mBIT Rookie 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 Rating System

If Chloe wins, she gains 10% of her opponent's rating, so the answer is c + o / 10. If Chloe loses, she loses 10% of her rating, which leaves her with 90% of her rating remaining, so the answer is c * 9 / 10.

Time Complexity: $O(1)$

Note: Since the input guarantees all ratings to be divisible by 10, you can just use the integer data type without worrying about decimals.

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

§2 Candy Bowl

Loop through all the integers and for each one, print YES if the number is divisible by 2 and 3 but not both and NO otherwise. You can check if a number a is divisible by b by checking if $a \ \% \ b == 0$, where % is the modulo operator.

Time Complexity: $O(N)$

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

§3 Patisserie

Loop through each character of the input string, and add one to the answer each time the character is a vowel. If the character is a y, only add one to the answer if the following character is a space or it is the last character in the input.

Time Complexity: $O(N)$ where N is the length of the input string.

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

§4 Sugar Cubes

Loop through each possible i from 1 to N, then each j from i to N, then each k from j to N. Each time $i \cdot j \cdot k = N$, increment a counter, and print this counter at the end.

This solution is $O(N^3)$. It may also possible to solve this more efficiently using a combinatorics trick called "stars and bars" and the principle of inclusion and exclusion. However, for the given bounds $O(N^3)$ is sufficient.

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

§5 Genotypes

Consider each trait individually. If the two parent plants have the same type (both capital or both lowercase), then the offspring's allele for that trait must be the same as the parents. Otherwise, the parents have different alleles and the offspring's allele can be either one, meaning that there are two possibilities for that trait. For each trait where the offspring can have either of the alleles, there are 2 times as many possible genotypes. Thus, we multiply by 2 each time and obtain an answer of $2^{(number\,of\,unequal\,traits)}$.

We look at each character in the strings once, so the solution is $O(N)$.

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

§6 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

§7 Number Cookies

Since N is extraordinarily small, you can check all $N!$ possible permutations of the expression. For each one, scan the expression from left to right and double check that the expression is valid (no leading zeroes, all operators are in valid locations). Leading zeroes can be checked by ensuring there exists a character before the zero that is a digit from 1-9. Operators can be checked by ensuring the preceding and subsequent characters are digits.

Once checked, an expression can be evaluated in $O(N)$ by either using a stack to deal with the order of operations or by first looping through to take care of all multiplication operators, then looping through again to evaluate the addition and subtraction operators. The answer will be the maximum of all $N!$ expressions.

Time Complexity: $O(N * N!)$

Note: Certain languages have certain functions to ease the implementation of this problem (e.g. Python and C++ have `itertools.permutations()` and `std::next_permutation` for iterating through all permutations of a sequence, and Python has `eval()` to evaluate arithmetic expressions).

Also, there exists a $O(N)$ solution that involves optimizing the placement of values in the expression to achieve a maximum value which is shown in the Java solution.

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

§8 Cinnamon Spiral

The bounds on X and Y are small enough to directly enumerate the road. Notice that there is a pattern on how much we move for each change in direction (this observation is not necessary, but it makes implementation easier). We first move 1 unit, then 2 units, then 2 units again, then 3 units, then 3 units again, and so on. At each point, we can just check if the current segment we're moving in contains our endpoint, and stop if so. This solution terminates in $O(\max(X, Y))$ time.

We can also solve this in $O(1)$ (say, in the harder problem of $X, Y \leq 10^9$). Observe that any coordinate (x, y) where $\max(x, y) < \max(X, Y)$ will always be filled in by the road. The length of the road is given by the number of visited coordinates minus one, so we can immediately add $(2\max(X, Y) - 1)^2$ to our answer to get to the point $(\max(X, Y), -\max(X, Y) + 1)$. It remains to enumerate four moves, which can be done with either casework or the strategy as above.

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

§9 Ice Cream

We are told the input looks like an ice cream. This means there is a general area with a lot of 'p' pixels and a general area with a lot of 'b' pixels. Even with some randomness thrown in, we can assume that the average location of the 'p' pixels is the general placement of the cherry ice cream, and the general location of the 'b' pixels is the general placement of the cone. We can compute these averages in $O(N^2)$ and then connect them to form a line segment, we can determine the orientation of the ice cream. Note that when we refer to the average of the pixels, we are really referring to the "center of mass" or the "centroid" of the points.

A classic way to determine the angle of a line segment to the horizontal is by computing the arctangent of the absolute difference in y values divided by the absolute difference in x values of the two endpoints of the segment. We can employ this technique to determine whether the segment is more horizontal (90° or 270°) or more vertical (0° or 180°). To determine which of the final two options is the correct orientations, simply compare the x values of the two points if it is more horizontal or the y values of the two points if it is more vertical.

However, there exists a simpler way than using the arctangent function because the problem asks for the angle rounded to the nearest 90 degrees. It follows that if the absolute difference in the x coordinates is greater than the absolute difference in the y coordinates, the ice cream must be oriented horizontally. The opposite also holds true. Then the final orientation can be determined by checking whether the 'p' pixels are above/below/left/right of the 'b' pixels.

Time Complexity: $O(N^2)$

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

§10 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

§11 Dessert Islands

This is a standard example of a flood fill problem (if you're not familiar with flood fill, you can read about it here). Basically, we loop through the graph from top to bottom and from left to right, and for each unvisited cell, we run a BFS or DFS from that cell to every adjacent cell of the same type of material and mark the cell as visited. By doing this, we can count the number of connected chunks of liquid and solid.

Finally, when printing the number of chunks, remember to subtract one from the number of solid chunks as the border does not count as an island.

Time Complexity: $O(NM)$

Note: If you use DFS in Python, you might hit the max recursion depth, so be careful.

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

§12 Sugar Sampling

Let's keep track of two different types of values at each cell of the grid: the number of paths to that cell while visiting a certain type of candy in the path, and the total number of paths to that cell. More specifically, we denote $dp(i,j,k)$ as the number of paths from the top left corner to the cell at row i and column j while visiting a candy of type k at least once and we denote $numPaths(i,j)$ as the total number of paths from the top left corner to the cell at row i and column j.

From this, we can develop a recursive relationship for these values. First, we initiate $numPaths(1,1) = 1$, since there is only one way to arrive at the top left corner: simply starting there. Similarly, $numPaths(i,1) = numPaths(1,j) = 1$ for all $1 \leq i,j \leq N$, since there is only one way to reach those cells by moving either strictly horizontally or vertically from the top left corner. Finally, for every other cell, you can arrive at that cell from either the cell above or to the left of the current cell, so $numPaths(i,j) = numPaths(i-1,j) + numPaths(i,j-1)$.

We compute the number of paths with specific candy types in the same manner. First, we initiate $dp(1,1,k) = 0$ for all $1 \leq k \leq N$, since the top left corner has no candy on it and thus no valid paths for any candy type. Next, if the cell at (i,j) contains a candy of type k, then all paths to that cell will be considered valid paths that contain at least one candy of type k, so $dp(i,j,k) = numPaths(i,j)$. Otherwise, the number of valid paths for a cell is equivalent to the number of valid paths to the cell above and to the left, since those are the two possible locations we could arrive from in order to land on the current cell. If we are on the leftmost column, then there is no cell to the left and we can only arrive from above, so $dp(i,1,k) = dp(i-1,1,k)$. Similarly, if we are on the topmost row, then there is no cell above and we can only arrive from the left, so $dp(1,j,k) = dp(1,j-1,k)$. Finally, for every other cell, $dp(i,j,k) = dp(i-1,j,k) + dp(i,j-1,k)$.

In order to implement this solution, we could use a bottom-up approach where we store $numPaths$ and dp as 2D and 3D arrays of size $N \times N$ and $N \times N \times N$. We loop through all i from 1 to N, then for each i we loop through all j from 1 to N, and finally for each j we loop through all k from 1 to N, so that we can compute our array values for each combination of i,j,k. This way, each $numPaths(i,j)$ and $dp(i,j,k)$ can be computed based on previous array values that were already computed in a previous iteration of the loop.

Time Complexity: $O(N^3)$

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

§13 Magical Calendar

This problem tests your understanding of the map data structure (`std::map` or `std::unordered_map` in C++, `java.util.HashMap` or `java.util.TreeMap` in Java, dictionaries in Python). There are multiple ways you could approach this problem, but one simple approach is to maintain a map with the event names as the keys and the dates of the events as the values. In order to process the dates, create a conversion map that converts the month names to numbers so they can be sorted (i.e. January = 1, February = 2, etc.). Then, each date can be stored using a pair container or custom class. Afterwards, we can deal with each of the four commands as follows:

ADD - Check if the calendar already contains the event name as a key, and if it doesn't, add a new entry to the map.

DELETE - Check if the calendar already contains the event name as a key, and if it does, remove that entry.

RESCHEDULE - Check if the calendar already contains the event name as a key, and if it does, change the value of that key to the new date.

PRINT - Check if the calendar already contains the event name as a key. If it does, print the date stored as the value of that key. If not, print "NOT SCHEDULED".

After processing all of the commands, sort the map first by value (dates in chronological order), then by key (events in alphabetical order). Print out all of the keys of the map in the new sorted order.

If you're confused on the specifics of how to implement this program, check the solution code files as the implementation differs slightly depending on the language.

Time Complexity: $O(N \log N)$ - this is the best we can do as even if we use a hash function to achieve $O(1)$ insertion and access for the map, the sorting at the end still takes at worst $O(N \log N)$ if we have exactly N events.

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

§14 Pie Predicament

The naive solution for this problem is fairly straightforward: for every pair of pies, check whether or not all of the flavors are between the two pies. However, this solution runs in $O(N^3)$ and times out, so some optimization is required.

A slightly more efficient solution would be to fix a pie on the left side and iterate from the fixed point until all of the pie flavors are included. Unfortunately, this solution is still too slow ($O(N^2)$), but we can make an important observation. For each fixed point on the left, we can use information computed for the previous fixed point. Each time you move your leftmost point, you encounter two cases: either all of your flavors are present (in which you don't have to do anything!) or you are missing one pie flavor. Thus if you are missing a pie flavor you can increment from the position of your rightmost point that corresponds with the previous leftmost point until you reach a pie with the flavor that you just removed.

Here, we introduce the central concept for this problem, which is two pointers. You can learn more about them here. We start with two pointers i and j at the first index of our pies array. From there, we first increment the j pointer until all of the pie flavors are included. Then we increment the i pointer once and check if all of the flavors are still present. If not, increment the j pointer until all of the flavors are included again. One way to check if all of the flavors are included is by keeping a counter array for each flavor and a counter for the number of pie flavors (decrement the number of pie flavors when a flavor reaches 0 in the counter array, and increment the number of pie flavors when a flavor goes from 0 to 1). This process is then repeated until j reaches the end of the pie array. The solution would be at the positions i and j where $j - i$ is minimized (the least amount of pies is kept) and runs in $O(N)$, which is fast enough to receive full credit.

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

§15 Candle Lighting

One thing to observe about this problem is that we only care about the end state. This means that we don't have to fully update the grid each time, but instead we could possibly leave some mark that represents a rectangle. Let's think of ways to do this in 1D, where for each row of each rectangle, we leave our mark on the row, and apply all the marks at once at the end.

A common technique for counting rectangles or line segments is known as the sweepline approach. We move from left to right, incrementing a counter each time a segment starts and decrementing it when a segment ends. At each point we visit, our counter represents how many segments go through our point. We can apply this strategy here, where for each query, we add a segment to each row that the query rectangle contains. After we apply all of the queries, the final state of a candle is only changed if there are an odd number of segments that go through that point.

A way to implement this type of approach is to add 1 to the point where a segment starts, and add -1 to the point directly after where a segment ends. At each point, we add the value at that point to our counter, then update the candle there based on the counter. This will cause our running counter to be increased by 1 for each point until the end of each segment, which is exactly what we want. The complexity of this is $O(NM + NQ)$. Without too bad of a constant factor, this will fully pass under our constraints.

However, notice what we're doing here. At any point, the counter represents the sum of the elements to the left of and including the point. This technique is called *prefix sums*, where the value at index i represents the sum of $A_{1\ldots i}$, or the sum of the prefix of the array up to i. All we're doing, for each segment $[L, R]$, is adding a 1 and a -1 to increase the sum of prefixes $L \ldots R$ by 1. So, can't we apply this strategy to the 2D problem? Certainly.

For our 2D prefix sums, $prefix[i][j]$ will store the sum of the first j columns in each of the first i rows (so the subrectangle with top left $(1, 1)$ and bottom right (i, j)). Similarly to the 1D solution, prefix sums will only be computed after all the queries have been applied. If either i or j is 0, then $prefix[i][j] = 0$. Otherwise, $prefix[i][j] = prefix[i-1][j] + prefix[i][j-1] - prefix[i-1][j-1]$, plus any value that we have at (i, j). It is left as an exercise to show why this includes each element exactly once. To represent a rectangle with top left (r_1, c_1) and bottom right (r_2, c_2), we add 1 to (r_1, c_1) and $(r_2 + 1, c_2 + 1)$, then subtract 1 from $(r_1, c_2 + 1)$ and $(r_2 + 1, c_1)$. This will increment the prefixes in our desired rectangle by exactly one, and not affect anywhere else.

Like the 1D solution, constructing a solution from these prefixes is simple. If the prefix sum at a point is odd, then it must be flipped, otherwise it is the same as it was initially. We can then simply count the number of candles that are on. Each query only causes 4 points to be added, and prefix sums take $O(NM)$ time, so the final complexity is $O(Q + NM)$.

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