Milestone 2 – Dynamic Programming Algorithms

1. Team Members

Name: Triet Nguyen

Role: Worked alone on all tasks: algorithm design, proofs, coding, and experiments.

2.1 Algorithm3 – Naive Exhaustive Search (Θ(n·2^{n−1}))

Problem

Solve the general version of ProblemG for any order of sculptures.

Design

Algorithm3 tries every possible way to place cuts between sculptures. Between sculpture i and i+1, either:

1. keep them on the same platform, or
2. place a cut and start a new platform.

This gives 2^(n−1) different arrangements.

The algorithm explores these arrangements with a recursive search:

It keeps the current platform's total width and tallest height, and a list of how many sculptures each platform contains.

At each sculpture index i it tries:

Option 1: add sculpture i to the current platform if the width still fits within W;

Option 2: finish the current platform and start a new platform containing only sculpture i.

When all sculptures are placed, it closes the last platform, computes the total cost (sum of platform heights), and updates the best answer if this total is smaller.

The program records the grouping that gives the smallest total height and returns it.

Running time

There are 2^(n−1) different patterns of cuts. For each pattern, scan up to n sculptures to check widths and compute heights. Therefore the running time is Θ(n·2^{n−1}). Space usage is Θ(n) to hold the current grouping and the best grouping.

Correctness

Any valid solution to ProblemG is exactly one choice of cuts in this fixed order of sculptures. Algorithm3 explores all such choices and, for each one, computes the total height using the same rule as the problem (each platform's height is the maximum height of its sculptures, and the total is the sum). Because it keeps the arrangement with the smallest total height among all possibilities, this total must be equal to the optimal total height.

2.2 Algorithm4 – Θ(n^3) Dynamic Programming

Idea

Algorithm4 uses dynamic programming on prefixes and the number of platforms.

First, it precomputes:

cost[i][j] = the height of a single platform containing sculptures i..j, or ∞ if the total width of i..j is greater than W.

This is done by fixing a start index i, then extending j from i to the right, keeping a running width and maximum height. If the width goes over W, we stop.

State definition

dp[k][j] = minimum total height to place sculptures 0..j using exactly k platforms.

Recurrence

Base case: with one platform,

$$dp[1][j] = cost[0][j]$$

For k ≥ 2, the last platform must cover sculptures i+1..j for some i < j:

$$dp[k][j] = \min_{i<j} \left(dp[k-1][i] + cost[i+1][j] \right)$$

The final answer is:

$$\min_{1 \le k \le n} dp[k][n-1].$$

While filling the table, also remember which i gave the minimum so it can later reconstruct the group sizes.

Running time

Precomputing all cost[i][j] uses $\Theta(n^2)$ time.

The DP table has $\Theta(n^2)$ states (k from 1..n and j from 0..n−1).

For each state (k, j) try up to O(n) possible split positions i.

So total time is $\Theta(n^3)$.

Space usage is $\Theta(n^2)$ for cost and dp.

Correctness

Take any optimal arrangement of sculptures 0..j using exactly k platforms.

Let the last platform in this arrangement cover sculptures i+1..j.

- The first k−1 platforms cover sculptures 0..i.
- If these first k−1 platforms were not arranged optimally, it could replace them by a better arrangement with cost dp[k−1][i] and get a lower total cost for 0..j, which contradicts optimality.

Thus, for some i, the optimal cost for 0..j with k platforms must equal:

$$dp[k-1][i] + cost[i+1][j].$$

Taking the minimum over all valid i gives the correct value for dp[k][j].

By induction on j and k, all entries in dp are correct.

Taking min_k dp[k][n−1] yields the optimal total height for the full sequence.

2.3 Algorithm5 – Θ(n²) Dynamic Programming

Algorithm5 uses a simpler DP that tracks only the starting index of the remaining sculptures.

State definition

Let:

dp[i] = minimum total height to place sculptures i..n−1.

For state i, the first platform can end at any j ≥ i as long as the total width from i to j is ≤ W.

The cost of choosing j is:

height of the first platform: max(heights[i..j]), plus

optimal cost of the rest: dp[j+1].

So the recurrence is:

$$dp[i] = \min_{\substack{j \geq i \\ \sum_{k=i}^{j} widths[k] \leq W}} \left(\max_{k=i}^{j} heights[k] + dp[j+1] \right), \quad dp[n] = 0.$$

The answer is dp[0].

also store which j gave the minimum for each i so it can reconstruct the grouping.

Program5A implements this recurrence with a top-down recursive function dp(i) plus memoization.

Program5B implements a bottom-up version that fills dp[i] from i = n−1 down to 0.

Running time

There are n+1 states dp[i] for i = 0..n.

For each i, extend j forward until the width is too large.

In the worst case this inner loop does O(n) work, so total time is $\Theta(n^2)$.

Space usage is Θ(n) for dp and the choice array.

Correctness

Fix any index i and consider an optimal arrangement of sculptures i..n−1.

Let its first platform cover i..j.

The cost of this first platform is max(heights[i..j]).

The remaining sculptures j+1..n−1 must be arranged optimally; otherwise it could replace that suffix by an arrangement with cost dp[j+1] and obtain a smaller total height, which contradicts optimality.

Therefore, the optimal cost for i..n−1 is exactly:

$$\min_{j} \left(\max(heights[i..j]) + dp[j+1] \right),$$

which is how dp[i] is defined.

By backward induction from i = n down to 0, every dp[i] equals the true minimal cost, and dp[0] is the optimal total height.

Since Programs5A and 5B both implement this recurrence, they return the same optimal value.

3. Experimental Comparative Study

To compare the running times, I generated random test cases for several input sizes n. For each n I created multiple instances with random heights and widths and measured the running time of:

Program3 (Algorithm3, exponential),

Program4 (Algorithm4, $\Theta(n^3)$),

Program5A (Algorithm5, top-down $\Theta(n^2)$),

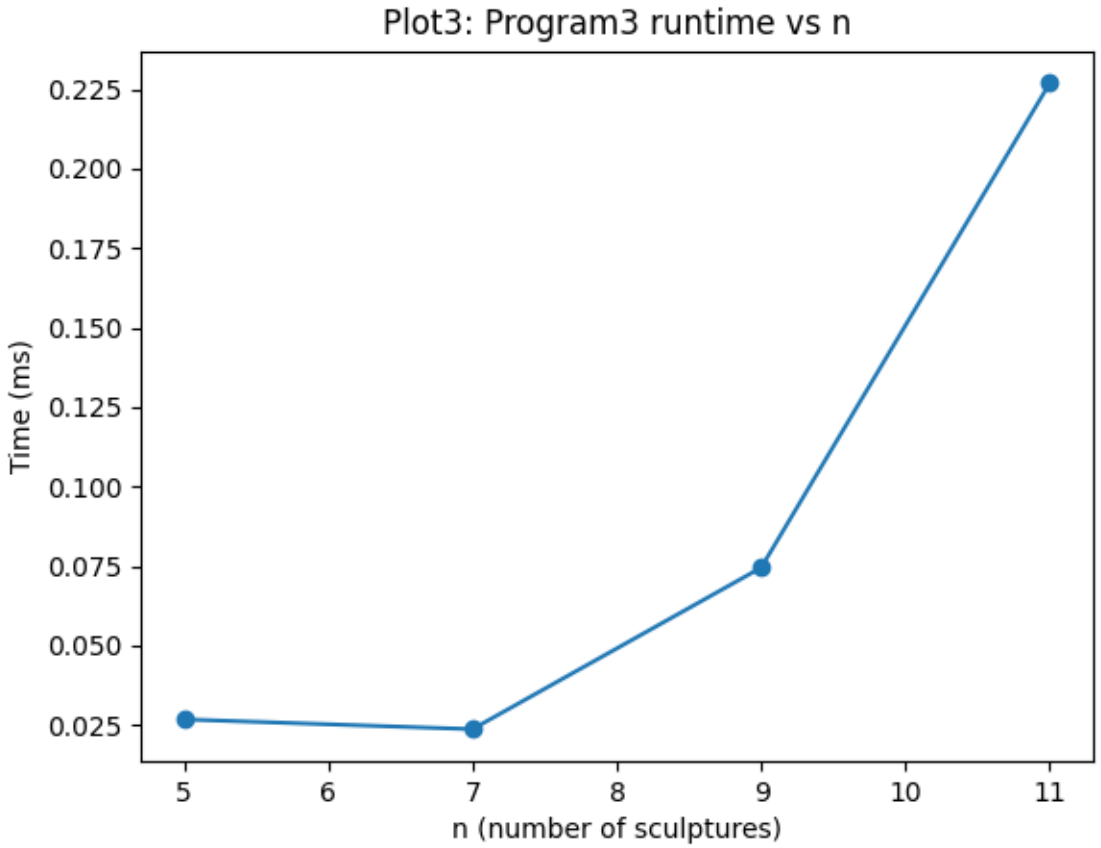
Program5B (Algorithm5, bottom-up $\Theta(n^2)$).

For each comparison I plotted:

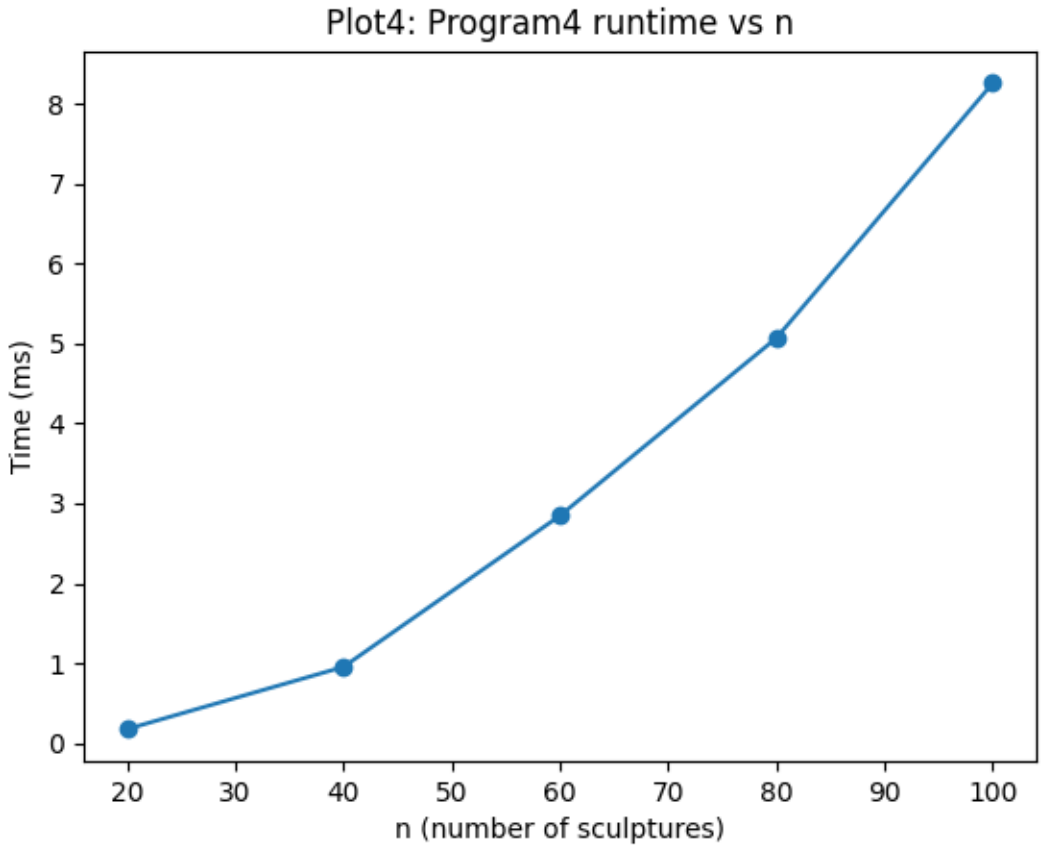
x-axis: input size n,

y-axis: average running time in milliseconds.

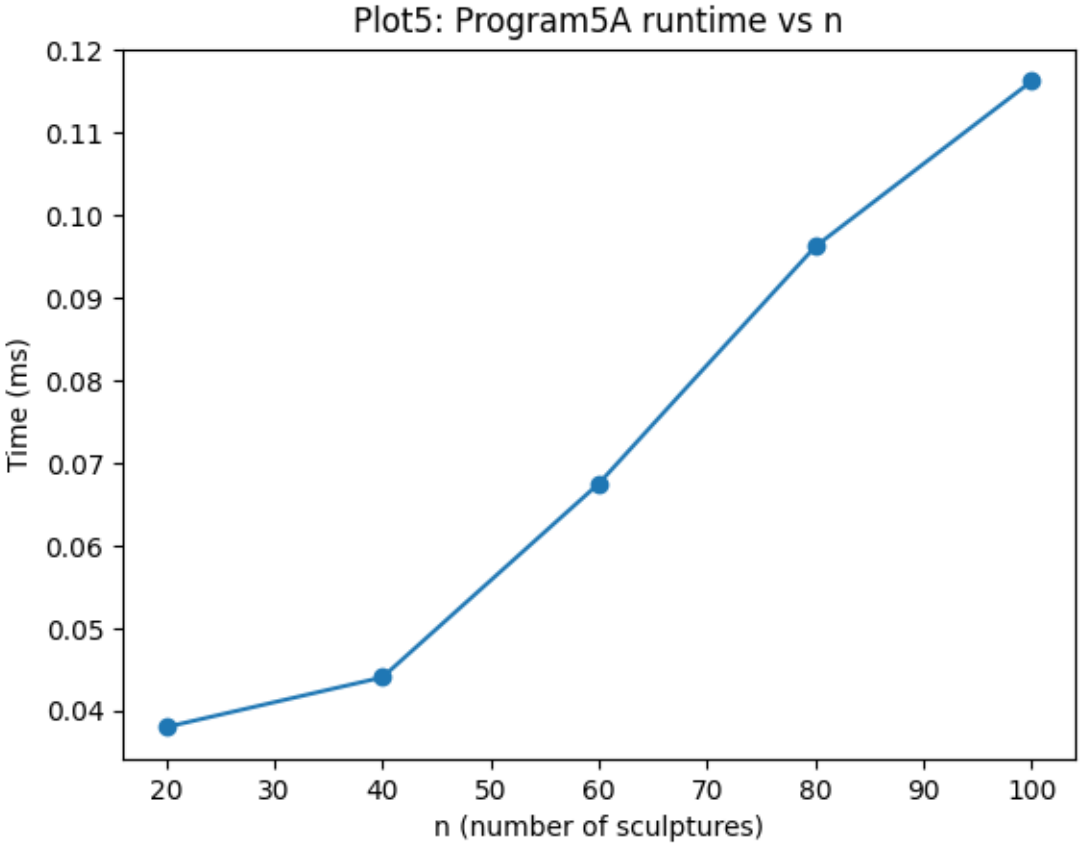
Plot3: Program3 runtime vs n. The curve grows very quickly and becomes impractical even for moderate n, which matches the $\Theta(n \cdot 2^{n-1})$ time bound.

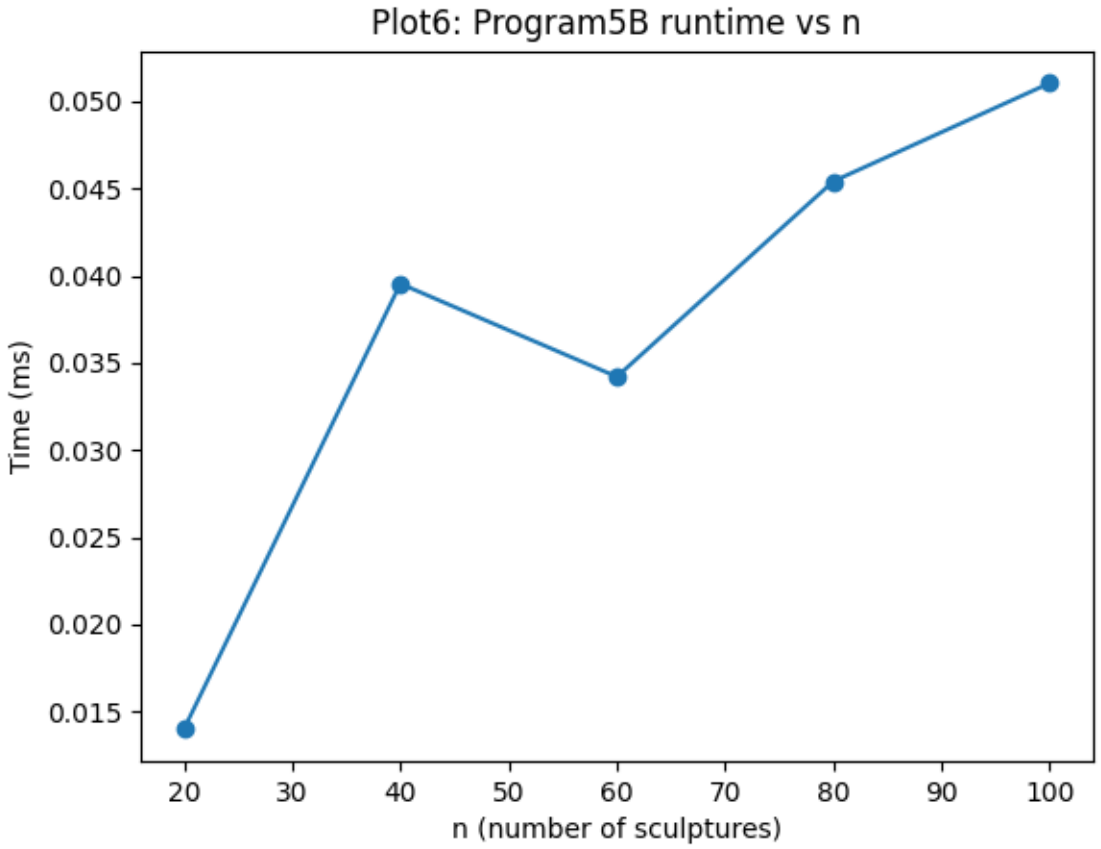


Plot4: Program4 runtime vs n. This curve grows smoother than Plot3 but still faster than quadratic, as expected for $\Theta(n^3)$.



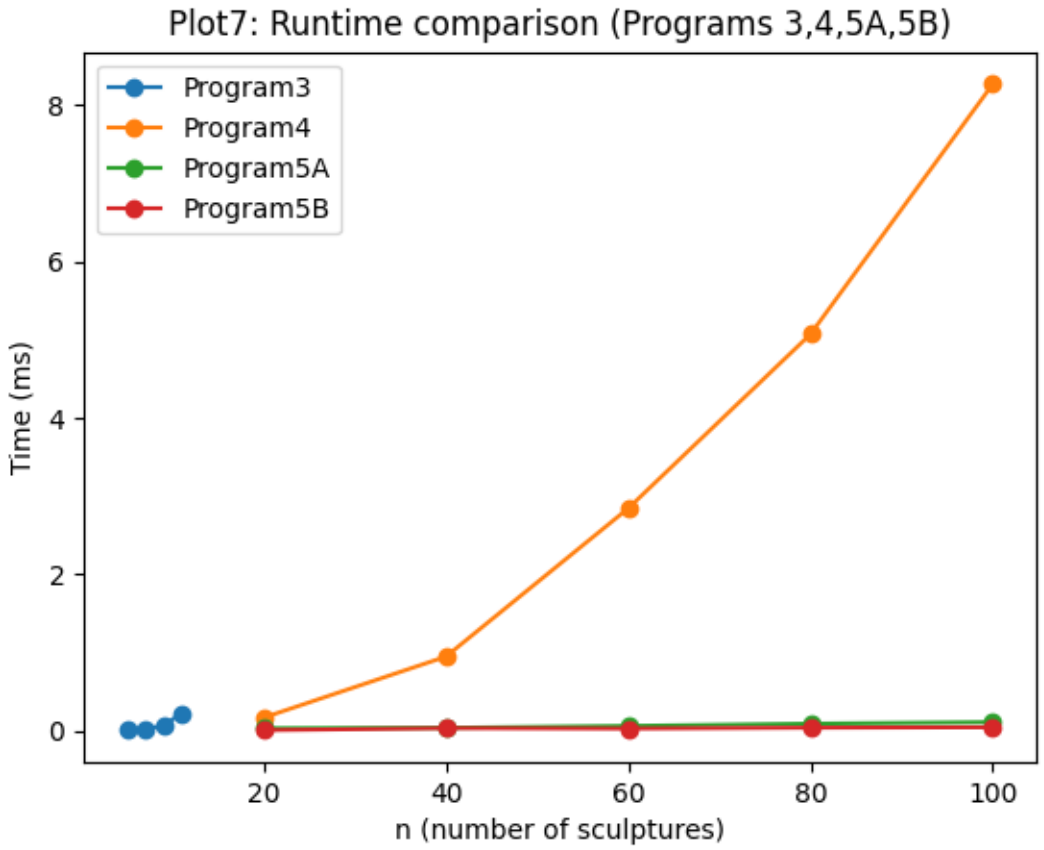
Plots5 and 6: Program5A and Program5B runtime vs n. Both curves look roughly quadratic and are far lower than Plots3 and 4 for larger n. Program5B is slightly faster because it avoids recursion overhead.


Plot5: Program5A runtime vs n
Time (ms)
0.12
0.11
0.10
0.09
0.08
0.07
0.06
0.05
0.04
20
30
40
50
60
70
80
90
100
n (number of sculptures)



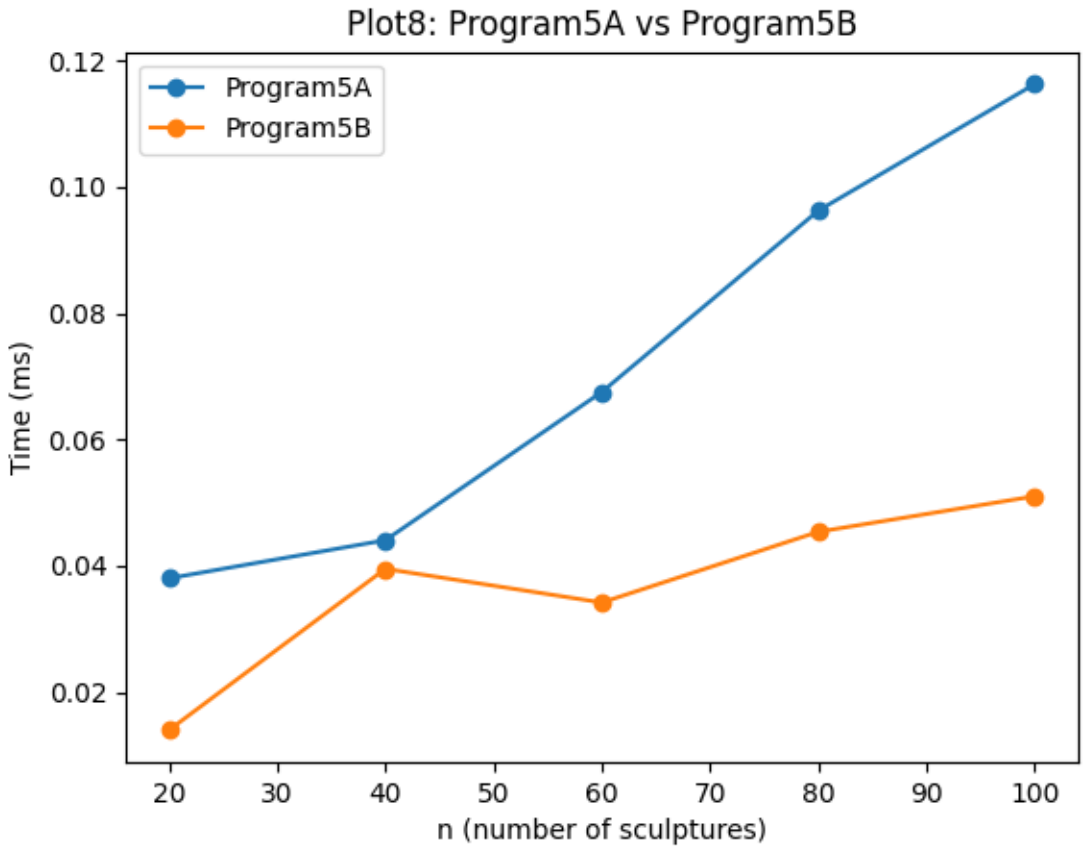
Plot7 (overlay): Plots3–6 together. Program3 dominates the graph for larger n, Program4 lies below it but above the two Algorithm5 curves, and Programs5A and 5B are the most efficient for

larger inputs.

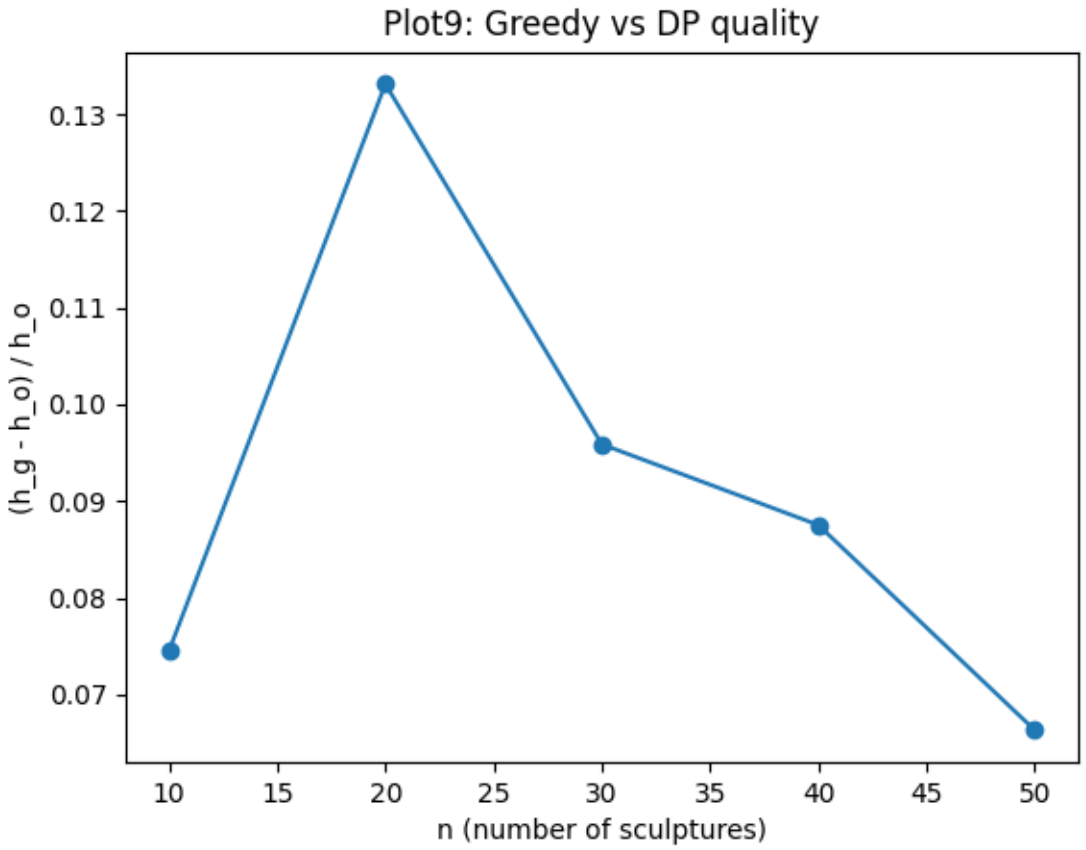


Plot8 (overlay): Plots5 and 6 only. This makes it easy to see that the two implementations of Algorithm5 have nearly the same running time. Bottom-up is slightly faster but the difference is

small.


Plot8: Program5A vs Program5B
Program5A
Program5B
Time (ms)
0.12
0.10
0.08
0.06
0.04
0.02
20
30
40
50
60
70
80
90
100
n (number of sculptures)

Quality of greedy solution (Plot9).



I also tested how far the greedy Algorithm1 (Program1 from Milestone 1) can be from optimal on random inputs.

For each instance I computed:

h_o: the optimal total height from one of the DP programs (3, 4, 5A, or 5B),

h_g: the total height from the greedy Algorithm1.

Then I plotted the average of (h_g − h_o) / h_o versus n.

The plot showed that:

On many inputs, the greedy solution matches the optimal one or is very close.

On some inputs, (h_g − h_o) / h_o is noticeably positive, meaning the greedy method is clearly worse.

This confirms that greedy Algorithm1 is not guaranteed to be optimal for ProblemG and that the dynamic programming solutions are needed for full correctness.

4. Conclusion

In this milestone I designed and implemented several algorithms for the general platform problem and compared them both theoretically and experimentally.

Algorithm3 is a very simple exhaustive search that checks all possible placements of cuts and always finds the optimal answer, but its running time grows like $\Theta(n \cdot 2^{n-1})$ and quickly becomes unusable.

Algorithm4 applies dynamic programming on two dimensions (number of platforms and prefix length) and runs in $\Theta(n^3)$ time, which is much better than exponential.

Algorithm5 uses a more compact $\Theta(n^2)$ dynamic program based on optimal substructure on suffixes; both its top-down and bottom-up implementations (Programs5A and 5B) run efficiently even for larger input sizes.

The experiments confirmed the theoretical running-time bounds and clearly showed the difference between exponential, cubic, and quadratic growth. They also showed that the greedy Algorithm1 can give non-optimal results on general data, while the dynamic programming algorithms always return the true optimal height.

This milestone helped me practice designing recursive formulations, turning them into top-down and bottom-up DP code, and using experiments to check algorithm behavior in practice.