. We already know which coffees have to be paired. Thus we can label the coffees that have to be paired together with the same numbers. Thus we can view this as a permutation. Now we can view this permutation in terms of cycles. To resolve each of the cycles, it requires the length of the cycle minus one swaps. Thus we can do cycle decomposition in $O(N)$, and thus the overall time complexity to solve this is $O(N)$.

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

§6 Hot Cake

First consider the directed graph formed by drawing arrows from a person to the person he/she will pass to. We can decompose this into cycles. This will take $O(N)$ time. If we compute the length of the cycle, we can easily figure out where the hot cake will be in $O(1)$ time, as the location will just follow the same loop of people over and over. Thus the overall time complexity is $O(N)$ from the cycle decomposition.

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

§7 Cupcake Distribution

Say that a chef wants to serve people with sweetness preferences of $S_l, \ldots, S_r$. We will find the optimal value that minimizes their unhappiness. Thus we want to find the x that minimizes $(S_l - x)^2 + \cdots + (S_r - x)^2 = S_l^2 + \cdots + S_r^2 + (r - l + 1)x^2 + 2x(S_l + \cdots + S_r)$. But this is a quadratic function that can be minimized by completing the square. Thus we find the optimal value is to choose $x = \frac{S_l + \cdots + S_r}{r - l + 1}$, which intuitively makes sense because this is the average of their preferences. Note that we can compute the lowest unhappiness if a chef were to serve the people with sweetness preferences of $S_l, \ldots, S_r$ in $O(1)$ with $O(N)$ preprocessing. We do this by maintaining prefix sums of the sum of the squares of the sweetness preferences and a prefix sums of just the sweetness preferences.

We will now solve this problem using DP. Let the DP state $dp[i]$ be the lowest total unhappiness of serving the first i people. We have that $dp[i] = \min_{1 \le a \le i-k}(dp[a] + cost(a + 1, i))$, where this comes from simply iterating the interval the ith chef will serve. This can be computed in $O(N^2)$ as we have shown the cost can be computed in $O(1)$.

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

§8 Secret Base

Consider a vertex that has at least K outgoing tunnels. If it has less than K, it cannot possibly get us an answer. Otherwise, we need to find K that are in the same component. How do we usually check for components? We can run a BFS from each possible source and mark components, making sure to not go through our chosen vertex. Then, for each component, we can take the K smallest tunnels from each one (provided there are at least K) and update our answer. This is $O(\frac{NM+M\log M}{K})$, which is too slow for small K.

Because this graph is a tree, we can speed up the component-finding process. Let's set up binary lifting for our tree (where we store the 2^j-th parent of each vertex for j in $[0, \lfloor \log_2 N \rfloor]$), because this allows us to find the least common ancestor (LCA) of any two vertices. For any tunnel that goes from our chosen vertex i to another vertex j, if the LCA of i and j is i, then j is a child of i. The component that j is in can be uniquely identified by the direct child of i in the subtree of j, which we can use the binary lifting to find in $O(\log N)$. If the LCA of i and j is anything else, then j is not a child of i, and is in a component with every other node that isn't a child of i.

We can use any map or list to store the costs of the tunnels in each component efficiently. For each component, if it has at least K tunnels that lead to it, then we will take the sum of the K tunnels of least cost, and update our minimum answer. If we run this for each vertex with at least K outgoing tunnels, we will have our answer. Finding the LCA and initializing/using binary lifting is $O(\log N)$ for each edge, and taking the K tunnels of least cost for components must be $O(M \log M)$, so our final complexity is $O(N + M(\log N + \log M))$

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

§9 Contradictory Canelé

First, we show which A, B for a given N, M yield no possible configurations of canelé. If $A|N, B|M$, note that we can divide the $N \times M$ grid into $A \times B$ rectangles, and since the sum within each of these rectangles is nonnegative, the overall sum is nonnegative. Thus in this case, there are no solutions.

Now we will construct solutions if $A \nmid N$ or $B \nmid M$. WLOG assume that $A \nmid N$. Let $N = pA + q$, $M = rB + s$, where p, q, r, s are positive integers and $q < A$ and $s < B$. We will try to now "almost" tile the $N \times M$ grid with the same $A \times B$ rectangle. Consider an $A \times B$ rectangle with every number negative except the bottom right corner, which is chosen such that the sum of the numbers in the $A \times B$ rectangle is 0. Clearly we can choose the positive numbers to be all different, thus every number in this $A \times B$ rectangle is different. Let this set of numbers be S. Now we consider a $(p+1)A \times (r+1)B$ rectangle and tile with $(p+1)(q+1)$ $A \times B$ rectangles of the format. Now we remove $p - q$ rows from the bottom and $r - s$ rows from the left. We are left with a $N \times M$ grid where every $A \times B$ rectangle consists exactly of the numbers in S, and the sum is negative because we cropped out part of the $A \times B$ rectangles tiling the bottom left of the original $(p+1)A \times (q+1)B$ rectangles such that we kept negative numbers, making the overall sum negative. A case of $A = 2, B = 3, N = 5, M = 7$ is shown below.

$$\begin{array}{ccccccc} -1 & -2 & -3 & -1 & -2 & -3 & -1 \\ -4 & -5 & 15 & -4 & -5 & 15 & -4 \\ -1 & -2 & -3 & -1 & -2 & -3 & -1 \\ -4 & -5 & 15 & -4 & -5 & 15 & -4 \\ -1 & -2 & -3 & -1 & -2 & -3 & -1 \end{array}$$

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

§10 Cake Cutting

We first introduce some basic terminology to help with the understanding of the solution. In this solution, angles are taken counterclockwise. Say we want to find the number of polygons that $P = (x, y)$ is contained in if we have a polygon with vertices $(x_1, y_1), \ldots, (x_N, y_N)$ in counter clockwise order, labeled $V_1, \ldots, V_N$ respectively. Say we choose vertices. Now say that we want to count the k sided polygons that contain this point. Let the degree angle $d_i = \angle V_i P V_{i+1}$ where $V_{N+1} = v_1$.

We will use complementary counting to count the number of k gons. There are $\binom{N}{k}$ polygons as a preliminary count. Now we subtract the number of polygons that don't contain P. Say that that we choose vertices $U_1, \ldots, U_k$. Then the only way this polygon doesn't contain P is if there is an i such that $\angle U_i P U_{i+1} \geq 180^\circ$ where $U_{k+1} = U_1$ Call such polygons *bad*. We count these such polygons by doing casework. Say $U_1 = V_i$ is in a polygon that is bad such that $U_1 P U_2 \geq 180^\circ$. Let's say V_j is the first counterclockwise vertex from V_i such that $V_i P V_j \geq 180^\circ$. Then any set of $k-1$ vertices we choose from $V_j, V_{j+1}, \ldots, V_{i-1}$ where these are going counterclockwise. Say that in this set there are M vertices. Then the number of polygons is $\binom{M}{k-1}$ we must sum this over all k that are greater than or equal to 3. By the binomial theorem this is $2^M - 1 - M$. We can compute the V_j for each V_i using line sweep. Thus for each point P, we can compute the number of polygons containing it in $O(N)$. Thus the overall complexity is $O(NQ)$.

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

§11 Lost Child

First of all, we solve a simpler version of the problem. Let's say that Gabe starts at a location which we will represent with $\vec{a}$ and ends at a location represented by $\vec{b}$, traveling in a straight segment, and Anna starts at a location which we will represent with $\vec{c}$ and ends at $\vec{d}$, also traveling in a straight segment. Furthermore, say they start at $\vec{a}$ and $\vec{c}$ at the same time and reach $\vec{b}$ and $\vec{d}$ at the same time. Thus at any point in time we can represent their locations with $t\vec{a} + (1-t)\vec{b}$ and $t\vec{c} + (1-t)\vec{d}$ where $t \in [0, 1]$. If we take the difference, we want to minimize the magnitude of the vector $t(\vec{a} - \vec{c}) + (1-t)(\vec{b} - \vec{d})$ for $t \in [0, 1]$. Letting $\vec{e} = \vec{a} - \vec{c}$ and $\vec{f} = \vec{b} - \vec{d}$. we want to minimize $t\vec{e} + (1-t)\vec{f}$. This is minimizing the distance from a point on the line segment from $\vec{e}$ to $\vec{f}$ to the origin, which we can do in $O(1)$ time. An alternative way without having to resort to vectors is noticing that the distance is a unimodal (monotonic) function, and thus we can apply ternary search.

Now we want to show that we can reduce the problem to this simpler version. We can calculate the time that Gabe and Anna will reach the endpoint of each segment, and compute where the other is at that time. Thus we can create add these points to the points that Gabe and Anna have to pass through, reducing the problem to the previous problem. This can be done in $O(N + M)$. Thus the overall time complexity is now $O(N + M)$, or $O(N \log N + M \log M)$ if ternary search is applied.

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

§12 Outbreak

Let's first solve the problem assuming all queries are on the whole array. Using exponent rules, we have that the number that we multiply by for each query, $\left(\prod_{i=0}^{N-1}(B_i)^{X(i+1)}\right)$, is equal to $\left(\prod_{i=0}^{N-1}(B_i)^{(i+1)}\right)^X$. Since all B_i are constant throughout, we can simply precompute $\prod_{i=0}^{N-1}(B_i)^{(i+1)}$ and raise it to the power of X for each update with modular exponentiation. With this strategy in mind, we can now apply it to a segment tree.

First, how do we apply an update to a segment, since the update doesn't necessarily start where the segment starts? If the update is on $[L_Q, R_Q]$ and the segment is on $[L_S, R_S]$, then the segment is being multiplied by $\left(\prod_{i=L_S}^{R_S}(B_i)^{(i-L_Q+1)}\right)^X$, which is equal to $\left(\prod_{i=L_S}^{R_S}(B_i)^{i-L_S}\cdot\prod_{i=L_S}^{R_S}(B_i)^{(L_S-L_Q+1)}\right)^X$.

The second part, $\prod_{i=L_S}^{R_S}(B_i)^{(L_S-L_Q+1)}$, can be further reduced to $\left(\prod_{i=L_S}^{R_S}(B_i)\right)^{(L_S-L_Q+1)}$. We can compute $\prod_{i=L_S}^{R_S}(B_i)$ in constant time for any segment by precomputing prefix products and the modular multiplicative inverses for each of those products, then raise it to any power when necessary.

The first part, $\prod_{i=L_S}^{R_S}(B_i)^{i-L_S}$, which is equal to $\frac{\prod_{i=L_S}^{R_S}(B_i)^i}{(\prod_{i=L_S}^{R_S}(B_i))^{L_S}}$, can be found for any segment with a similar strategy. If we compute the prefix products (and their inverses) of $(B_i)^i$, then we can obtain $\prod_{i=L_S}^{R_S}(B_i)^i$ in constant time. We can then divide (multiply by the multiplicative inverse) by $\left(\prod_{i=L_S}^{R_S}(B_i)\right)^{L_S}$, which we can find in logarithmic time using our precomputation of the second part and modular exponentiation.

Before we go further, let's use some shorthands so this editorial doesn't have five more pages of pi notation. For a segment which spans some range $[L, R]$, let its *raised product* be the product obtained in the paragraph above: $\prod_{i=L_S}^{R_S}(B_i)^{i-L_S}$ and let its *range product* be simply the product of each B_i in the range: $\prod_{i=L_S}^{R_S}(B_i)$. Then, we can represent a segment as the exponents on these two values. Initially for each segment, the coefficient for the raised product is 0, and the coefficient for the range product is 1.

Each node in the segment tree will store the current product of the segment. We have range updates, so we need to apply lazy propagation. We'll represent each segment in the lazy array with two values: its coefficients on its raised product and range product. Let the coefficient on the raised product be E (for exponent) and the coefficient on the range product be R. For a segment $[L, R]$ with children $[L, M]$ and $[M+1, R]$ (where $M = \lfloor\frac{L+R}{2}\rfloor$), its coefficients will have different effects on its left and right children.

When we evaluate the lazy values of some parent segments, we must update the lazy values of its children (unless it's a leaf). For the left child: $E_{left} = E_{left} + E_{parent}$ and $R_{left} = R_{left} + R_{parent}$. For the right child, we still have that $E_{right} = E_{right} + E_{parent}$, but the update for R_{right} is different. Because the right child's start is $(M+1) - L$ positions to the right, we have to add $E_{parent} \cdot ((M+1) - L)$ to each of the coefficients, which is the same as adding that to R_{right}. Therefore, the change in R_{right} is $E_{parent} \cdot ((M+1) - L)$.

Now that we have a way of updating children, updating the actual segments based on the coefficients is simple. We just raise the range product to R_i and the raised product to E_i, then multiply them both with the current value at the segment.

This approach will have the standard $O(Q \log N)$ from the segment tree, plus an extra log factor from modular exponentiation. For initializing the segment tree, we visit $O(N)$ segments and perform modular exponentiation on each of them at least once, so it takes $O(N \log 10^{18})$ overall. Combining these, our total complexity is $O(N \log 10^{18} + Q \log N \log 10^{18})$.

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