Spring 2021 mBIT Advanced Editorial

June 12, 2021

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 Pokémon Permutation

Note that we only care about the frequency of each character. This is because if we know the frequency of each character in the Pokémon's name, we can just have its name be the repetition of characters in sorted order. For example, the following frequency table:

Character	Frequency
a	3
b	5
e	2

becomes the name `aaabbbbbee` (after all, the problem never said the unknown name had to be a legit Pokémon name).

Now let's find the frequency of each character in the input string, and store it in a table $freq$. If the name was repeated $k \geq 2$ times, then we know the name contains $\frac{freq[a]}{k}$ copies of `a`, $\frac{freq[b]}{k}$ copies of `b`, and so on, so k must evenly divide the frequency of each character. Since k can't be larger than the length of the input string, we could just check all possible candidates of k. Alternatively, we could just take the GCD of the frequencies of all characters. If the GCD is 1, then there is no way we can find a $k \geq 2$ that evenly divides the frequency of each character, so the answer is `IMPOSSIBLE`. Otherwise, we let k equal the GCD. After finding k, we'll print each character $\frac{freq[character]}{k}$ times as our answer.

Time complexity: $\mathcal{O}(N)$, where N is the length of the input.

Problem: Jeffrey Tong
Flavortext: Gabriel Wu
Editorial: Maxwell Zhang
Code: C++, Java, Python

§2 Azran Tablets

Observe that each end of s must also be the end of some segment that we topple. Since the ends of segments cannot lie within already-toppled segments, the segments must also be disjoint. Combining these two facts, if we sort the segments we choose, each segment sequentially covers a prefix of the remaining parts of s. This structure motivates a DP solution.

Let $dp[i]$ be the answer for the first i letters of s, which means $dp[0] = 0$. If the first i letters end with a character c, the last segment chosen must have c at both ends. Thus,

$$dp[i] = \min_{j<i:\ s_{j+1}=s_i} dp[j] + 1.$$

To compute these minimums efficiently, as we iterate from left to right, let

$$minPre[c] = \min_{j<i:\ s_{j+1}=c} dp[j]$$

for all characters c. Then, $minPre[c]$ can only change when $s_i = c$, and $dp[i] = minPre[s_i] + 1$. Note that this algorithm would still work if the problem used an array instead of a string.

Time complexity: $\mathcal{O}(N + M)$, where M is the size of the alphabet.

Problem: Jeffrey Tong
Flavortext: Timothy Qian
Editorial: Jeffrey Tong
Code: C++, Java, Python

§3 Goomba Grouping

First, let's determine when the answer is `-1`. After trying some cases on paper, we will find that Bowser's algorithm is always optimal when $N \leq 4$.

$N = 1$: The only possible partition is one Goomba on its own, for a difference equal to its own weight.

$N = 2$: For two Goombas of positive weight A and B, it's always optimal to place them in separate groups, since $|A - B| = \max(A - B, B - A) < A + B$, and Bowser's algorithm does just that.

$N = 3$: For three goombas of positive weights $A \leq B \leq C$, Bowser will place C in one group and A and B in the other. This is always optimal. Placing three goombas in one group gives the worst difference. Let's say we isolate B instead (proof is same for isolating A). Our difference becomes $A + C - B$. Compare that to isolating C for a difference of $\max(A + B - C, C - A - B)$, and we note that $A + C - B \geq A + B - C$ and $A + C - B \geq C - A - B$, so isolating C is always more optimal.

$N = 4$: For four goombas of positive weights $A \leq B \leq C \leq D$, Bowser will partition into $\{A, D\}$ and $\{B, C\}$ if $B + C > D$, and $\{A, B, C\}$ and $\{D\}$ otherwise. You can do more math to show that this is always optimal.

Now for $N \geq 5$ and any value of K, we can always construct a counter case to Bowser's algorithm. First, let's understand the flaw with Bowser's reasoning. Bowser assumes that it is always optimal to keep the two groups as even as possible at each intermediate step, so his logic breaks when it is optimal to place two large weights in one group, and many small weights that add up to the same amount in the other group. Knowing this, let's begin our construction with two large goombas each of weight 10^{18}, which we want to be in the same group in the optimal partition.

Since the optimal difference needs to be K, let's make the weights in the other group sum up to $S = 2 \cdot 10^{18} - K$. If N is odd, then we can fill the other group with goombas of weight $\left\lfloor \frac{S}{N-2} \right\rfloor$ or $\left\lfloor \frac{S}{N-2} \right\rfloor + 1$, such that they add up to S. Bowser's algorithm will place the two massive 10^{18} goombas into different groups, and because there are an odd number of the smaller goombas, his final difference will be of order of magnitude $\sim \left\lfloor \frac{S}{N-2} \right\rfloor$. Since $K \leq 10^9$ and $N \leq 20$, $\left\lfloor \frac{S}{N-2} \right\rfloor \gg K$, so this is ok.

What about even N? This can be remedied by adding a goomba of weight 1, which will hardly change the difference between the two groups from Bowser's algorithm because 1 is so small in comparison. So the even N case reduces to the odd N case. In fact, this observation also leads to an alternative solution, which is to simply pad the set with $N - 5$ ones, and then hard-code a solution for the $N = 5$ case.

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

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

§4 Cytus Craze

Consider $V = 0 \oplus \cdots \oplus (N-1)$. Then consider $a_i = i \oplus (i-1)$. This problem reduces to computing the number of subsets S of $\{a_1, \ldots, a_N\}$ that satisfy

$$\left(V \oplus \bigoplus_{x \in S} x \right) \leq K.$$

This is because including a_i in the subset is equivalent to hitting the beat, while excluding it is equivalent to missing the beat. It can be shown that $a_i = i \oplus (i-1) = 2^{v_2(i)+1} - 1$, where $v_2(i)$ denotes the number of 2's that divide i (so $v_2(24) = v_2(2^3 \cdot 3) = 3$ and $v_2(18) = v_2(2^1 \cdot 9) = 1$). This can be seen by considering the binary representation of i.

Let $c_i = \left\lfloor \frac{N}{2^i} \right\rfloor - \left\lfloor \frac{N}{2^{i+1}} \right\rfloor$, the number of integers j in the range $[1, N]$ such that $v_2(j) = i$. Let L be the largest number for which c_L is nonzero. We claim that for any j in the range of $[0, 2^{L+1} - 1]$, there are exactly $2^{c_0 + \cdots + c_L - (L+1)}$ subsets of $\{a_1, \ldots, a_N\}$ that XOR to j. Further, if j is not in this range, there are no such subsets. This is because only the parity of the number of occurrences of each $2^{i+1} - 1$ matters in the final XOR. Since the sum $c_1 + \cdots + c_L$ is simply N, each integer in the range $[0, 2^{L+1} - 1]$ can be created in exactly 2^{N-L-1} different ways.

Letting $K' := \min(K, 2^{L+1} - 1)$, our final answer is simply $(K'+1) \cdot 2^{N-L-1}$. More concisely, the answer can be written as $(K'+1) \cdot 2^{N - \text{bits}(N)}$, where $bits(N)$ is the number of digits in the binary representation of N, excluding leading zeroes. The answer mod $10^9 + 7$ can be computed using binary exponentiation in $\mathcal{O}(\log N)$ time.

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

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

§5 Knockout Tournament

Consider a fixed tournament. WLOG, sort the strengths so that $a_1 \le a_2 \le \cdots \le a_{2^N}$. We also arbitrarily break ties so that if two people with equal strength level face off the person with higher index wins. Then the unfairness is the sum of the differences between the strength of the winner and the loser of each match. Let's say the i-th person ends up winning w_i matches. Then their contribution to the unfairness is $w_i \cdot a_i - a_i$, except for the overall winner of the tournament, who will have contribution $w_i \cdot a_i$. But that person is always the person with the highest strength level. Thus, the total unfairness is just

$$a_{2^N} + \sum_{i=1}^{2^N} (w_i - 1) \cdot a_i.$$

To compute this expected value, we can use linearity of expectation. Consider person i, and say we want to compute the probability he wins at least j times. Then the tournament must be set up so that he is the strongest out of all 2^j competitors in his j-bracket. The probability of this happening is

$$\frac{\binom{i-1}{2^j-1}}{\binom{2^N-1}{2^j-1}}.$$

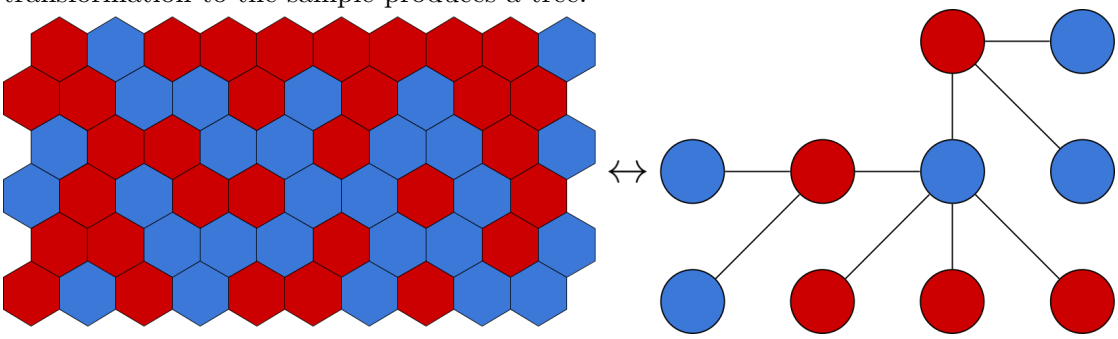
This is because there are $i-1$ people who are weaker than person i. We can calculate this in $\mathcal{O}(1)$ if we precompute factorials and inverse factorials. Since we can easily compute the probability that person i beats exactly j people, we can find the expected value of w_i in $\mathcal{O}(N)$. We iterate over all 2^N people, so overall this takes $\mathcal{O}(N2^N + 2^N \log MOD)$ time.

Time complexity: $\mathcal{O}(N2^N + 2^N \log MOD)$

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

§6 Squid Art

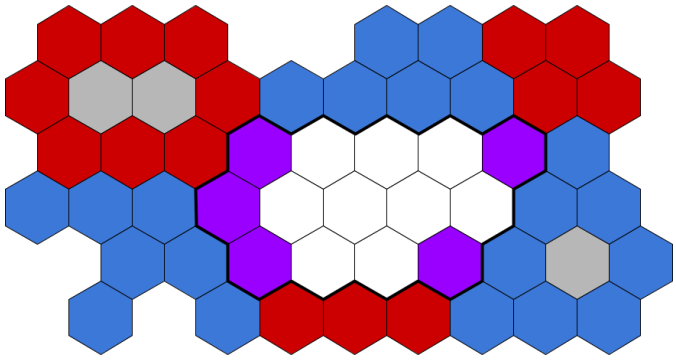
If we view each component of uniform color as a single vertex and draw an undirected edge between each pair of adjacent components, we can produce an undirected graph G equivalent to the original grid. By definition, G is bipartite. Notice that applying this transformation to the sample produces a tree:



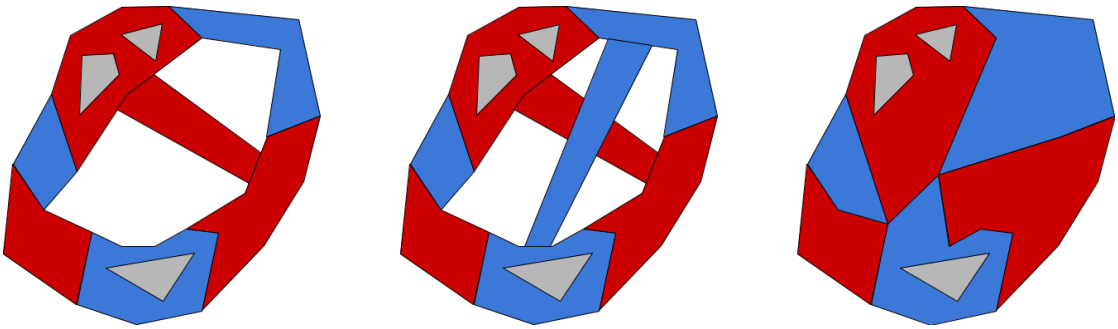
Theorem 6.1

G is a tree.

Proof. Assume for the sake of contradiction that G has a cycle C; since C must have an even length, it has at least two red and two blue vertices. The loop L in the original grid that transforms to C thus has at least two red and two blue components. There cannot be a central "hole" enclosed by L, because otherwise some cells (shown in purple) in the hole would be adjacent to components of both colors; these cells must actually be part of a component in L and thus not in the hole. Individual components can still have holes (shown in gray), but these do not affect the proof.



We now analyze the center of this filled loop. WLOG, consider any two red components in L: they cannot connect through the center, and they also cannot be cut off by a contiguous wall of blue cells (that would make two blue components connect through the center.) Thus, the only way the center of L can close up is that some pairs of same-color components touch, but only at single points:



In a hexagonal grid, however, no two cells touch at exactly one point, so the third situation is actually impossible, a contradiction.

Thus, G is a connected acyclic graph. □

Theorem 6.2

The minimum number of operations to turn the whole grid into a single color is the radius of G.

Proof. Consider the equivalent of filling in a component in the original graph on G: we switch the color of a vertex, which is equivalent to merging the vertex and its neighbors into a single vertex of the new color.

Let r be the radius of G. This pattern makes it easier to see that filling in the component corresponding to a centroid of G decrements r, and repeating the process r times reduces G to a single vertex, so r operations are sufficient.

To show that r operations are necessary, consider a diameter $D \subseteq G$. One operation can reduce at most 3 vertices (the chosen vertex and one neighbor to either side) of D to a single vertex, decreasing the diameter by at most 2. Since it is well-known that radius $= \lceil \frac{\text{diameter}}{2} \rceil$ for a tree, the radius is therefore decreased by at most 1. □

Computing the diameter and radius of a tree is a well-known problem solvable in linear time. There is a bit of casework dealing with parity to be done at the end because the problem requires all of the tiles to be blue (not just the same color).

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

Problem: Gabriel Wu
Flavortext: Timothy Qian
Editorial: Jeffrey Tong
Code: C++, Java, Python

§7 Scribble Roads

In "Connect the Graph," each player draws a new edge on their turn, and the first player to connect the graph into one component wins. Let's call edges that have not been drawn in yet and connect two vertices in the same component *useless*, and edges that have not been drawn but connect vertices of separate components *useful*. We claim that given a game configuration, the only information necessary to determine who will win is the current number of components with an odd number of vertices, the current number of components with an even number of vertices, and the **parity** of the number of useless edges.

We begin by reframing the game. Consider a scenario where we have components of sizes $c_1, \ldots, c_k$ and e useless edges. The locations of these e useless edges is irrelevant, so we can treat the game as if all k components are already complete (have no useless edges), with the existence of a global counter currently set to e. On a player's turn, they can choose to either reduce the counter by one (this corresponds to drawing a useless edge) or replace c_i and c_j with $c_i + c_j$ and raise the counter by $c_i c_j - 1$ (this corresponds to drawing a useful edge between components i and j). This is because one new useless edge is created for each pair of the form $\{\text{vertex in } i, \text{vertex in } j\}$, except the one which is used to connect the components in the first place.

Now, imagine the transition graph T between game states $S = \{\{c_i\}, e\}$. We have a node for each possible S, and we draw a directed edge from S to S' if a player's move in S can transition the game to S'. Now, define a new transition graph T' to be the original T, except for the purposes of transition edges it pretends that e in every state is always $e \pmod 2$. This results in many edges of T not appearing in T' (moving from an $e = 4$ state to an $e = 3$ state is now impossible), but every edge of T' appears in T. Note that both T and T' must be acyclic, so every state S in either graph must have a well-defined winner: either the "next player wins" or the "previous player wins". We now claim that if a player is the winner of state S in T', they are also the winner of state S in T. To win from state S in T, the player may simply follow their original winning strategy on T'. Every one of their moves will be valid, and if their opponent makes a move that didn't appear in T', they can immediately undo it. For example, if their opponent goes from $e = 4$ to $e = 3$, they can go directly to $e = 2$. Thus, we have shown that we only need the parity of the number of useless edges to determine the winner.

This also lets us demonstrate that we do not need to keep track of the sequence $c_1, \ldots c_k$ itself, only the number of odd components and the number of even components. Since we only care about the parity of e, when increasing e by $c_i c_j - 1$, we only need to know the parities of c_i and c_j. Notice that we never need the exact values of $c_1, \ldots, c_k$ anywhere else in this reframing of the game, so two components with the same parity can be treated exactly the same. Thus, the entire state of the game can be encoded in the number of odd components, the number of even components, and the parity of useless edges.

Now we can make a recurrence to solve the game. Let $dp[i][j][e]$ (for $0 \le i, j \le N$ and $0 \le e \le 1$) be `True` iff the next player to make a move will win, given that the game has i odd components, j even components, and e is the parity of the number of useless edges. Each dp state references up to three previous states (corresponding to merging odd-odd, even-even, and even-odd components), in addition to $dp[i][j][0]$ if $e = 1$. We set $dp[i][j][e]$ to be `True` iff at least one of these previous states is `False` (the next player can force their opponent into a losing state). Note that if i or j is 0, some of these transitions may

be ignored. At the end, we calculate number of odd components, even components, and useless edges in the original graph, then output the corresponding dp state. This gives us an overall $O(N^2)$ solution.

To speed this up, it helps to visualize the $N \times N \times 2$ grid of dp values. Notice that our transitions only reference "nearby" dp states, and it does so in a very predictable way. As such, we would expect a repeating pattern to form in the dp values for sufficiently large values of i and j, provable by induction. If you print this grid, you can easily see that such a pattern exists. It turns out that as long as one of i, j is larger than 5, all that is needed to predict $dp[i][j][0]$ and $dp[i][j][1]$ is $i \pmod 4$ (with some special cases for $j < 3$). Thus, it is possible to hard-code this pattern into your solution to answer queries in $O(1)$, after computing the starting numbers of odd and even components and useless edges (which can be done with DSU or DFS).

Time complexity: $\mathcal{O}(N + M)$
Problem: Claire Zhang
Flavortext: Timothy Qian
Editorial: Gabriel Wu
Code: C++, Java, Python

§8 Future Gadget Lab

Consider the set of leaves of the tree L. Consider g, which is the gcd of all elements in the set $\{\text{depth}(x)+1 | x \in L\}$. Now let $r \equiv J^K \pmod{g}$. We now know that after $T = J^K$ uses of the time machine, we must have that Hyounin is at depth equivalent to $r \pmod{g}$ in the tree no matter what.

Let p_u denote the probability that Hyounin is at node u if we randomly choose some number of usages t for the time machine such that $t \in [1, T]$, and we take T to ∞. p_u will eventually converge to a fixed value. For more details, we encourage the reader to look into Markov chains. Let q_r denote the probability that Hyounin is at a depth of $r \pmod{g}$ if we similarly choose a number of usages t for the time machine such that $t \in [1, T]$ and we take T to ∞. Note that no matter what, Hyounin always increments his depth $\pmod{g}$ in the tree by 1 every time he uses his time machine. So $q_0 = \cdots = q_{g-1} = \frac{1}{g}$, because the residues $\pmod{g}$ are uniformly distributed in $[1, T]$, and as T approaches ∞, everything will converge accordingly.

We that for $u \neq 1$, we note that we have

$$p_u = \frac{p_{\text{par}(u)}}{\text{number of children}(\text{par}(u))}$$

So therefore, we may perform DFS to express p_u for $u \neq 1$ in terms of kp_1 for some constant k. But then note that $\sum p_i = 1$. So therefore we can solve for $p_1, \ldots, p_N$. Now note that if we force ourselves to be at a depth that is $r \pmod{g}$, then we simply look at the timelines that are at a depth $r \pmod{g}$, and we may multiply their probabilities by g. This is due to the fact that $q_0 = \cdots = q_{g-1}$. So we can simply now directly compute the expected depth, where the overall time complexity comes from the DFS, which is $\mathcal{O}(N)$.

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

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

§9 Immortality Potion

First, we assume WLOG that $A \geq B \geq C$, and assume $A + B + C = 2$ for simplicity. We will demonstrate that we can create a solution of volume 1 using only cups $1, 2$ that has a ratio of $x : 1 - x$ of chemical r and chemical s (the remaining chemical is not present) for any $x \in [0, 1]$ with suitable precision in less than 140 moves. Say that we currently have a solution of volume 1 and ratio $y : 1 - y$ in cup 2, and cup 1 is empty. Then note that if we fill cup 1 with r completely and add it to cup 2, then we pour cup 2 into cup 1, and then empty cup 1, we now have a solution in cup 2 of volume one liter with ratio $\frac{y+1}{2} : 1 - \frac{y+1}{2}$. Similarly, if we repeat this process, but instead fill cup 1 with s in the beginning, we have a one liter solution in cup 2 of volume one liter with ratio $\frac{y}{2} : 1 - \frac{y}{2}$. If we view y as a decimal in binary, this is equivalent to adding a 1 or 0 after the decimal place in the binary representation of y. Therefore, if we compute the binary representation of x to a certain precision, we can make a solution of $x : 1 - x$ in less than 140 moves with precision of 10^{-6}. This bound of 140 is very loose.

Now, simply make a solution of ratio $1 - B : B : 0$ of volume 1 liter as shown above, and add it to cup 3. Then make a solution of ratio $1 - C : 0 : C$ of volume one liter, and add it to cup 3. Now we have a solution of ratio $2 - B - C : B : C$, which is just $A : B : C$. Note that this is possible because $B, C \leq 1$. Assume for contradiction this is not true. Then A could not be the largest of A, B, C if $A = 2 - B - C$. Therefore, we can make our desired solution in less than 300 moves.

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

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

§10 Luigi's Mansion

Consider Luigi's optimal path. It is easy to see that we can break the path into several parts, where in each part he moves down to some point in a hallway, and then moves back to the foyer. Let's label the hallways $1, 2, 3$, and label each of these parts in Luigi's path by their hallway number. Let the depth to which he traverses a hallway be the maximum room in the hallway he visits before turning back. Clearly, no two adjacent parts in Luigi's path should be labeled the same number, otherwise Luigi should have just continued down the hallway to obtain a better path. Furthermore, each time he returns to the foyer, it is clear that he should next go down the hallway that he has visited with the least depth. He must also make the depth to which he explores this hallway no longer his lowest explored depth.

Let $a_1, \ldots, a_k$ be the depths of the hallways that Luigi visits in each part of his path, in that order. Then the expected time that he arrives at King Boo can be broken in two parts: the time he wastes by going down a hallway and back without encountering King Boo and the time he takes to travel directly to King Boo. The latter part is constant regardless of $a_1, \ldots, a_k$ (it is $\sum_{i=1}^{N} 3 \cdot i \cdot p_i$ by linearity of expectation).

The former part is more difficult to account for. Call this remaining time *wasted time*. Let's say he's visited hallways $1, 2, 3$ with depths a, b, c respectively, with WLOG $a \le b \le c$. Then we can assume he visits the lowest depth hallway, and say he explores to a depth d. Then this exploration will take $2d$ time, and this time will only account for the total exploration time if King Boo is not in the first d rooms along the first branch, the first b rooms along the second branch, and the first c rooms along the third branch. So this contributes exploration will contribute $2d(1 - (P_d + P_b + P_c))$ in expectation, where we define $P_x = p_1 + \cdots + p_x$.

We will now use dynamic programming. Define $dp[i][j]$ as the expected minimum wasted time if the two highest depth hallways that Luigi has explored are i, j with $i \le j$. Note that we can actually ignore the lowest branch's exploration depth; this will be more clear when we write out the dp. There are two ways we could get to a state $dp[i][j]$. We could either come from $dp[k][j]$ $(k \le j)$, and then Luigi explores the lowest branch to a depth i, or we come from $dp[k][i]$ $(k \le i)$, and Luigi explores the lowest branch to a depth j. So we have

$$dp[i][j] = \min \left[\begin{array}{l} \min\limits_{1 \le k \le i} dp[k][j] + 2i \cdot (1 - (P_i + P_j + P_k)), \\ \min\limits_{1 \le k \le i} dp[k][i] + 2j \cdot (1 - (P_i + P_j + P_k)) \end{array} \right] \tag{1}$$

If we compute $dp[i][j]$ in increasing order of j and then increasing order of i, this leads to an $\mathcal{O}(N^3)$ solution. We note that this can be solved using Convex Hull Trick, which leads to an $\mathcal{O}(N^2 \log N)$ solution. Furthermore, the slopes that we insert into the Convex Hull Trick are monotonic, as well as our query points. So we can use a deque to perform Convex Hull Trick, which makes our solution $\mathcal{O}(N^2)$ overall.

An interesting thing to note is that we have observed by computation that the depths to which Luigi explores to are nondecreasing. This would only make the second argument of the min function necessary in the dp, but we have not formally shown that this is true. We would welcome any proofs or ideas that anyone has.

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

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

§11 Kirby's Buffet

Consider some fixed $[L, R]$ along with an X. Clearly, if an element a_i for $i \in [L, R]$ is a supermask of X, meaning that $a_i \& X = X$, it does not hurt to include a_i in our final subset. On the other hand, if a_i is not a supermask of X, we cannot include it in our subset. Thus, we reduce the problem to asking if the bitwise AND of all elements in $[L, R]$ that are supermasks of X is exactly equal to X.

Now say we break the array a_i into C blocks of roughly equal size. For each block, we iterate over all possible values of X, and we want to compute the bitwise AND of all elements in the block that are supermasks of X. To do this, we use bitmask dynamic programming. Let $A = 2^{17} - 1$. Consider an array dp indexed by $0, \ldots, A$, where $dp[i]$ is set to a null value by default (it is convenient to use $2^{18} - 1$). For all elements x in the current block, set $dp[x] = x$. Now we iterate masks downwards from A to 0. Say we are at a mask of x. Then for every set bit b in x, we do $dp[x - 2^b] = dp[x - 2^b]$ & $dp[x]$. One can show that by the end of this process, $dp[x]$ will contain the value of the bitwise AND of all elements that are supermasks of x in the current block. This works in $\mathcal{O}(A \log A)$ for each block.

We can solve the full problem with Q queries in $\mathcal{O}\left(Q\left(C + \frac{N}{C}\right)\right)$ using square root decomposition. For each query, simply find the bitwise AND of $dp[X]$ for all blocks fully contained in $[L, R]$ (there are $O(\frac{N}{C})$ of these), then individually iterate over the a_i in the interval which have not yet been accounted for (there are $\mathcal{O}(C)$ of these). Therefore, this solution works overall in $\mathcal{O}\left(\frac{N}{C} \cdot (C + A \log A) + Q\left(C + \frac{N}{C}\right)\right)$. Noting that $Q \approx N \approx A$, we can refer to all three of these variables as N. This function is minimized if we choose $C = \sqrt{\frac{N}{\log N}}$, obtaining the final complexity of $\mathcal{O}(N\sqrt{N \log N})$. This solution also has a very low constant factor, so it works very fast in practice.

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

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

§12 Pillar Path

First, we try to solve the problem if all costs to remove circles are infinite. Let S denote the start and E denote the end. A nice simplification we can do is treat the start and the end like circles of radius 0.

Theorem 12.1

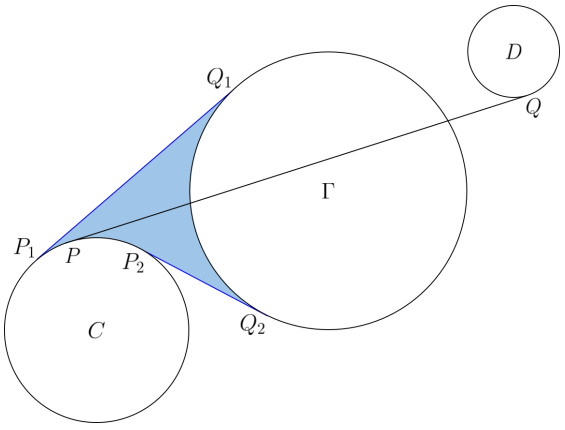
The shortest path from S to E consists only of tangent line segments between two circles and paths along a circle's circumference.

Proof. Appeal to physics: Imagine if the shortest path was a string from S to E. Now pull the string taut. Clearly pulling it taut decreases the length of the path, and it will get pulled taut until all parts of the string are tangent lines between two circles or paths along a circle's circumference. □

Consider creating all $\mathcal{O}(N^2)$ tangents between all pairs of circles. We leave it as an exercise to the reader to figure out how to compute these tangents (it takes some geometry). Let the endpoints of this collection of tangents be vertices on a graph, and draw an edge for each tangents. We can then construct edges connecting adjacent vertices around the circumference each circle. In this manner, we may run Dijkstra's algorithm in $\mathcal{O}(N^2 \log N)$.

The issue with this solution is that we must check if each tangent intersects another circle. If it does, we may not use it in our path. To do this, consider a central circle C. We examine at all tangents to C from the other circles, and want to determine which of these tangents do not intersect any other circle. Consider another circle Γ. We can split the tangents of Γ into two types, *clockwise* tangents and *counterclockwise* tangents. If we treat the tangents as rods and imagine pulling on them, clockwise tangents are those that would cause the circle to rotate clockwise, and counterclockwise tangents are those that would cause it to rotate counterclockwise. We limit ourselves to only looking at clockwise tangents for now. We may handle counterclockwise tangents similarly later.

The following diagram shows the two clockwise tangents from central circle C to circle Γ.



Now we view a clockwise tangent PQ with P on C and Q on another circle $D \neq \Gamma$ by $\vec{Q} - \vec{P}$, and we map this tangent to a point represented by (θ, d), where θ and d are the argument and magnitude of $\vec{Q} - \vec{P}$. Say PQ intersects circle Γ. As in the figure above, let P_1Q_1 be mapped to (θ_1, d_1) and P_2Q_2 be mapped to (θ_2, d_2). First, note that θ must

lie in the range $[\theta_2, \theta_1]$ if PQ is to intersect Γ. Furthermore, we claim that (θ, d) must lie directly above some point on the line segment between (θ_2, d_2) and (θ_1, d_1) if we view these as normal points in the Cartesian plane (replacing the x-axis with the θ-axis and the y-axis with the d-axis)[1]. To justify this, we note that if we move a point from Q_2 to Q_1 on the inner arc of Γ, the length of the tangent to C is a concave up function with respect to θ. Similarly, if we move a point from Q_2 to Q_1 on the outer arc of Γ, the length of the tangent to C is a concave down function. This can be formally shown by power of a point. Thus, we must have the point representation of PQ in θd-space lie above the curve U modeled by $(\alpha, len(\alpha))$ for α in $[\theta_2, \theta_1]$, where $len(\alpha)$ is the length of segment tangent to C with argument α and endpoints on C and the outer arc of Γ. Since U is concave down, it lies above the straight line between its endpoints. In summary, we have shown that (θ, d) must lie above some point on the segment between (θ_2, d_2) and (θ_1, d_1).

Consider the other clockwise tangent from C to D, and let it be called XY. Now let the line segment between the point representations of PQ and XY be called segment a, and the line segment between (θ_1, d_1) and (θ_2, d_2) be called b. All of this is being done in the θd Cartesian plane. a and b can't intersect because all circles are nonintersecting. If an endpoint of a is directly above a point on b, then no endpoint of b can be directly above an endpoint of a. Thus, we may draw an "arrow" from a to b signifying a is "above" b. If we do this for all segments formed by the point representations of the tangents for each circle, we obtain a directed acyclic graph. There are clearly no cycles because the line segments must be straight[2]. Therefore, we can perform a topological sort on these segments to find an ordering of circles in which the tangent segment from C to A can only intersect B if A is before B in the ordering.

Now how can we find this topological ordering? We can construct the necessary arrows using an algorithm like Bentley Ottman, or the one used in Cow Steeplechase II from the USACO Silver 2019 US Open. This will work in $\mathcal{O}(N \log N)$, and the topological sort will work in $\mathcal{O}(N)$.

Now we add the circles in reverse topological order. At the same time, we maintain a segment tree (or Fenwick tree) indexed on the angles. Each time we add a circle with segment endpoints $(\theta_1, d_1), (\theta_2, d_2)$, we update the interval represented by $[\theta_2, \theta_1]$ (making sure to take the shorter arc interval). We check if each endpoint intersects any other interval, and if so, we know that tangent intersects a previous circle. This works in $\mathcal{O}(N \log N)$ after performing array flattening on the angles.

Now back to the original problem. How do we take care of non-infinite costs? Instead of making tangents unusable if they intersect a circle, we simply add the costs of the circles intersected by the tangents to the "distance" of the tangent. The previous solution is easily amended to solve the new problem by performing range add updates to intervals in the segment tree. Since we perform two angle sweeps (one clockwise and one counterclockwise) for each circle, the overall algorithm runs in $\mathcal{O}(N^2 \log N)$.

Side Note: This problem was meant to be extremely difficult both to solve and implement. The coding can be made slightly easier by a well-built library and templates, but the

[1] To visualize this mapping from tangents to points, it is helpful to note that the blue area in the diagram corresponds to some area directly underneath the line segment between (θ_1, d_1) and (θ_2, d_2) in θd-space (although it is more strictly bounded by the concave up function below the segment).

[2] The only exception to this is a cycle formed by segments that span all 2π radians, and this can be dealt with by splitting all segments that cross an arbitrary "branch cut" in two.

ideas in this problem are not easy. It also may be possible to squeeze asymptotically slower $\mathcal{O}(N^3)$ solutions under the time limit due to the heavy constant factor involved in our solution, but we tried our best to prevent that. Also note that our model solution has small bug which we have not located, likely due to precision issues (we identified and fixed an incorrect test case during the contest, so nobody lost points because of our mistake).

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

Problem: Gabriel Wu and Timothy Qian
Flavortext: Timothy Qian
Editorial: Timothy Qian
Code: C++

§A Among Us References

Just for fun, we hid three references to the infamous game *Among Us* in our problem sets. Each reference was in a problem shared by the Standard and Advanced divisions.

- In *Pokémon Permutation*, the first letter of each sentence in the first paragraph spells `AMONGUS`.
- In *Goomba Grouping*, the sample output for the second test case contains eight numbers. If you convert these numbers to letters (1=`A`, 2=`B`, ..., 26=`Z`), it spells `IMPOSTOR`.
- The second pretest in *Squid Art* depicts a character from the game:

```
19 30
11111111111000000000011111111
11111111000000000000000011111
11111111000111111110000000001111
11111110001110000000000000111
11111110001100000111111000001
11000000011100000111111111001
10000000011100000000000000001
10001100011100000000000000001
10001100011110000000000000011
10001100011111111111111100011
10011100011111111111111100011
10011100011111111111111100011
10001100011111111111111100011
10000100011111111111111100011
11000000011111000000001000111
11111110011111000100011000111
11111110011111000100011000111
11111110000000000100000001111
11111110000000000111111111111
```