HW3 P1

a)

[100,150, 200, 70, 80, 90, 20, 30, 50]

I constructed the sequence by creating 3 increasing subsequences with size 3. Each subsequence is smaller than the one before it.

b)

image there are two different labels:

1. how many numbers you can pick in a row if you want it go up.
2. how many numbers you can pick in a row if you want it go down.

If every number only allowed groups of 3 numbers or fewer, then there are only 9 possible different label combinations (since each label can be 1, 2, or 3, giving 3×3 = 9 combinations). But we have 10 numbers, and each number must have a unique combination of labels. This is impossible if there are only 9 combinations available. So, at least one number must have a label of 4 (or more) for either increasing or decreasing sequences. That means somewhere in the list you can pick 4 numbers that go up, or 4 numbers that go down.

c)

Every increasing subsequence in S appears in S* because S* is sorted in increasing order. Any subsequence common to S and S* must be increasing, since it appears in the sorted S*. Thus, the longest common subsequence of S and S* is exactly the longest increasing subsequence of S. For the 9-integer sequence, the longest increasing subsequence has length 3, so the LCS is 3. In general, for any sequence of n distinct integers, the length of the longest increasing subsequence equals the LCS of S and S*.

d)

Algorithm: LIS(S)

```
Let T be an empty list
For each element x in S:
  Find the first position i in T where T[i] ≥ x (using binary search)
  If such a position i exists:
    Replace T[i] with x
  Else:
    Append x to T
End For
Return the length of T
```

I keep a list of the smallest possible last numbers for increasing subsequences of each length. For every number in S, I use binary search to find its correct spot in this list—if it's larger than all, I add it, otherwise I replace the existing number. This guarantees that the list's length equals the length of the longest increasing subsequence, and since each step is O(log n), the whole algorithm is O(n log n).

HW3 P2

a)

```
Algorithm FractionalKnapsack(items, capacity):

    // 1. Compute value-to-weight ratio for each item.
    For each item in items:
        item.ratio = item.value / item.weight

    // 2. Sort items in descending order by ratio.
    Sort items by item.ratio in descending order

    totalValue = 0
    remainingCapacity = capacity
    selectedItems = empty list

    // 3. Select items greedily.
    For each item in sorted items:
        If remainingCapacity is 0:
            Break out of loop
        If item.weight <= remainingCapacity:
            Add (item, fraction = 1) to selectedItems
            totalValue = totalValue + item.value
            remainingCapacity = remainingCapacity - item.weight
        Else:
            fraction = remainingCapacity / item.weight
            Add (item, fraction) to selectedItems
            totalValue = totalValue + item.value * fraction
            remainingCapacity = 0

    Return totalValue, selectedItems
```

b)

Time Complexity:

O(n log n):

The dominant step in the algorithm is sorting the items by their value-to-weight ratio, which takes O(n log n) time. Calculating the ratios and iterating through the sorted list each take O(n) time, so overall, the algorithm runs in O(n log n)

Space Complexity:

O(n):

We only need extra space to store the computed ratios and the list of selected items. Both of these require O(n) space at most, so the space complexity is O(n).

c)

Key difference:

Fractional knapsack:

You can take any fraction of an item, making the problem able to break into smaller pieces

0-1 knapsack:

you must either take the entire item or leave it. This makes the problem much harder to solve optimally with a greedy algorithms.

Counterexample for 0-1 Knapsack:

• **Item A:** Weight = 10, Value = 60, Ratio = 6

• **Item B:** Weight = 20, Value = 100, Ratio = 5

• **Item C:** Weight = 30, Value = 120, Ratio = 4

A greedy algorithm that picks based on the highest value-to-weight ratio will choose Item A first (ratio 6) and then Item B (ratio 5). This selection gives a total weight of 10 + 20 = 30 and a total value of 60 + 100 = 160. However, the optimal solution is to pick Item B and Item C, which exactly fill the knapsack (20 + 30 = 50) and yield a total value of 100 + 120 = 220. Thus, the greedy approach fails for the 0-1 knapsack.

why greedy is optimal for fractional knapsack:

For the fractional knapsack, you can take parts of items. This means you can always fill your knapsack with the best “value per weight” available. The greedy algorithm picks the item with the highest ratio first, and if it doesn’t fully fit, you just take the part that does. Since you’re always using your capacity in the best possible way, the solution is optimal.

HW3 P3:

a)

The time it takes for a sleeping cat at position (x, y) to wake up is equal to the shortest distance (in terms of moves) from the nearest awake cat in the matrix.

```
def WakeUpTime(matrix, m, n, x, y):
    queue = empty queue
    time = 0
    directions = [(0,1), (0,-1), (1,0), (-1,0)]

    // Add all awake cats to the queue
    For i in range(0, m):
        For j in range(0, n):
            If matrix[i][j] == 1:
                queue.push((i, j, 0))  // (row, col, time)

    // BFS to wake up sleeping cats
    While queue is not empty:
        (r, c, t) = queue.pop()

        // If we reach (x, y), return the time taken
        If (r, c) == (x, y):
            Return t

        // Explore all four directions
        For each (dr, dc) in directions:
            nr = r + dr
            nc = c + dc
            If (nr, nc) is within bounds and matrix[nr][nc] == -1:
                matrix[nr][nc] = 1  // Wake up the cat
                queue.push((nr, nc, t + 1))


    Return -1
```

b)

The total time for all sleeping cats to wake up is the longest shortest distance from any initially awake cat to a sleeping cat.

```
def WakeUpAllCats(matrix, m, n):
    queue = empty queue
    maxTime = 0
    directions = [(0,1), (0,-1), (1,0), (-1,0)]


    For i in range(0, m):
        For j in range(0, n):
            If matrix[i][j] == 1:
                queue.push((i, j, 0))

    // BFS to wake up all cats
    While queue is not empty:
        (r, c, t) = queue.pop()
        maxTime = max(maxTime, t)


        For each (dr, dc) in directions:
            nr = r + dr
            nc = c + dc
            If (nr, nc) is within bounds and matrix[nr][nc] == -1:
                matrix[nr][nc] = 1
                queue.push((nr, nc, t + 1))

    // Check if any sleeping cats remain
    For i in range(0, m):
        For j in range(0, n):
            If matrix[i][j] == -1:
                Return -1  // Not all cats can wake up

    Return maxTime
```

The maximum time is the number of levels in BFS, which is at most **O(m + n)** (worst case when the matrix is stretched in one direction).

c)

We slightly alter the multi-source BFS algorithm to follow the wake-up journey of a particular sleeping cat. For every cat that is awakened, we additionally keep the parent (previous) cell in addition to the minutes. By going backwards from the target sleeping cat to the closest awake cat, we can reconstruct the route.

```
def WakeUpPath(matrix, m, n, x, y):
    queue = empty queue
    parent = empty dictionary  // To track where each cat was awakened from
    directions = [(0,1), (0,-1), (1,0), (-1,0)]

    // Add all awake cats to the queue and mark them
    For i in range(0, m):
        For j in range(0, n):
            If matrix[i][j] == 1:
                queue.push((i, j))
                parent[(i, j)] = None  // No parent for initially awake cats

    /
    While queue is not empty:
        (r, c) = queue.pop()


        If (r, c) == (x, y):
            path = []
            while (r, c) is not None:
                path.append((r, c))
                (r, c) = parent[(r, c)]
            Return reversed(path)

        // Explore all four directions
        For each (dr, dc) in directions:
            nr = r + dr
            nc = c + dc
            If (nr, nc) is within bounds and matrix[nr][nc] == -1:
                matrix[nr][nc] = 1
                queue.push((nr, nc))
                parent[(nr, nc)] = (r, c)  // Track who woke up this cat

    Return []  // If (x, y) was never reached
```

1. Start with all awake cats and store their positions in a queue.

2. Use BFS to wake up sleeping cats, recording which cat woke up each one.

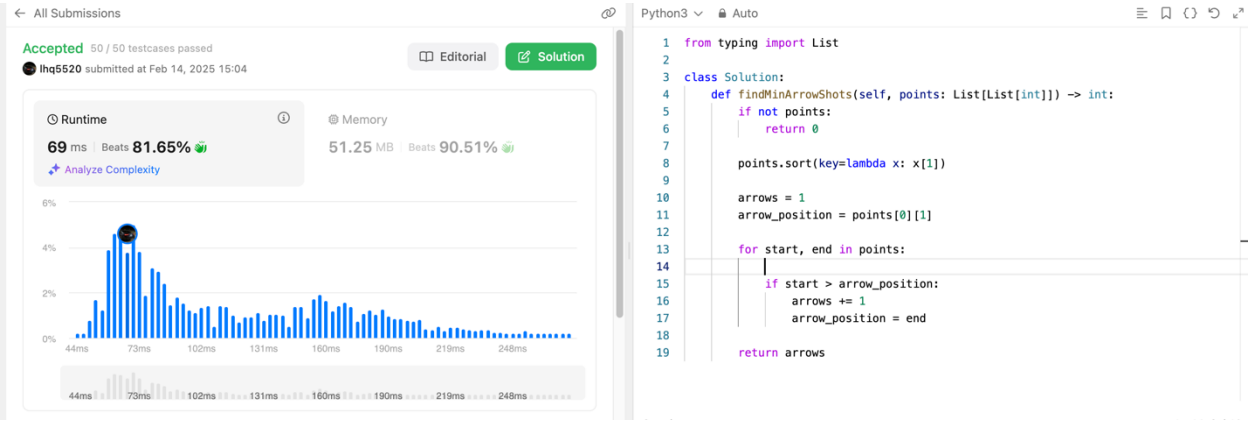
3. When the target cat at (x, y) wakes up, trace back from it to the first awake cat using the recorded path.

4. Return the path showing how it got woken up.

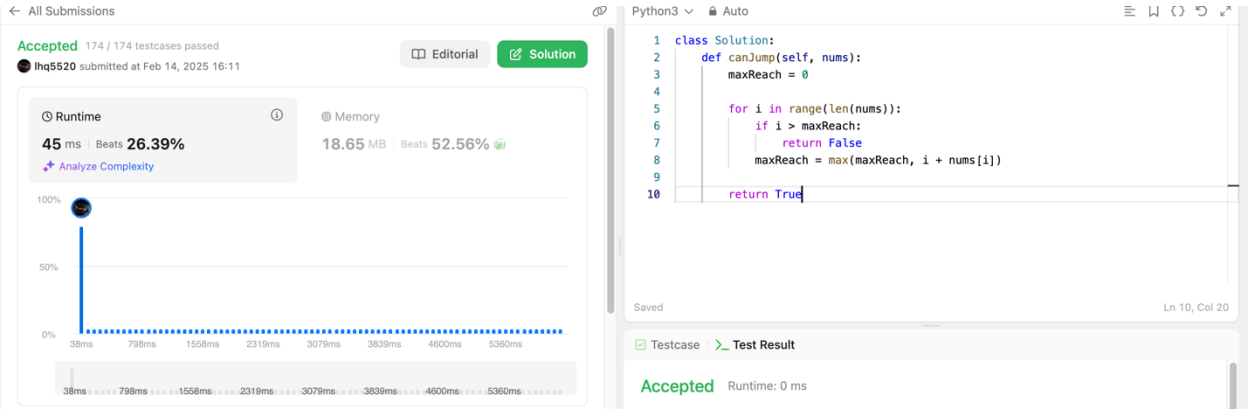
HW3 P4

a)

452. Minimum Number of Arrows to Burst Balloons - Medium



55. Jump Game – Medium



b)

At first, I tried brute force, checking every possible jump, but it was too slow and didn't work for large inputs. Then, I tried dynamic programming to store results and avoid repeating work, but it was still too slow. What finally worked was a greedy approach. Instead of tracking every jump, I just kept a variable to store the farthest index I could reach. If I ever got stuck before reaching the last index, I returned False. This was much simpler and ran in O(n) time. From this, I learned that sometimes greedy is better than DP and that tracking the farthest reach is a good way to solve problems like this.