# 2021 Rocky Mountain Regional Programming Contest

# Solution Sketches

# Credits

- Darko Aleksic
- Darcy Best
- Howard Cheng
- Ryan Farrell
- Zachary Friggstad
- Brandon Fuller
- Noah Weninger

# A - Betting (44/47)

- For each option, the answer is simply $\frac{100}{p}$ where $p$ is the percentage bet on that option.

- To lose an election, you can afford to win as many as $\lfloor \frac{N}{2} \rfloor$ regions—assume you win all votes in these regions.
- For the remaining regions, you may win up to $\lfloor \frac{p}{2} \rfloor$ votes and still lose.
- Greedy algorithm: win the regions with the $\lfloor \frac{N}{2} \rfloor$ highest population.

- If there is a gap of length $g$ between consecutive people, we can fit $\lfloor (g-1)/2 \rfloor$ more people in that gap.
- Sum this value over all gaps.
- Don't forget about the gap between the last and first person in the input.

# H - RSA Mistake (17/108)

- Factor both numbers using trial division up to the square root. Takes $O(\sqrt{n})$ time to factor $n$. Fast enough for this problem as both numbers are $\leq 10^{12}$.
- If either number is divisible by a prime more than once or if the two numbers share a prime in common: `no credit`.
- Otherwise, if either number is not a prime: `partial credit`.
- Otherwise, `full credit`.

# D - Pawn Shop (16/100)

- Scan left-to-right through both arrays at once.
- Maintain a frequency counter as you scan: $freq[x]$ is the difference between the number of copies of item $x$ scanned so far from the first array minus the number of copies of $x$ scanned so far from the second array.
- Also maintain a value $\Delta$ indicating how many keys $x$ are such that $freq[x] \neq 0$.
- If $\Delta$ ever becomes 0, place a divider.

- Simulate the algorithm and remember a “timestamp” so that each time $s$ or $e$ is incremented, the timestamp is incremented.
- In other words, the timestamp counts the number of windows encountered so far.
- During the simulation, record the timestamp at which $w_i$ enters the window (i.e. when $e = i$)
- When the window leaves $w_i$ (i.e. when $s = i + 1$), compute the difference in timestamps.

# L - Ticket Completed? (11/54)

- Create a graph $G$ with the $n$ cities as vertices and any claimed rail segments as edges.
- Find the connected components $C_i$ within the graph (e.g. using BFS or DFS).
- For each connected component of $k$ vertices, there are $\binom{k}{2}$ destination tickets that can be satisfied.
- The probability that a random pair of cities will be connected is the total number of satisfied destination tickets (across all connected components) divided by the total number of unique destination tickets:

$$\frac{\sum_{C_i} \binom{|C_i|}{2}}{\binom{n}{2}}$$

- For each candidate word in the dictionary:
  - Check whether each guess' feedback is consistent given the candidate word.
  - If the feedback is consistent for all guesses, output the word.

# G - Loot Chest (6/10)

- Recall: the expected number of times you need to flip a coin until you see heads if the coin has probability $p$ of being heads is $1/p$.
- So the expected number of times you need to open a prize pack is $1/(G/100)$.
- Just need to compute the expected number of games until you open a prize pack.
- **Dynamic Programming**: If $e[P]$ is the expected number of games until you open a prize pack given that your current probability of getting a pack is $P$ is then:
  - $e[100] = 1/(1 - L/100)$ (keep playing until you win)
  - $e[P] = 1 + \frac{L}{100} \cdot e[P + \Delta_L] + \left(1 - \frac{L}{100}\right) \cdot \left(1 - \frac{P}{100}\right) \cdot e[P + \Delta_W]$ for $0 \leq P \leq 99$. That is, you play a game. If you lose, $P$ goes up by $\Delta_L$ and if you win but don't get a prize pack, then $P$ goes up by $\Delta_W$. Make sure to cap the new $P$ value at 100 (eg. use $\min(100, P + \Delta)$ whenever $P$ goes up by $\Delta$).

- Single-step simulation is too slow.
- Look for patterns. For example, if all three are distinct, say $A < B < C$, then we can simulate $\Delta := \min B - A, C - B$ steps in a single calculation: subtract $\Delta$ from $B$ and $2\Delta$ from $A$. After this, two values are the same.
- If two values are the same, they will follow the same pattern until they are within, say, 4 of each other (or some get close to 0). Example: $A = B = 80$, $C = 100$. Every 2 rounds, $A$ and $B$ will go down by 1 and $C$ by 4 until $C$ is within 1 of $A$, $B$.
- If they are within 4 of each other, just do single step simulation until they are within 1 of each other.
- If all 3 have the same health: `Rubble!`

# J - Snowball Fight (3/8)

- If they are within 1 of each other, every 3 rounds each will go down by 3.
- If there are only 2 left, easy to tell.
- If two of them have small health (say $\leq 4$), then you should just simulate to avoid corner cases in the big-step simulation rules.
- Carefully combining these ideas leads to a solution with running time $O(1)$. Just be extra careful to get the details right!

# F - Protect the Pollen! (1/1)

- The flowers (nodes) and vines (edges) can be represented as a graph. In fact it is a tree.
- We can solve this recursively on the tree.
- For each node $r$, define $f(r, s, b)$ as the largest total pollination power possible for the subtree rooted at $r$ and the total size of the selected families is $s$.
- $b$ is a boolean flag indicating whether the root $r$ must be skipped (e.g. if parent node has been chosen).

# F - Protect the Pollen! (1/1)

- At each node, combining the answers from subtrees is essentially a knapsack problem.
- This can be solved in $O(NS^2)$ time.

# K - Team Change (1/2)

- Consider a graph $G$ with vertices == players and edges == conflicts.
- Label each vertex as **must change**, **must not change**, and **doesn't matter**.
- After deleting some players, it is possible to form teams if and only if each component of the resulting graph does not have both a **must change** and a **must not change** player.
- Cast as a min-cut problem where you cut vertices. Create 2 new nodes $C, N$ representing **change** and **not change.** Connect $C$ to each vertex that must change, $N$ to each vertex that must not change, and find a min-size $N - T$ vertex cut.
- Input was small enough that even Ford-Fulkerson is fast enough.

# M - Trade Routes (1/5)

- The greedy algorithm is correct: process the routes $i$ in order of value (greatest to least). If adding $i$ to the current set of chosen routes is feasible, do it.
- But that is too slow.
- Idea: push the solution "upward". For each vertex $j$, compute the optimal solution for nodes lying in the subtree under $j$ (i.e. as if $j$ was Rome) and store in an ordered set $R_j$
- To compute $R_j$ for $j$, take the $b_j$ most valuable items in $\{j\} \cup_{j' \text{ child of } j} R_{j'}$ (or all of them if there are less than $b_j$).
- This is still too slow.

# M - Trade Routes (1/5)

- The final trick is when merging two sets, say $R_j$ and $R_{j'}$, to always add the items from the smaller of the two to the larger and regard the larger as the new merged set.
- Each item is "moved" to a new set $O(\log n)$ times since the size of the resulting set is at least twice as large as the original set. Each movement takes $O(\log n)$ time if you use an ordered set (or a binary heap). So $O(n \cdot \log^2 n)$ time in total.
- Can do in $O(n \cdot \log n)$ times using heaps that support $O(1)$ insertion, but they aren't in standard libraries. The above idea is fast enough.

# B - Antialiasing (0/7)

- For each query, there are four boundary segments for that pixel.
- Repeatedly clip given polygon against the four segments.
- Exact arithmetic needs to be used.