To create the C++ program for the "Sculpture" problem from Day1\_Sculpture.pdf, we need to find the maximum number of stones (M) that can be used to build a sculpture, and identify all possible positions where a single support device can be placed to achieve this maximum M.

**Problem Analysis**

The problem involves stacking stones with weight (w\_i) and load capacity (L\_i). The rules are:

1. Stones are placed from top to bottom in increasing order of their original indices (e.g.,

s\_1 on s\_2, where s\_1 < s\_2).

1. The total weight of stones above any stone must not exceed its load capacity.
2. A single support device can be placed on one stone (not the bottommost). If placed on stone

j, the stone directly below j (stone j+1) does not bear the weight of stones above j.

This problem can be broken down into:

* Finding the maximum length of a valid stack of stones.
* Considering a single support device that effectively splits the stack into two independent parts: one above the supported stone and one below.

**Dynamic Programming Approach (O(N log N))**

Since N can be up to 20,000, an O(N^2) solution would be too slow. We need an O(N log N) approach, which typically involves dynamic programming optimized with a segment tree.

We'll use two DP arrays and a segment tree for optimization:

1. **Left[i]**: Stores (max\_length, min\_total\_weight\_of\_stack) for a valid sculpture where stone i is the *bottommost* stone, using only stones with indices less than or equal to i.
2. **Right[i]**: Stores (max\_length, min\_total\_weight\_of\_stack) for a valid sculpture where stone i is the *topmost* stone, using only stones with indices greater than or equal to i.

**Segment Tree:** The segment tree will store (max\_length, min\_total\_weight\_for\_that\_length) pairs. It will be indexed by *compressed coordinate values* of the relevant weights (w\_i, L\_i, and accumulated stack weights). Since all w\_i and L\_i are up to 5,000,000, and intermediate accumulated weights can exceed this, we will only consider accumulated weights up to the maximum L\_i for updates (as any stack heavier than that cannot be placed on any other stone).

The segment tree's update operation will take a compressed weight index and a (length, total\_weight) pair. It will update the node by maximizing length, and if lengths are equal, minimizing total weight. The query operation will take a range of compressed weights and return the (max\_length, min\_total\_weight) pair within that range.

**Algorithm Steps:**

1. **Coordinate Compression:** Collect all unique w\_i and L\_i values into a sorted list (all\_coords). Create a map (coord\_map) from actual values to their compressed indices. This maps values up to 5,000,000 to indices 0 to K-1, where K is the number of unique values.
2. **Compute Left DP:**
   * Initialize a segment tree (all nodes represent 0 length and 0 total weight for an empty stack).
   * Iterate i from 1 to N:
     + Query the segment tree for the best stack (max\_len, min\_sum) that can be placed *on top of* stone i. This means querying the range [0, coord\_map[stones[i].L]] (all stacks whose total weight is less than or equal to stones[i].L).
     + Left[i].first = max\_len + 1 (add stone i to the best stack found).
     + Left[i].second = min\_sum + stones[i].w (add stones[i].w to the total weight).
     + Update the segment tree at coord\_map[Left[i].second] with (Left[i].first, Left[i].second). **Important:** Only update if Left[i].second is within the maximum L\_i range (i.e., Left[i].second <= all\_coords.back()), because heavier stacks cannot be placed on any other stone.
3. **Compute Right DP:**
   * Re-initialize a new segment tree for this pass.
   * Iterate i from N down to 1:
     + Query the segment tree for the best stack (max\_len, min\_sum) that can be placed *below* stone i. This means querying the range [0, coord\_map[stones[i].L]] (all stacks whose total weight is less than or equal to stones[i].L).
     + Right[i].first = max\_len + 1.
     + Right[i].second = min\_sum + stones[i].w.
     + Update the segment tree at coord\_map[Right[i].second] with (Right[i].first, Right[i].second). Only update if Right[i].second is within the maximum L\_i range.
4. **Find Maximum M and Support Positions:**
   * Initialize max\_M = 0.
   * First, calculate max\_M considering single stacks (no support used): max\_M = max(max\_M, Left[i].first) for all i.
   * Next, calculate max\_M considering two segments (with support at stone j):
     + Iterate j from 1 to N-1.
     + If Left[j].first and Right[j+1].first are both valid (greater than 0), calculate current\_M = Left[j].first + Right[j+1].first.
     + Update max\_M = max(max\_M, current\_M).
   * Finally, collect all valid j positions for the support device:
     + Iterate j from 1 to N-1.
     + If Left[j].first + Right[j+1].first == max\_M (and both parts are valid stacks), add j to a std::set to ensure uniqueness and automatic sorting.
5. **Output:** Print max\_M on the first line. On the second line, print the size of the set of support positions, followed by the sorted positions themselves. If the set is empty, print 0.