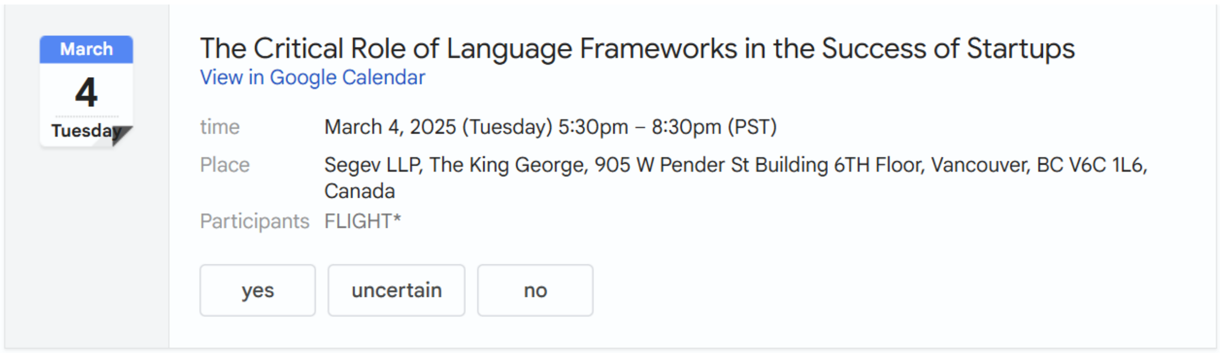
#Q1:

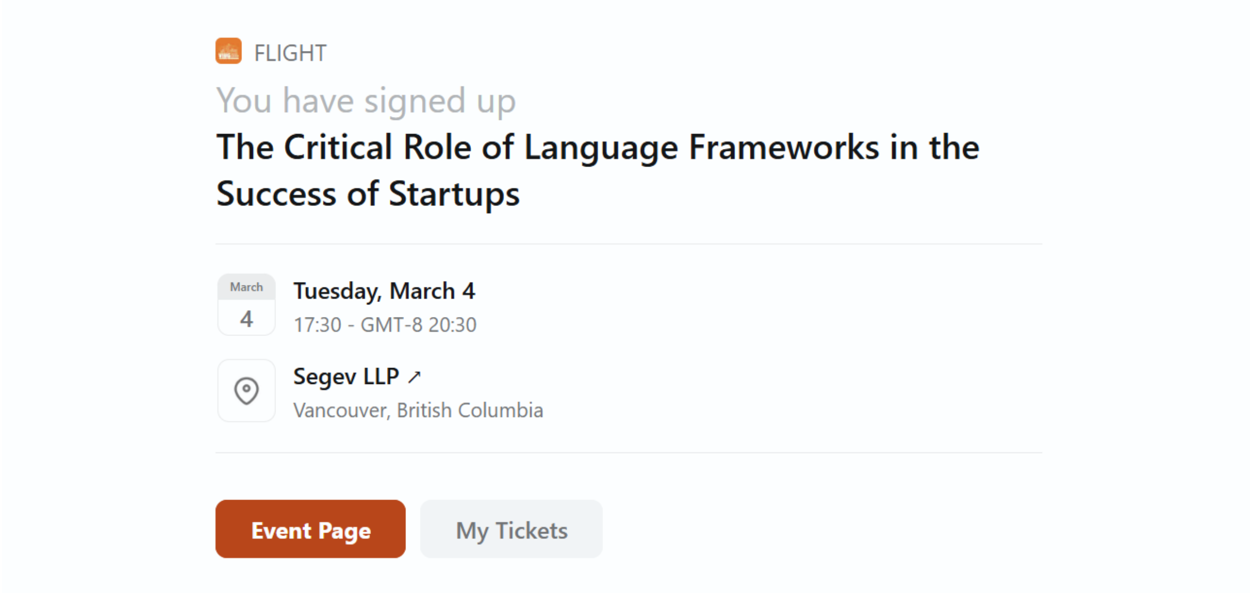


March
4
Tuesday

The Critical Role of Language Frameworks in the Success of Startups
View in Google Calendar

time	March 4, 2025 (Tuesday) 5:30pm – 8:30pm (PST)
Place	Segev LLP, The King George, 905 W Pender St Building 6TH Floor, Vancouver, BC V6C 1L6, Canada
Participants	FLIGHT*

yes | uncertain | no



FLIGHT

You have signed up

The Critical Role of Language Frameworks in the Success of Startups

March 4

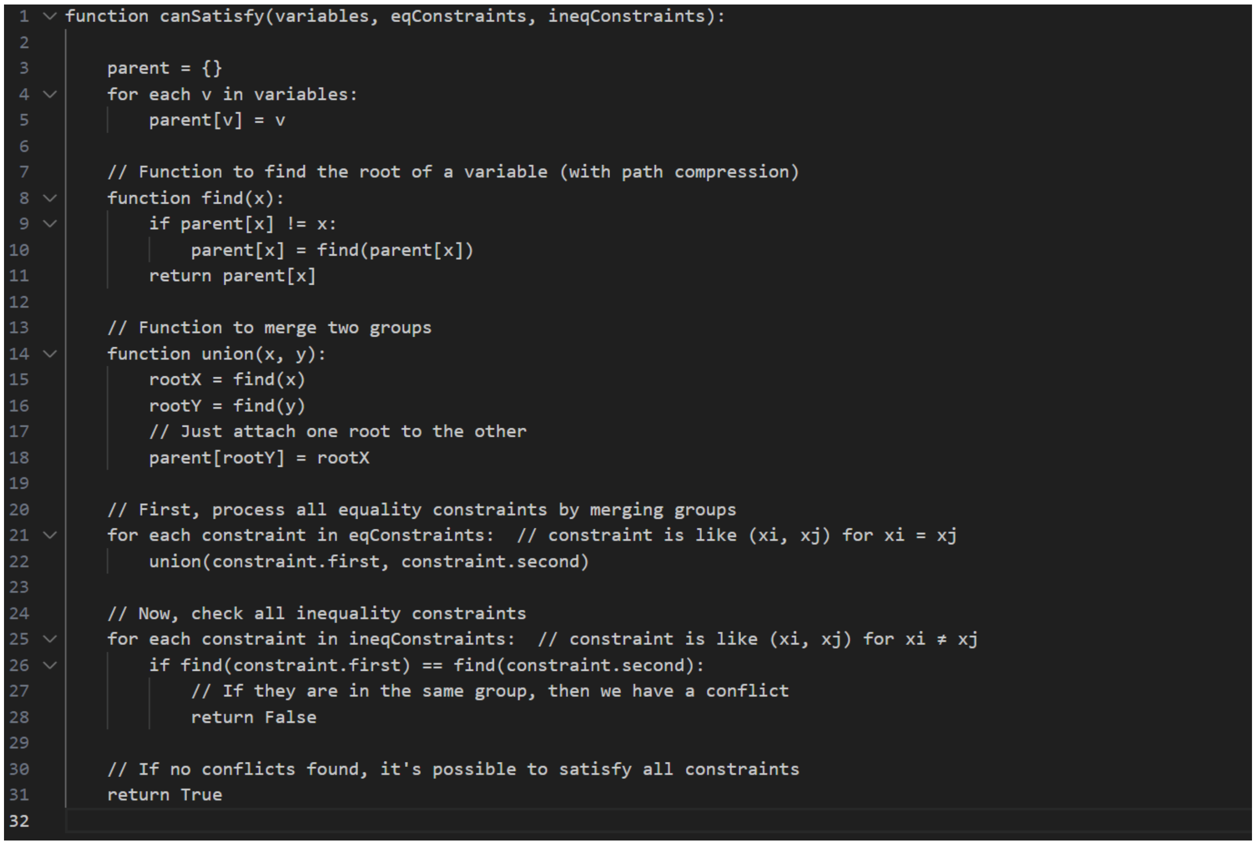
Tuesday, March 4
17:30 - GMT-8 20:30

Segev LLP ↗
Vancouver, British Columbia

Event Page | My Tickets

#Q2:

a)


```
function canSatisfy(variables, eqConstraints, ineqConstraints):

    parent = {}
    for each v in variables:
        parent[v] = v

    // Function to find the root of a variable (with path compression)
    function find(x):
        if parent[x] != x:
            parent[x] = find(parent[x])
        return parent[x]

    // Function to merge two groups
    function union(x, y):
        rootX = find(x)
        rootY = find(y)
        // Just attach one root to the other
        parent[rootY] = rootX

    // First, process all equality constraints by merging groups
    for each constraint in eqConstraints:  // constraint is like (xi, xj) for xi = xj
        union(constraint.first, constraint.second)

    // Now, check all inequality constraints
    for each constraint in ineqConstraints:  // constraint is like (xi, xj) for xi ≠ xj
        if find(constraint.first) == find(constraint.second):
            // If they are in the same group, then we have a conflict
            return False

    // If no conflicts found, it's possible to satisfy all constraints
    return True

```

First, we set up a union-find structure to merge variables that have to be equal. Then we just check every inequality to see if the two variables end up in the same group. If they do, it's not possible to satisfy everything. Otherwise, it works.

b)

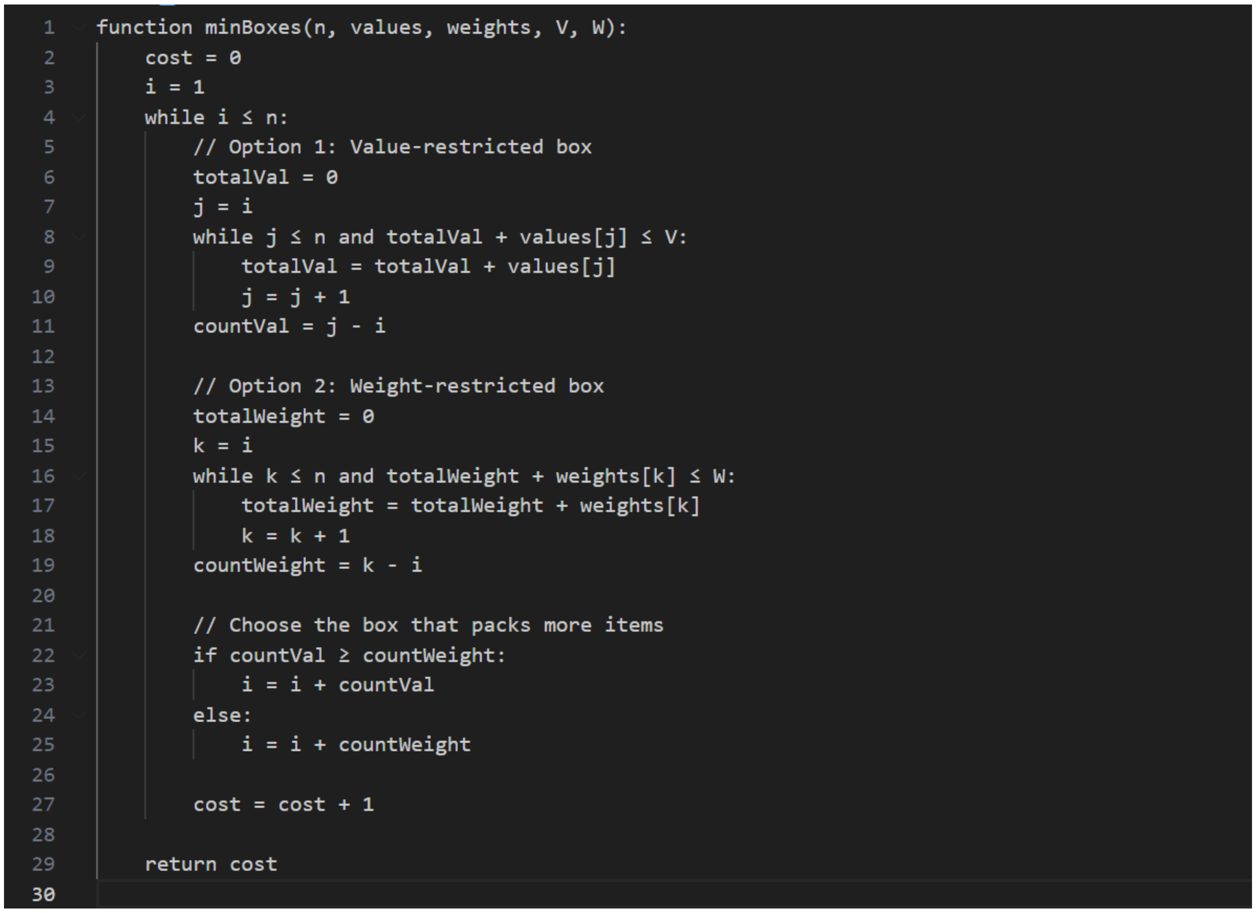
The initialization takes O(n) time. Processing m_1 equalities and m_2 inequalities takes $O(m_1 + m_2)$ time (amortized nearly constant per operation), so the overall running time is $O(n + m_1 + m_2)$.

Q3

a)

Beginning with the initial item, the greedy algorithm seeks to fit as many things as it can into a value-restricted box while staying within the upper limit of the permitted value. It then determines how many objects can be placed inside a weight-restricted box without going over the weight limit. The algorithm chooses the box type that can accommodate more items by comparing the amount of items that can be packed in each type of box. After then, it advances to the following set of items and raises the shipping cost. In order to minimize the amount of boxes utilized, this process is repeated until all things are packed into boxes.

b)

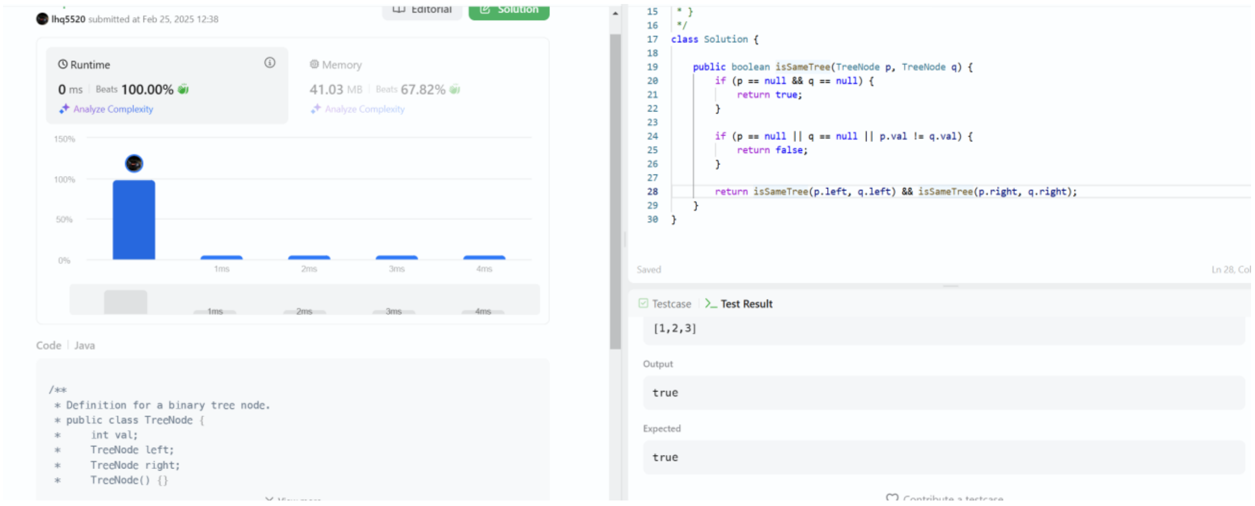


```
function minBoxes(n, values, weights, V, W):
    cost = 0
    i = 1
    while i ≤ n:
        // Option 1: Value-restricted box
        totalVal = 0
        j = i
        while j ≤ n and totalVal + values[j] ≤ V:
            totalVal = totalVal + values[j]
            j = j + 1
        countVal = j - i

        // Option 2: Weight-restricted box
        totalWeight = 0
        k = i
        while k ≤ n and totalWeight + weights[k] ≤ W:
            totalWeight = totalWeight + weights[k]
            k = k + 1
        countWeight = k - i

        // Choose the box that packs more items
        if countVal ≥ countWeight:
            i = i + countVal
        else:
            i = i + countWeight

        cost = cost + 1

    return cost
```

Each item is considered only once because the inner loops move the index forward without rechecking items. This means the overall running time grows linearly with the number of items, i.e., O(n).

Q4:

a)


lhq5520 submitted at Feb 25, 2025 12:38

Runtime 0 ms Beats 100.00%

Memory 41.03 MB Beats 67.82%

Code | Java

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
```

```
 * }
 */
class Solution {

    public boolean isSameTree(TreeNode p, TreeNode q) {
        if (p == null && q == null) {
            return true;
        }

        if (p == null || q == null || p.val != q.val) {
            return false;
        }

        return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
    }
}
```

Testcase | Test Result

```
[1,2,3]
```

Output

```
true
```

Expected

```
true
```

b)

When I originally started working on the "Same Tree" problem, I thought about utilizing a queue or stack to solve it iteratively. Comparing similar nodes while performing a level-order or depth-first traversal was the concept. But it became difficult to handle the situations where one node was null while the other wasn't. The approach was overly complicated due to the additional bookkeeping required for pushing and popping nodes. Although the iterative method was effective, it was more difficult to troubleshoot and felt counterintuitive.

After struggling with the iterative approach, I realized that recursion might be a better fit. Trees have a naturally recursive structure, and the problem itself is inherently self-similar—each subtree should also be identical. I rewrote the solution using recursion, checking the base cases first: if both nodes were null, they were equal; if one was null while the other wasn't, or if their values differed, they were not. Then, I recursively compared their left and right subtrees.

Compared to my iterative attempt, recursive method was more cleaner and functioned effectively. Refining the base cases helped me better comprehend the edge cases, which I had first missed, such as when both subtrees were empty.

After giving this procedure some thought, I saw that I was struggling because I was attempting to apply an iterative technique to an issue where recursion was a better fit. This experience improved my ability to choose the best strategy for various data structures and reaffirmed the significance of identifying when recursion simplifies an issue.