

Intra-IUB Programming Contest - Autumn 2025

Problem Analysis

Sajed I Haque

Independent University - Bangladesh

April 11, 2026

Problem Statistics

Problem	# Attempts	# Solves
A - A Tale of Two Merchants	37	?
B - Balanced Duels	17	?
C - Course Outcomes	10	?
D - Dark Streets	13	?
E - Exam Admittance	37	?
F - Fuel Shortage	12	?

A - A Tale of Two Merchants

- The objective is to check whether the characters on one string can be rearranged to form the other string.
- One simple approach is to sort the characters in the two strings and see if the sorted results match.
 - C++ strings are stored as vectors, so they can be sorted using `std::sort` from the `algorithm` STL
 - Python strings can be sorted using `sorted` (which also converts the strings into lists, but this should not affect the solution)
 - Python strings can also be converted to lists first in order to use the `sort` method in the `list` type.

A - A Tale of Two Merchants (continued)

- Another approach (which does not involve sorting) is to carefully track the frequencies of each letter and ensure they are the same for the two strings.
 - The frequencies can be stored using a dictionary/map (from letters to frequency) or an array of size 26 (each entry corresponding to a letter).
 - Note that it is possible for one string to have a letter that does not exist in the other string, so the checking should either be done in both directions or go through all 26 letters.
- Comparing the lengths of the strings or comparing the sets of characters may help in rejecting some pairs. However, most approaches that handle trickier cases should usually work even without a separate check for the length or the character sets.

E - Exam Admittance

- Seems very simple at first glance, where the total number of classes is $P + A$, and we need to check if P is at least 60% of that. In other words, we need to check:

$$\frac{P}{P + A} \geq \frac{60}{100}$$

- However, performing division would produce a floating-point number, which has limited precision. When dividing very large numbers, the inaccurate result may cause the verdict to be the opposite of what it should be.
 - The sample test case of 5999999999999999999 4000000000000000001, whose answer should be NO, would probably have an output of YES when running naive implementations.
- To avoid these issues, the comparisons should involve exact numbers, i.e., avoid comparing floating-point numbers.

E - Exam Admittance (continued)

- Through cross-multiplication, we can observe that

$$\begin{aligned}\frac{P}{P+A} &\geq \frac{60}{100} \implies 100P \geq 60(P+A) \\ \implies 5P &\geq 3(P+A) \implies 2P \geq 3A\end{aligned}$$

- Even the first step should be sufficient to avoid floating-point errors, as it now involves comparing integers.
- For C++ programmers, the integer type is important. The `int` type would fail to even store large inputs. The `long long` type can overflow for computations like $100P$, but should work with a more simplified comparison (e.g., $5P \geq 3(P+A)$), even if signed.
- On the other hand, C++ programmers can use `long double` to perform the divisions, bypassing the need to cross-multiply.

C - Course Outcomes

- The scenario might sound complicated, but ultimately, the problem simply asks to multiply two given matrices.
- We can check if the number of columns in the first matrix matches the number of rows in the second matrix. If they do not, then the multiplication is invalid.
 - However, make sure you still read all the input pertaining to the two matrices in these invalid cases, before moving on to the next test case.
- When multiplying an $M \times N$ matrix with an $N \times Q$ matrix, the resulting matrix would have dimensions $M \times Q$. This would already require two nested `for` loops (iterating M and Q times respectively)
- Furthermore, computing the element at row i and column j of the result would require check all elements in row i of the first matrix and all elements in column j of the second matrix, so a third nested loop is required (iterating N times).

F - Fuel Shortage

- There may be a lot of input variables, but the task itself is relatively straightforward.
- We start with S fuel. For each refueling station, there are three steps:
 - Travel to the refueling station, causing the fuel to decrease by D . This is not possible if fuel becomes negative.
 - Refuel at the station, causing fuel to increase by R . However, if the result exceeds M , the fuel is capped at M .
 - Return home, causing the fuel to decrease by D . This is not possible if fuel becomes negative.
- We can simulate the three steps for each of the refueling stations and then choose the largest result.
- Note that staying home is an option as well. A simple way to incorporate this is to initialize the base solution to S before comparing with the results from each station.

D - Dark Streets

- Since the objective is to ensure that every row and every column has a light, we only need to install new lights on rows/columns that have no lights.
- Given a row/column, we can simply use a `for` loop to check every light in the row/column; if all values are 0 for some row/column, then this row/column has no lights (which we can refer to as a “dark” row/column).
- This needs to be done for all rows and columns. One option is to make two separate arrays, one for the rows and the other for the columns, such that if we find a light at row i and column j , then neither row i nor column j are dark, so we can update the corresponding array entries (e.g., to `True` or 1 or simply add 1)

D - Dark Streets

- Note: it is sufficient to determine how many dark rows and how many dark columns there are. We do not need to know exactly which rows/columns are dark.
- As for how many lights to install, each new light can be placed at the intersection of a dark row (if any) and a dark column (if any), thus illuminating both the row and the column together with a single light.
- We can apply this for each new light until we either run out of dark rows or run out of dark columns, then we simply need enough lights to cover the remaining of the dark rows or columns.
- In other words, the minimum number of lights required is simply the larger between the number of dark rows and the number of dark columns.

B - Balanced Duels

- Given an input string composed of Bs and Fs, we need to determine the length of the longest substring that has an equal number of Bs and Fs.
- Checking every substring is too slow; for a string of length N , there are $\Theta(N^2)$ substrings, and checking each of them independently whether they are balanced would take $\Theta(N^3)$ time.
- We can speed this up to $\Theta(N^2)$ by iterating over the starting points first, and then we can iterate over the ending points quickly by only adjusting our counts based on the new ending character being added.
- We can further observe that it's not necessary to actually count the number of Bs and Fs in each substring, as we can simply store the difference between the number of Bs and Fs. A difference of 0 means the substring is balanced.
- Although $\Theta(N^2)$ is still too slow, these two ideas can help refine our strategy to improve efficiency even further.

B - Balanced Duels (continued)

- The idea of fixing a starting point and extending the ending point suggests that we can efficiently pre-compute the result for every prefix of the string. We only need to compute the difference between the number of Bs and Fs, allowing us to construct a prefix difference array.
- A balanced substring is one where the difference is 0. If some entry of the prefix difference array is 0, it means the prefix up to this index is a balanced substring. The rightmost 0 would be the largest balanced prefix.
- But what about substrings that are not prefixes? For the substring from index L to R , the prefix difference at index $L - 1$ must be equal to the prefix difference at index R in order for the substring from L to R to be balanced.

B - Balanced Duels (continued)

- In other words, if a value x appears twice in the prefix difference array, then these two points correspond to a balanced substring.
- For the longest balanced substring, we can restrict our attention to only the first x and the last x in the prefix difference array.
- We can use a map/dictionary that maps x to the indices of the prefix difference array that contain x , allowing us to find the longest balanced substring enclosed by prefix differences of x .
- Since the difference can only change by 1 with each index, the number of distinct possible values of x range from $-N$ to N .
 - We can further observe that a value of x that's greater than $N/2$ or less than $-N/2$ can never show up twice, so they do not help with finding balanced substrings. This is not required, but speeds up the implementation slightly.

B - Balanced Duels (continued)

Overall Algorithm

- Initialize a dictionary mapping difference values to arrays/lists of indices.
- Scan the array, updating the prefix difference for each index (no need to actually store a prefix difference array).
 - We add each index to the array mapped (in the map/dictionary) by the prefix difference arising at that index.
- After the first scan, we then check each map/dictionary entry. Each entry is an array of indices, with the first and last indices corresponding to the largest balanced substring enclosed by the prefix difference.
- The longest index range is the length of the longest balanced substring.

The runtime becomes $\Theta(N \times \text{Dictionary Lookup})$, which becomes $\Theta(N \log N)$ for ordered maps and has an average runtime of $\Theta(N)$ for unordered maps.