Train Sorting

Necessary Components

- Subproblem
  - LIS[i] + LDS[i] - 1
  - So it comes down to constructing LIS and LDS
- Base Case
  - LIS and LDS when i = j is 1
- Recursive Steps

$$\text{LIS}[i] = \begin{cases} 1, & \text{if } i = n \\ \max\left(\text{LIS}[i],\ 1 + \text{LIS}[j]\right), & \text{for all } j > i \text{ such that } w[j] > w[i] \\ 1, & \text{if no such } j \text{ exists} \end{cases}$$

$$\text{LDS}[i] = \begin{cases} 1, & \text{if } i = n \\ \max\left(\text{LDS}[i],\ 1 + \text{LDS}[j]\right), & \text{for all } j > i \text{ such that } w[j] < w[i] \\ 1, & \text{if no such } j \text{ exists} \end{cases}$$

- Final Answer
  - max(LIS[i] + LDS[i] - 1) for all i in trains)
- Order of Construction
  - Bottom up
- Time Complexity

    - O(n^2)
- Space Complexity
    - O(2n)
        - LIS and LDS

Problem

- Idea: for any pivot p, max possible ordered train is LIS[p] + LDS[p] - 1
    - we consider each train as a pivot (the first car to fix)
- Subproblem: LCS and LDS starting at index i
    - if we know i, we can reason i - 1

The core idea is to reframe the problem. Instead of thinking about adding cars to the front or back, we can think about the final, sorted train. Any car in that train can be considered the **pivot**—the first car that was chosen for that specific train.

- Every car added *after* the pivot that is **heavier** must have gone to the **front**.
- Every car added *after* the pivot that is **lighter** must have gone to the **back**.

This means that for any chosen pivot car, the final train is formed by two sequences of cars that arrived *after* it:

1. An **increasing** sequence of weights (to be placed at the front).
2. A **decreasing** sequence of weights (to be placed at the back).

The algorithm finds the longest possible combination of these two sequences for every possible pivot car.

The Algorithm Explained

The code uses two arrays, `max_increasing` and `max_decreasing`, to store the lengths of these potential sequences.

- `max_increasing[i]`: Stores the length of the **Longest Increasing Subsequence (LIS)** of car weights that *starts* with the car at index `i`. This corresponds to the pivot car `i` plus all the heavier cars that can be added to the front.
- `max_decreasing[i]`: Stores the length of the **Longest Decreasing Subsequence (LDS)** of car weights that *starts* with the car at index `i`. This corresponds to the pivot car `i` plus all the lighter cars that can be added to the back.

Here's a step-by-step breakdown of the code:

1. Initialization

Python

```
max_increasing = [1] * num_cars
max_decreasing = [1] * num_cars
```

Both arrays are initialized with `1`s. This is because any single car, by itself, is a valid increasing (and decreasing) subsequence of length one.

---

2. Dynamic Programming Calculation

```
for current_index in range(num_cars - 1, -1, -1):
    for next_index in range(current_index + 1, num_cars):
        if train_weights[current_index] < train_weights[next_index]:
            max_increasing[current_index] = max(
                max_increasing[current_index],
                max_increasing[next_index] + 1
            )
        else:
            max_decreasing[current_index] = max(
                max_decreasing[current_index],
                max_decreasing[next_index] + 1
            )
```

This is the heart of the algorithm. It calculates the values for `max_increasing` and `max_decreasing`.

- **Backwards Loop**: The outer loop runs from the end of the `train_weights` list to the beginning. This is crucial for dynamic programming because to calculate the LIS/LDS for a car `i`, we need to have already calculated the results for all cars `j` that arrive after it (`j > i`).
- **Finding Subsequent Cars**: The inner loop looks at every car (`next_index`) that arrives *after* the `current_index`.
- **Building the Subsequences**:
  - If a future car (`next_index`) is **heavier**, it can extend the *increasing* subsequence starting from `current_index`. The new potential length is `1` (for the current car) `+` `max_increasing[next_index]` (for the rest of the chain).
  - If a future car is **lighter**, it can extend the *decreasing* subsequence. The logic is the same.
  - The code takes the `max` to ensure we find the *longest* possible chain.

---

3. Combining the Results

```
longest_train = 0
for pivot_index in range(num_cars):
    train_length = max_increasing[pivot_index] + max_decreasing[pivot_index]
- 1
    longest_train = max(longest_train, train_length)
print(longest_train)
```

After filling the arrays, the code iterates through every car, treating each one as a potential pivot.

- For a given `pivot_index`, the longest train it can anchor is the length of its increasing sequence (`max_increasing[pivot_index]`) plus the length of its decreasing sequence (`max_decreasing[pivot_index]`).
- **Why subtract 1?** The pivot car itself is counted in *both* `max_increasing` and `max_decreasing`. We subtract one to avoid double-counting it.
- The code keeps track of the maximum length found across all possible pivots, which gives the final answer.

Formal Recurrence Equation

Train = for any pivot p, LIS starting at p + LDS starting at p - 1

Longest Increasing Subsequence

$$\text{LIS}[i] = \begin{cases} 1, & \text{if } i = n \\ \max\left(\text{LIS}[i],\ 1 + \text{LIS}[j]\right), & \text{for all } j > i \text{ such that } w[j] > w[i] \\ 1, & \text{if no such } j \text{ exists} \end{cases}$$

Longest Decreasing Subsequence

$$\text{LDS}[i] = \begin{cases} 1, & \text{if } i = n \\ \max\left(\text{LDS}[i],\ 1 + \text{LDS}[j]\right), & \text{for all } j > i \text{ such that } w[j] < w[i] \\ 1, & \text{if no such } j \text{ exists} \end{cases}$$

- When adding a new element to an already existing LIS/LDS, you only need 1 comparison
  - whether the first element of the LIS/LDS is greater (or smaller) than the element we're considering
  - if smaller, we can add, total length is += 1
  - if greater, total length doesn't change.

Example

1. Input

```
Weights = [5, 2, 6, 4, 8, 3, 7]
n = 7
```

We'll compute two DP arrays:

Index	Weight	LIS[i]	LDS[i]
0	5	?	?
1	2	?	?
2	6	?	?
3	4	?	?
4	8	?	?
5	3	?	?
6	7	?	?

Initialization: `LIS[i] = LDS[i] = 1`.

---

2. Backward DP filling

We go from right → left so later cars (arriving after) are known.

i = 6 (7)

No cars after it → LIS[6] = LDS[6] = 1.

i = 5 (3)

- next = 6 (7): 3 < 7 → `LIS[5] = max(1, 1+1)=2`
- no smaller car after → `LDS[5]=1`

→ LIS[5]=2, LDS[5]=1

i = 4 (8)

- next = 5 (3): 8 > 3 → `LDS[4] = max(1, 1+1)=2`
- next = 6 (7): 8 > 7 → `LDS[4] = max(2, 1+1)=2`
- no heavier after → `LIS[4]=1`

→ LIS[4]=1, LDS[4]=2

i = 3 (4)

- next = 4 (8): 4<8 → LIS[3]=2
- next = 5 (3): 4>3 → LDS[3]=2
- next = 6 (7): 4<7 → LIS[3]=max(2,1+1)=2

  → LIS[3]=2, LDS[3]=2

i = 2 (6)

- next = 3 (4): 6>4 → LDS[2]=2
- next = 4 (8): 6<8 → LIS[2]=2
- next = 5 (3): 6>3 → LDS[2]=max(2,1+1)=2
- next = 6 (7): 6<7 → LIS[2]=max(2,1+1)=2

  → LIS[2]=2, LDS[2]=2

i = 1 (2)

- next = 2 (6): 2<6 → LIS[1]=2
- next = 3 (4): 2<4 → LIS[1]=2
- next = 4 (8): 2<8 → LIS[1]=2
- next = 5 (3): 2<3 → LIS[1]=2
- next = 6 (7): 2<7 → LIS[1]=2

  No lighter car after → LDS[1]=1

  → LIS[1]=2, LDS[1]=1

i = 0 (5)

- next = 1 (2): 5>2 → LDS[0]=2
- next = 2 (6): 5<6 → LIS[0]=2
- next = 3 (4): 5>4 → LDS[0]=max(2,1+2)=3

- next = 4 (8): 5<8 → LIS[0]=max(2,1+1)=2
- next = 5 (3): 5>3 → LDS[0]=max(3,1+1)=3
- next = 6 (7): 5<7 → LIS[0]=max(2,1+1)=2

  → LIS[0]=2, LDS[0]=3

---

3. Combine for each pivot

Train length = `LIS[i] + LDS[i] - 1`

i	Weight	LIS	LDS	LIS+LDS−1
0	5	2	3	4
1	2	2	1	2
2	6	2	2	3
3	4	2	2	3
4	8	1	2	2
5	3	2	1	2
6	7	1	1	1

---

4. Result

Maximum = **4**, achieved at pivot **weight 5**.

That train corresponds roughly to **[8, 6, 5, 4]** (heaviest to lightest order).