Algorithm Abstraction & Design

Programming Project

Milestone 2

Jessica Lourenco, Ozlem Polat, Zhengxiao Wang

COP4533

University of Florida

November 2024

1. Team Member Contributions

The project was a collaborative effort where all team members constantly communicated and discussed every aspect of the work. Jess focused on implementing the dynamic programming algorithms and drafting the initial report, including descriptions for the algorithms. Zhengxiao and Ozlem worked together on writing and testing the algorithms, developing test cases, writing proofs, analyzing complexities, and filling out sections of the report. Ozlem and Zhengxiao also scripted the experimental study, generated the graphs, analyzed the results, and finalized the report. The entire team collaborated closely throughout, discussing experimental designs, algorithm outputs, inputs, and results, making it feel more like a group study session, which was both productive and enjoyable.

2. Algorithms - Design & Analysis

In milestone 2 of the project, we explore different ways we can design an algorithm that solves the general version of the given problem,

ProblemG:

> Given the heights $h_1, \ldots, h_n$ and the base widths $w_1, \ldots, w_n$ of *n* sculptures, along with the width *W* of the display platform, find an arrangement of the sculptures on platforms that minimizes the total height.

2.1. Algorithm 3

2.1.1 $\Theta(n2^{n-1})$ Time Naive Algorithm Design

The naive algorithm involves generating all possible ways to split the sculptures into different platforms and calculating the total cost for each arrangement. Since the sculptures must remain in a specific order, the problem can be interpreted as finding all feasible possible arrangements (i.e., those with platforms that respect the width constraint *W)* and selecting the one that minimizes the total height cost.

Our design uses recursion to generate all possible arrangements of sculptures (in order). For each sculpture, it considers adding it to the current platform (if it remains feasible with the width constraint *W*) or starting a new platform. If adding a sculpture causes the total width to exceed *W*, the arrangement is considered infeasible, and further exploration on that path is stopped.

It also keeps track of the *minimum* total height cost and the best arrangement found so far. Once all sculptures are placed, we check if the total height cost of the current arrangement is lower than the minimum total height cost we have found. If so, we update the variables tracking the minimum total height cost and the best arrangement.

<u>Time Complexity</u>

The time complexity of Algorithm3 is $\Theta(n2^{n-1})$, where n is the number of the given sculptures.
The main function in the program, responsible for performing the recursion, generates all possible ways to arrange the sculptures into platforms while maintaining their order. For each sculpture (except the first one), the algorithm decides whether to start a new platform or not to place that sculpture. This results in a total of n - 1 decisions. Furthermore, since we have two choices (continue on the current platform or start a new one), the total number of ways to arrange the sculptures is given by 2^{n-1}. Since each arrangement requires creating platforms that collectively cover *n* sculptures, the total amount of

work per platform takes Θ(n) time. Notice that to keep this time complexity, we calculate the total width and max height of a platform and the total cost of the arrangement as we process each sculpture – avoiding any extra nested loop operations and allowing the algorithm to complete them in Θ(1) time. Finally, as we get 2^{n-1} total arrangements, this results in a time complexity of $\Theta(n2^{n-1})$ for the recursion.

Outside the recursion, we have a loop for creating the array with the counts of sculptures per platform, which takes Θ(n) time to complete.

In conclusion, the dominant operation in program3 is performing the recursion, which takes $\Theta(n2^{n-1})$ time to complete.

Therefore, program3 has a time complexity of $\Theta(n2^{n-1})$.

2.1.2 Proof of Correctness - Algorithm3

A. Defining the Subproblems

Let OPT(i) represent the minimum total height cost of arranging the first i sculptures onto platforms, subject to the width constraint W, where $i \leq n$. Each subproblem finds the optimal arrangement for a subset of the sculptures from position 1 to i.

B. Formulating the Recurrence

The recurrence relation for OPT(i) is as follows:

Base Case: If i = 0, there are no sculptures, so OPT(0) = 0.
Recursive Case: For i > 0, we consider all possible previous points j (from 0 to i-1) where a platform could end. Each choice of j defines a potential platform from sculpture j+1 to i. For any valid choice, we calculate the maximum height on this platform and ensure that the total width does not exceed W. The recurrence relation is:

$$OPT(i) = \min \{0 \leq j < i\} \{OPT(j) + \max (heights [j+1, i])\}$$

where max(heights[j+1, i]) is the maximum height for sculptures from j+1 to i, provided their total width is within W.

C. Proof of the Recurrence Correctness

Assume that OPT(i) correctly represents the minimum total height cost for arranging sculptures up to index i within the width constraint.

To confirm this recurrence's correctness, we consider all possible configurations:

Case 1: If a platform ends at j, then OPT(i) must add the maximum height for sculptures j+1 to i, while also accounting for the optimal height arrangement up to j, which is OPT(j).

Case 2: The algorithm considers all feasible splits from j to i, ensuring that each configuration is

evaluated. By taking the minimum height cost among these configurations, OPT(i) makes sure the optimal solution is chosen.

By generating all possible arrangements, the evaluation of every combination is guaranteed, hence this naive approach guarantees correctness as it brute forces all feasible platform assignments, selecting the minimal cost arrangement.

D. Evaluation of the Recurrence Relation

The algorithm iteratively computes OPT values by generating all valid arrangements up to each index i and evaluating each subproblem solution and storing the result. Since each reference to OPT(j) only relies on previously computed values, the recurrence relation is followed accurately. By evaluating every feasible combination, the algorithm meets all dependencies and accurately applies the recurrence relation.

E. Correctness of the Algorithm

For n = 0, the base case returns (0, 0, []).
For n> 0, the algorithm exhaustively generates all possible arrangements of sculptures i into platforms, ensuring that the width constraint W is always respected. Since every feasible arrangement is evaluated, the minimum height cost among them is guaranteed to be the optimal solution. The final result represents the minimum cumulative height cost for arranging all sculptures, thus confirming correctness.

F. Pseudocode for Algorithm3

```
function program3(n, W, heights, widths):
        if n == 0:
        return (0, 0, [])
        initialize min_cost to infinity
        initialize best_arrangement to None
        recursive function generate_arrangements(current, start, current_cost):
        if start >= n:
        if current_cost < min_cost:
        min_cost = current_cost
        best_arrangement = copy of current
        else:
        platform_width_sum = 0
        platform_max_height = 0
        for i from start to n - 1:
        platform_width_sum += widths[i]
        if platform_width_sum <= W:
                platform_max_height = max(platform_max_height, heights[i])
                next_cost = current_cost + platform_max_height
                next_arrangement = copy of current with range(start, i + 1) added
            generate_arrangements(next_arrangement, i + 1, next_cost)
        call generate_arrangements([], 0, 0)
        sculptures_count = length of each platform in best_arrangement


        return (length of best_arrangement, min_cost, sculptures_count)
```

2.2. Algorithm 4

2.2.1 Θ(n^3) Time Dynamic Programing Algorithm Design

The key idea of the dynamic programming approach is to break *ProblemG* into smaller, manageable subproblems and use the solutions of those subproblems to build up the solution to the larger problem. Here, we're focusing on finding the minimum total height cost for arranging sculptures on platforms (and in order).

We cache the intermediate results of computations in three separate arrays:

- *min_cost*: each element *min_cost[i]* represents the minimum cumulative height cost of platforms required to arrange the sculptures up to index *i.*
- *platforms:* keeps track of the number of platforms used to arrange the sculptures up to each index *i.* In other words, each element *platforms[i]* represents the number of platforms used to reach *min_cost[i]*.
- *Sculptures_on_last_platforms:* stores the number of sculptures on the **last** platform used to reach *min_cost[i]*. We also used this array later on in the program to figure out the number of sculptures per platform used in the optimal solution.

The design decision to use three separate arrays instead of an *n* x *n* x *n* table was made to keep the code and array states easier to manage.

The program systematically builds up the solution by considering each sculpture individually. At each step, it explores different ways to group sculptures into platforms and calculates the resulting costs. For each sculpture *i*, it checks all potential starting points *j* of a platform that ends at *i*. Essentially, the algorithm explores all sub-arrays (groups of sculptures) from *j* to *i* to find the optimal placement. Note that instead of considering all the possible 2^{n-1} possible arrangements, we now find the optimal solution incrementally using these subarrays.

<u>Time Complexity</u>

The time complexity of Algorithm 4 is Θ(n^3), where n is the number of the given sculptures.

The algorithm primarily consists of three nested loops. The outermost loop iterates over each sculpture, represented by the index *i*, running for a total of *n* times. For each value of *i*, the algorithm attempts to find the best possible configuration of platforms ending at sculpture *i*. This loop completes in Θ(n) time.

Within the outer loop, there is a middle loop that iterates over all possible starting points *j* for a platform that ends at sculpture *i*. Eventually, this loop must cover all *n* sculptures (when *i* equals *n-1*), completing in Θ(n) time.

The final nested loop is simpler. It iterates over each sculpture between indices *j* and *i* to calculate the total width of the subarray and identifies the maximum height among its sculptures. This one will also eventually cover all *n* sculptures, completing in Θ(n) time.

The final time complexity for the nested loops is Θ(n^3). Before the program terminates, we have one simple final loop to determine the number of sculptures per platform in the optimal solution. We also

must reverse it before returning. Each of these operations can run up to *n* times but do not contribute to additional time for the overall program since the most costly operation is the nested loops.

Therefore, program4 has a time complexity of $\Theta(n^3)$.

2.2.2 Proof of Correctness

A. Defining the Subproblems

Let OPT(i) represent the minimum total height cost for arranging sculptures up to and including index *i* into platforms, subject to the width constraint W. This function considers each platform arrangement and calculates the cumulative height, accounting for all preceding platforms. Subproblems are defined by finding optimal configurations for subsets of sculptures from index 0 up to i, where i < n.

B. Formulating the Recurrence

The recurrence relation for OPT(i) is as follows:

Base Case: If i = 0, no sculptures exist, so OPT(0) = 0, and the function returns (0, 0, []) without accessing the min_cost array.

Recursive Case: For each i > 0, consider possible platform start points j (from 0 to i-1), where each j defines a potential platform from sculpture j+1 to i.

If the platform width from j+1 to i does not exceed W, the recurrence for OPT(i) is:

$$OPT(i) = \min_{0 \le j < i} \{ OPT(j) + \max(heights[j+1, i]) \}$$

where max(heights[j+1, i]) denotes the maximum height for sculptures from j+1 to i, making sure the cumulative width respects W.

C. Proof of the Recurrence Correctness

Assume that Algorithm 4 correctly computes the minimum total height cost up to each index i. This means for each position i, OPT(i) is computed based on an optimal placement of platforms up to i, given the constraints. To confirm the recurrence's correctness, we examine the configurations:

Case 1: If a platform ends at j, then OPT(i) must add the maximum height for sculptures spanning from j+1 to i, while also accumulating the optimal height cost for preceding sculptures up to j (OPT(j)).

Case 2: For each feasible arrangement ending at i, all valid configurations are evaluated through j, ensuring OPT(i) minimizes the total height cost by selecting the optimal configuration across all possible placements.

Thus, the recurrence is correct because it evaluates each valid configuration by placing platforms at every potential starting point j. The recurrence relation includes all feasible combinations for grouping sculptures from j+1 to i, ensuring the minimum height is recorded by choosing the optimal platform height at each step. The solution satisfies the minimal height requirement while respecting platform constraints.

D. Evaluation of the Recurrence Relation

The algorithm iterates from i = 0 to n-1, incrementally building the minimum cost arrangement using previously computed subproblem solutions stored in min_cost. As each subproblem only references earlier computed values (from j < i), the solution respects dependencies and evaluates all required configurations for the recurrence relation. Thus, it correctly fills the dynamic programming table by never referencing an uncomputed value.

E. Correctness of the Algorithm

For n = 0, the base case returns (0, 0, [])

For n> 0, the solution for the entire problem is stored in min_cost[n-1], representing the minimum cumulative height cost for arranging all sculptures from index 0 to n-1, on platforms within the width constraint. By returning min_cost[n-1], the algorithm provides the correct optimal height cost arrangement for all sculptures, thereby confirming correctness.

F. Pseudocode for Algorithm4

```
function program4(n, W, heights, widths):
    if n == 0:
    return (0, 0, [])

    initialize min_cost array of size n to infinity
    initialize platforms array of size n to 0
    initialize sculptures_on_last_platforms array of size n to 0
    for i from 0 to n - 1:
    platform_max_height = 0
    platform_width = 0

    for j from 0 to i:
    platform_width += widths[j]
    platform_max_height = max(platform_max_height, heights[j])

    if platform_width <= W:
    if j == 0:
            if min_cost[i] > platform_max_height:
            min_cost[i] = platform_max_height
            platforms[i] = 1
        sculptures_on_last_platforms[i] = i + 1
    else:
            if min_cost[i] > min_cost[j - 1] + platform_max_height:
            min_cost[i] = min_cost[j - 1] + platform_max_height
            platforms[i] = platforms[j - 1] + 1
        sculptures_on_last_platforms[i] = i - j + 1
    initialize sculptures_per_platform as an empty list
    i = n - 1
    while i >= 0:
    add sculptures_on_last_platforms[i] to sculptures_per_platform
    decrement i by sculptures_on_last_platforms[i]

    return (platforms[n - 1], min_cost[n - 1], reverse of sculptures_per_platform)
```

2.3. Algorithm 5

2.3.1 Θ(n^2) Time Dynamic Programing Algorithm Design

We can further optimize Algorithm 4 down to Θ(n^2) by avoiding the innermost for loop in the program. The key idea is to precompute cumulative widths up to each sculpture, allowing us to efficiently check if a platform is feasible and track the maximum height. The rest of the logic remains the same.

We design the dynamic programming Algorithm 5 with two different approaches:

Program5A: A Top-Down Recursive Design with Memoization

We designed a helper function *find_min_cost(i)* to perform the recursion. Starting from the highest level (i.e., the larger problem *find_min_cost(n-1)*), the function works its way down by solving progressively smaller subproblems through recursive calls. To avoid recalculating results, the function uses memoization (*min_cost_memo*). This array stores the minimum cost for each index *i* once computed.

The recursion completes with the base case (when index i == 0). It represents the smallest possible problem (just one sculpture), which has a straightforward solution.

2.3.2 Proof of Correctness

A. Defining the Subproblems

Let OPT(i) represent the minimum total height cost for arranging sculptures from index 0 to i into platforms, considering every possible way to split the sculptures, subject to the width constraint W. This function considers each platform arrangement and calculates the cumulative height, accounting for all preceding platforms.
Subproblems are defined by finding optimal configurations for subsets of sculptures from index 0 up to i, where i < n.

B. Formulating the Recurrence

The recurrence relation for OPT(i) is as follows:

Base Case: If i = 0, no sculptures exist, so OPT(0) = 0.
Recursive Case: For each i > 0, the algorithm recursively evaluates all possible platform start points j (from i to 0),

where each j defines a potential platform from sculpture j to i. If the platform width from j to i does not exceed W, the recurrence for OPT(i) is:

$$OPT(i) = \min_{0 \le j \le i} \{ OPT(j-1) + \max(heights[j:i+1]) \}$$

where max(heights[j:i+1]) denotes the maximum height for sculptures from j to i, making sure the cumulative width respects W.

C. Proof of the Recurrence Correctness

To confirm the recurrence's correctness, we examine how the algorithm computes OPT(i):

Case 1: If a platform ends at j, then OPT(i) must add the maximum height for sculptures spanning from j to i, while also accumulating the optimal height cost for preceding sculptures up to j-1 (OPT(j-1)).

Case 2: For each feasible arrangement ending at i, all valid configurations are evaluated through recursive calls to find min_cost_memo(j-1). The algorithm ensures OPT(i) minimizes the total height cost by selecting the optimal configuration across all possible placements.

Thus, the recurrence is correct because it evaluates each valid configuration by recursively placing platforms at every potential starting point j. By caching results in min_cost_memo, redundant evaluations are avoided.

D. Evaluation of the Recurrence Relation

The algorithm recursively evaluates the recurrence OPT(i) for each index i from 0 to n-1, using memoization to store previously computed results in min_cost_memo.
For each index i, it only references earlier computed values (from $j \leq i$) to compute the current subproblem solution. This ensures that the recurrence relation is evaluated correctly without redundant computation. By the time OPT(n-1) is computed, all dependent subproblems have been resolved.

E. Correctness of the Algorithm

For n = 0, the base case returns (0, 0, []), since there are no sculptures.
For n > 0, the solution for the entire problem is stored in min_cost_memo[n-1], representing the minimum cumulative height cost for arranging all sculptures from index 0 to n-1, on platforms within the width constraint. By returning min_cost_memo[n-1], the algorithm provides the correct optimal height cost arrangement for all sculptures, thereby confirming correctness.

F. Pseudocode

```
function program5A(n, W, heights, widths):
        if n == 0:
                return (0, 0, [])

        initialize min_cost_memo to -1 for each index in range 0 to n-1
        initialize platforms and sculptures_on_last_platforms to -1 for each index
        cumulative_width[0] = 0

        for i = 1 to n:
                cumulative_width[i] = cumulative_width[i - 1] + widths[i - 1]

        function find_min_cost(i):
        if i == 0:
                platforms[0] = 1
                sculptures_on_last_platforms[0] = 1
                min_cost_memo[0] = heights[0]
                return heights[0]

        if min_cost_memo[i] != -1:
                return min_cost_memo[i]
        set min_cost to infinity
```

```
        max_height = 0

    for j = i down to 0:
        max_height = max(max_height, heights[j])
        width = cumulative_width[i + 1] - cumulative_width[j]
        if width <= W:
        current_cost = max_height
        if j > 0:
            current_cost += find_min_cost(j - 1)

        if current_cost < min_cost:
            min_cost = current_cost
            platforms[i] = platforms[j - 1] + 1 if j > 0 else 1
            sculptures_on_last_platforms[i] = i - j + 1

        min_cost_memo[i] = min_cost
        return min_cost

    total_min_cost = find_min_cost(n - 1)
    sculptures_per_platform = []
    i = n - 1

    while i >= 0:
        sculptures_per_platform.append(sculptures_on_last_platforms[i])
        i -= sculptures_on_last_platforms[i]
    return (platforms[n - 1], total_min_cost, reverse(sculptures_per_platform))
```

Program5B: An Iterative Bottom-Up Design

The design is straightforward, as we already used a bottom-up approach for Algorithm 4. We just make the modification of precomputing the cumulative widths for each sculpture.

In contrast to *program5A*, *program5B* starts from the smallest subproblem (just the first sculpture) and progressively builds up the solution to the larger problem (up to all sculptures), filling in the *min_cost* array from left to right.

2.3.2 Proof of Correctness

A. Defining the Subproblems

Let OPT(i) represent the minimum total height cost for arranging sculptures from the start up to and including index i into platforms, with each platform constrained by the width W. The subproblems are defined by calculating optimal configurations for sculptures from index 0 up to i, where i < n.

B. Formulating the Recurrence

The recurrence relation for OPT(i) is as follows:

Base Case: If i = 0, no sculptures exist, so OPT(0) = 0.

Recursive Case: For each i > 0, the algorithm iterates over possible start points j (from i down to 0). Each j defines a potential platform from sculpture j to i. If the width of the platform from j to i does not exceed W, the recurrence for OPT(i) is:

$$OPT(i) = \min_{0 \le j \le i} \{ OPT(j-1) + \max(heights[j:i+1]) \}$$

Here, max(heights[j:i+1]) is the maximum height of sculptures from j to i, while OPT(j-1) represents the cost up to the start of this platform. This recurrence relation ensures the minimum height cost is accumulated up to each i by examining each valid configuration for the current platform.

C. Proof of the Recurrence Correctness

To confirm the recurrence's correctness, we analyze the computation of OPT(i):

Case 1: If the platform ends at j, then OPT(i) adds the maximum height of sculptures spanning from j to i and accumulates the optimal height cost up to j-1 (stored in OPT(j-1)).
Case 2: The algorithm evaluates all feasible configurations from j to i, selecting the configuration that yields the minimum cumulative height cost. By iterating from i down to 0, each valid configuration is examined in a structured manner.
This iterative approach ensures all possible configurations for each subproblem are considered, capturing the minimal height cost across valid platform placements.

D. Evaluation of the Recurrence Relation

Algorithm 5B fills the min_cost array iteratively from i = 0 up to n-1. Each iteration for i accesses only previously computed values (j < i), ensuring that the dependencies for each subproblem are resolved in the correct order. The algorithm guarantees that the recurrence relation is accurately evaluated by iterating through all feasible configurations and updating min_cost[i] with the minimum cumulative cost.

E. Correctness of the Algorithm

For n = 0, the base case returns (0, 0, []), which is correct as no sculptures need to be arranged.
For n > 0, the final solution for the problem is stored in min_cost[n-1], representing the minimum cumulative height cost for arranging all sculptures from index 0 to n-1 within the width constraint W.
The algorithm concludes by returning min_cost[n-1], confirming that it provides the optimal solution.

F. Pseudocode

```
function program5B(n, W, heights, widths):
    if n == 0:
    return (0, 0, [])

    initialize min_cost array of size n to infinity
    initialize platforms array of size n to 0
    initialize sculptures_on_last_platforms array of size n to 0
    cumulative_width[0] = 0
    for i = 1 to n:
    cumulative_width[i] = cumulative_width[i - 1] + widths[i - 1]

    for i from 0 to n - 1:
    set platform_max_height to 0
    for j from i down to 0:
    platform_max_height = max(platform_max_height, heights[j])
    platform_width = cumulative_width[i + 1] - cumulative_width[j]
```

```
            if platform_width <= W:
            if j == 0:
                    if min_cost[i] > platform_max_height:
                    min_cost[i] = platform_max_height
                    platforms[i] = 1
                sculptures_on_last_platforms[i] = i + 1
            else:
                    if min_cost[i] > min_cost[j - 1] + platform_max_height:
                    min_cost[i] = min_cost[j - 1] + platform_max_height
                    platforms[i] = platforms[j - 1] + 1
                sculptures_on_last_platforms[i] = i - j + 1

            initialize sculptures_per_platform as an empty list
            set i to n - 1
            while i >= 0:
            add sculptures_on_last_platforms[i] to sculptures_per_platform
            decrement i by sculptures_on_last_platforms[i]

        return (platforms[n - 1], min_cost[n - 1], reverse(sculptures_per_platform))
```

Time Complexity

The time complexity of Algorithm 5 is $\Theta(n^2)$, where n is the number of the given sculptures.

In *program5A*, the recursion depth is Θ(n). For each call, we have an inner loop over all potential subarrays (iterating backward from *j* to *i*), leading to Θ(n) amount of work. This is the most expensive operation in the program, completing in $\Theta(n^2)$ time.

Similarly, in *program5B*, the main loop in the program is nested. The outer loop iterates over each sculpture *i*, taking Θ(n) time. For each sculpture *i*, it iterates backward over all possible starting points *j*, which take Θ(n) amount of work. This is the most expensive operation in the program, completing in $\Theta(n^2)$ time.

Note that for Algorithm 5, we avoid a third loop by precomputing cumulative widths in the *cumulative_width* array (which takes Θ(n) time but is done once and before the main loops), allowing us to compute the width of any subarray of sculptures in Θ(1) time.
Therefore, program5A and program5B have a time complexity of $\Theta(n^2)$.

3. Experimental Comparative Study

We conducted a comprehensive experimental comparative study to evaluate the performance and correctness of various algorithm implementations for solving ProblemG. This study involved testing our implementations extensively across multiple input sizes, focusing on both running time and output quality.

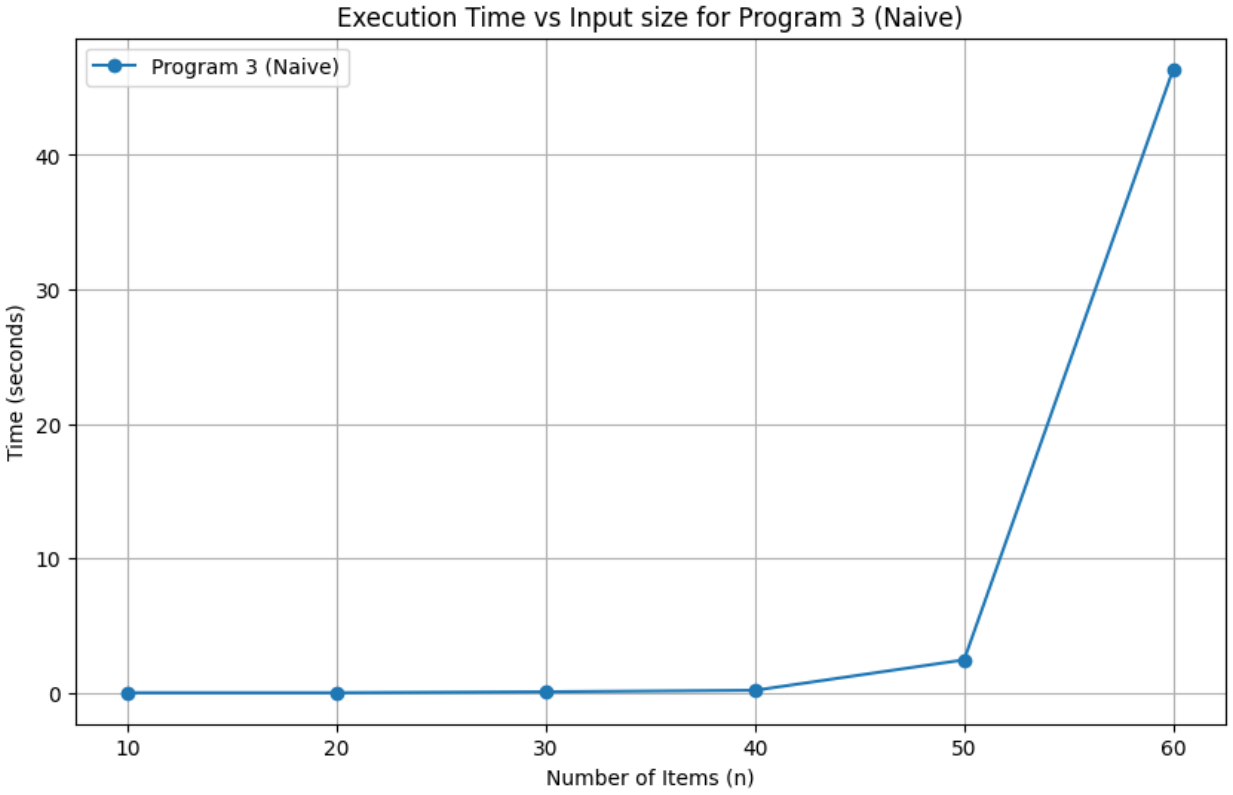
The constraints provided (e.g., 1 ≤ n, W < 10^5 and 1 ≤ h[i], w[i] < 10^5) were used to guide the design and optimization of each program.

We generated random input files of varying sizes (e.g., n = 1000, 2000, 3000, 4000, 5000) to test each program's efficiency. The experimental data sets allowed us to observe how well each implementation

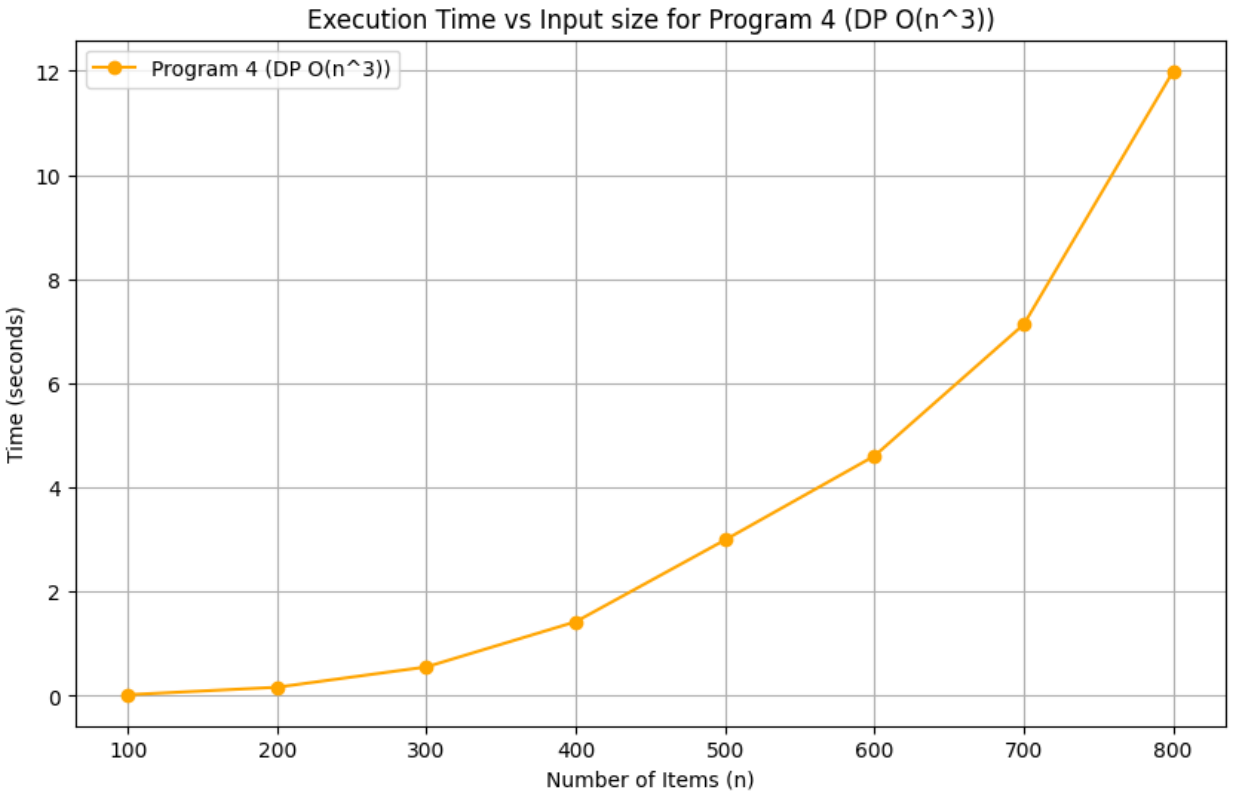
scaled with input size, depending on the quality and design of the algorithm. Performance comparisons were presented in a series of plots, showing the running time for each program with respect to input size. These visualizations provided clear insights into each algorithm's time complexity and scalability:

- Plot 3: Running time of Program 3 (naive approach) as input size increases
- Plot 4: Running time of Program 4 (dynamic programming with $\Theta(n^3)$ complexity).
- Plot 5: Running time of Program 5A (top-down dynamic programming with memoization).
- Plot 6: Running time of Program 5B (bottom-up dynamic programming).
- Plot 7: Overlay of Plots 3, 4, 5, and 6, comparing the performance of Programs 3, 4, 5A, and 5B.
- Plot 8: Overlay of Plots 5 and 6, highlighting the comparative performance of Programs 5A&5B.
- Plot 9: Output quality comparison of Program 1 against Programs 3, 4, 5A, and 5B, showing the deviation of the greedy algorithm's results from optimal solutions using the ratio (hg − ho)/ho.

Plot 3: *This plot shows the running time of* ***Program 3*** *as input size increases. The steep curve indicates that Program 3 has high time complexity, likely due to its brute-force or naive approach. As input size grows, the runtime quickly becomes impractical, emphasizing its inefficiency for larger inputs.*

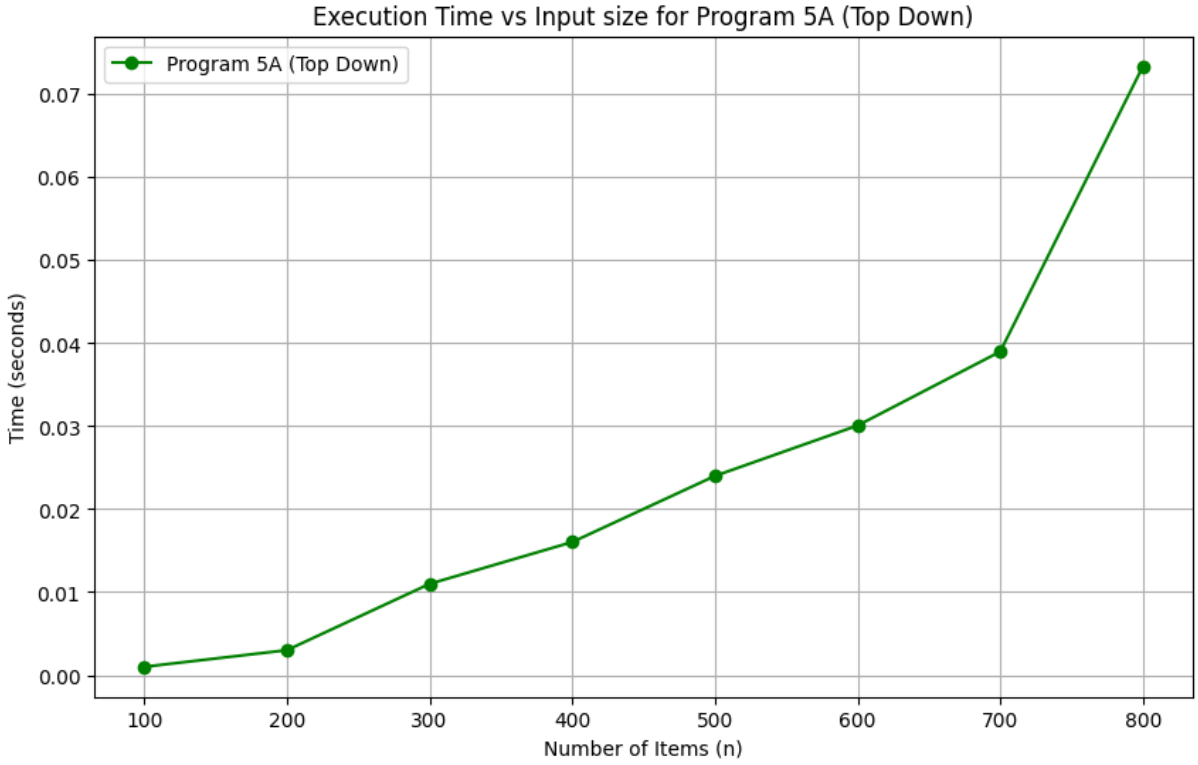


Plot 4: *This plot illustrates the running time of* ***Program 4****, a dynamic programming approach with* $\Theta(n^3)$ *complexity. The cubic growth pattern aligns with the expected time complexity, and the graph demonstrates that Program 4 handles moderate input sizes more efficiently than Program 3, though it still faces limitations as input size grows.*

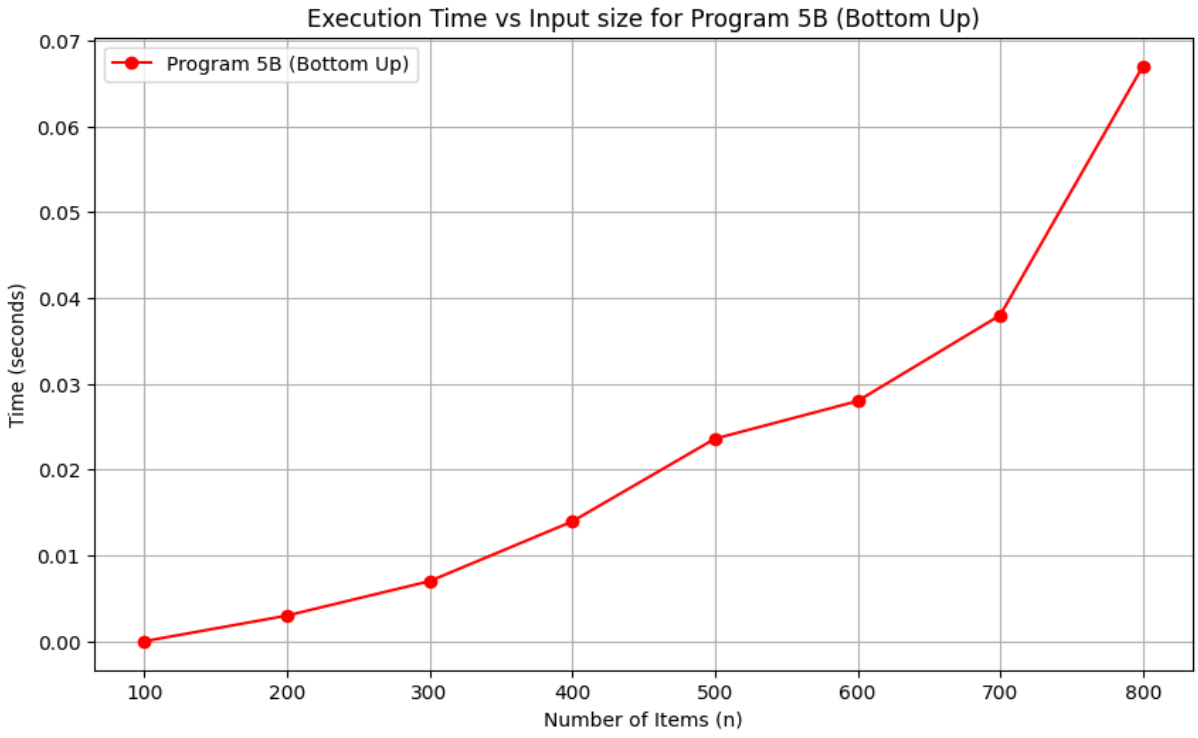


This plot illustrates the execution time of Program 4, which implements a dynamic programming approach with a $\Theta(n^3)$ time complexity. As the input size increases, the cubic growth pattern becomes evident, with execution time escalating rapidly. For input sizes approaching 800 items, the runtime exceeds 10 seconds, highlighting the computational intensity of this approach. Although Program 4 is more efficient than a naive exponential solution, its $\Theta(n^3)$ complexity makes it suitable only for moderately large datasets, as performance degradation becomes significant with larger inputs. This highlights the need for more optimized algorithms in applications requiring scalability.

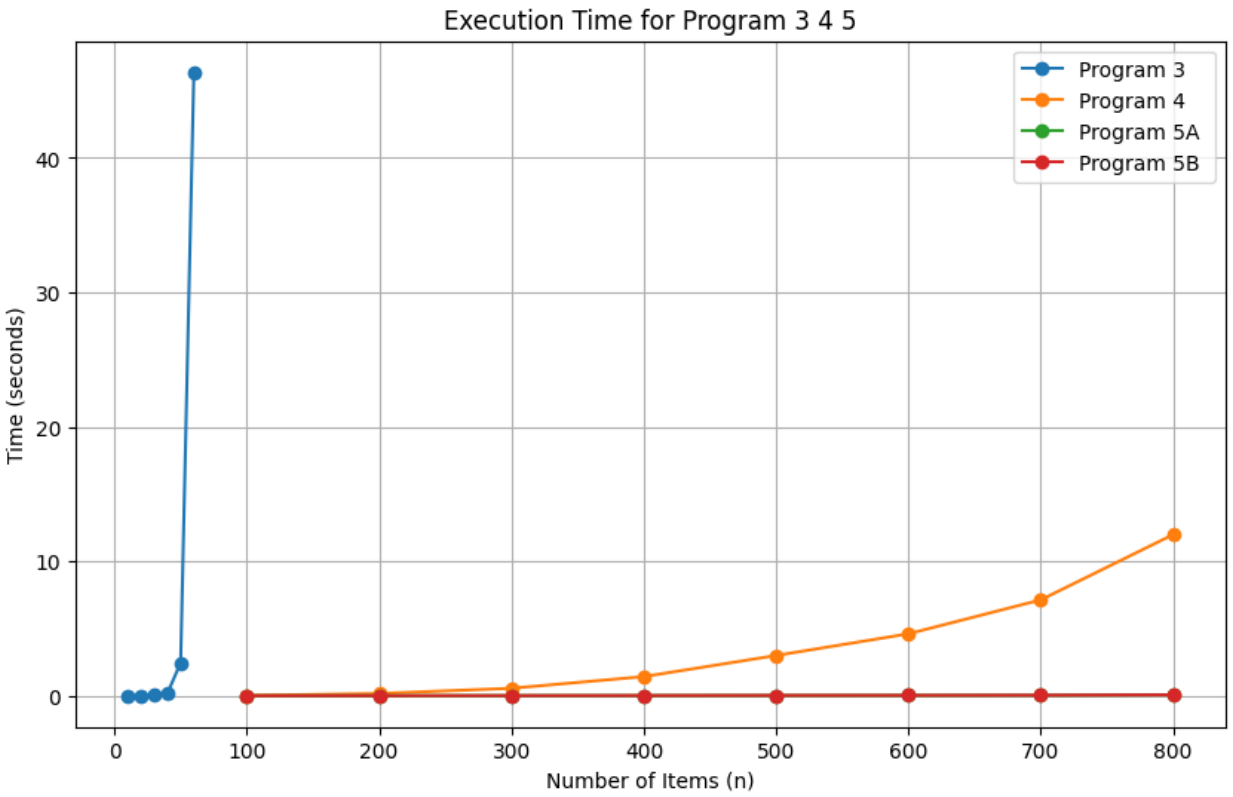
***Plot 5**: This plot shows the running time of **Program 5A**, a top-down dynamic programming algorithm with memoization. Quadratic growth is evident, and the program's performance is significantly better than the naive and $\Theta(n^3)$ approaches, making it feasible for larger inputs.*



***Plot 6**: This plot presents the running time of **Program 5B**, which uses a bottom-up iterative approach. Like Program 5A, it exhibits quadratic growth but performs slightly better due to its iterative design, making it more efficient for larger input sizes.*



Plot7: *Overlay Plots 3,4,5,6 and contrasting the performance of* ***Programs 3,4,5A,5B.***



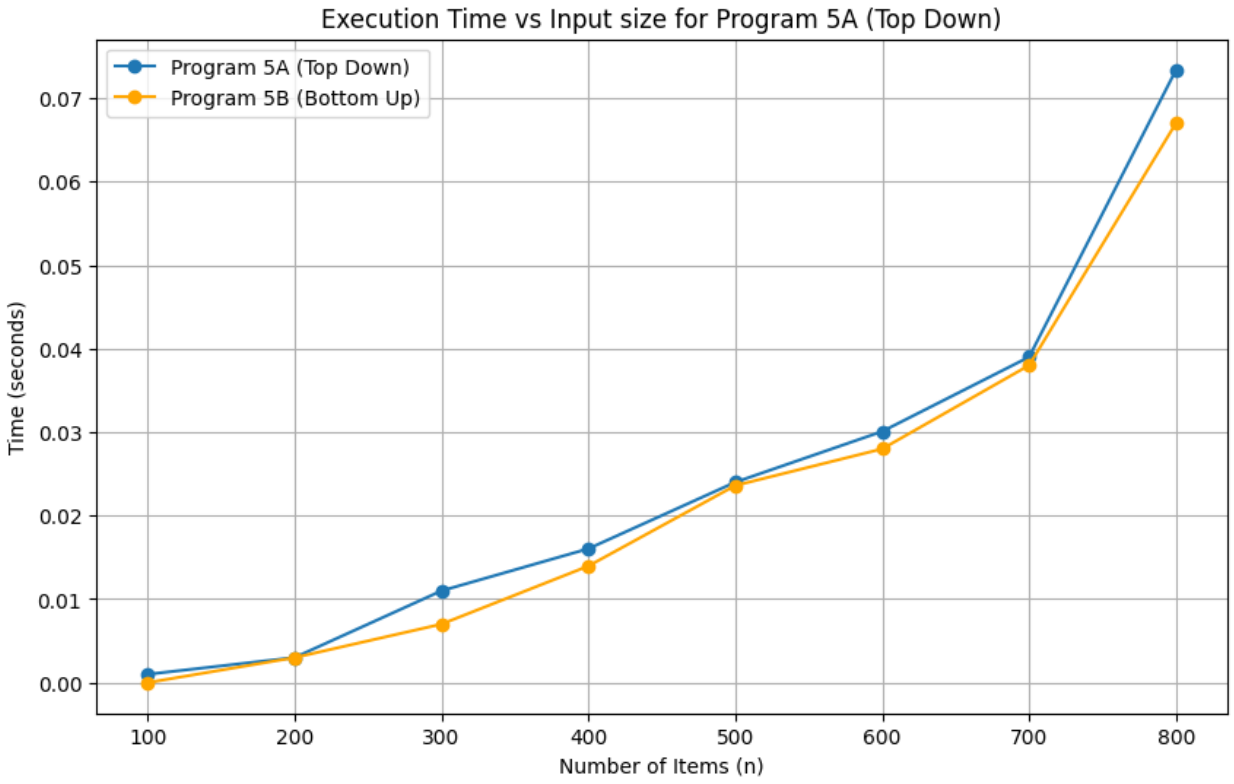
This overlay plot highlights the stark differences in execution time between the naive, cubic, and quadratic algorithms as input size increases. Program 3, with its exponential complexity, exhibits an extreme increase in execution time, making it impractical for inputs beyond a certain size. This rapid growth reflects the inefficiency of brute-force or naive methods, which become infeasible even at relatively small input sizes.
Programs 4, 5A, and 5B, on the other hand, show much more controlled growth in execution time:

- **Program 4** ($\Theta(n^3)$): As expected, Program 4 demonstrates a cubic time complexity. Although significantly faster than Program 3, it still shows a pronounced upward trend, making it suitable only for moderate input sizes.

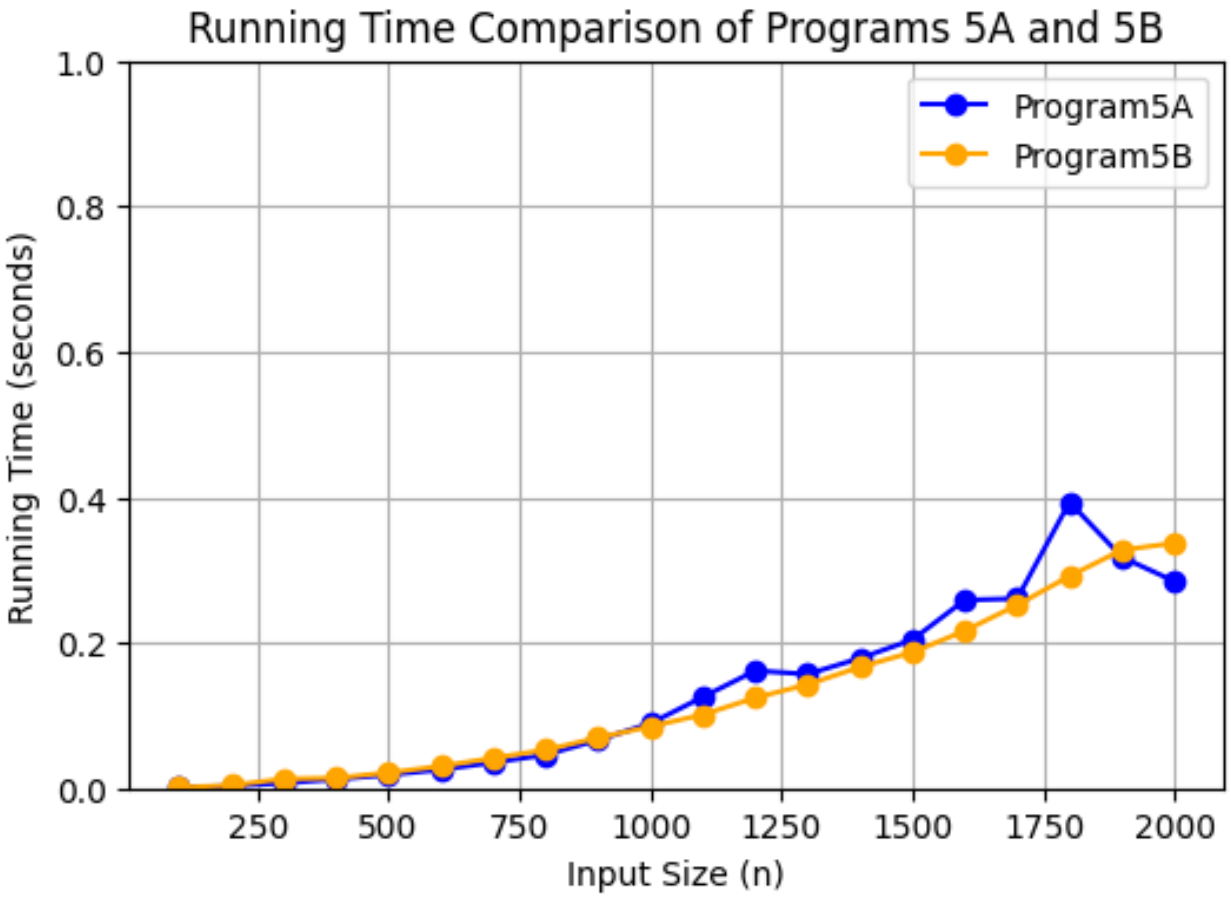
- **Program 5A** (Top-Down DP with Memoization, $\Theta(n^2)$) and **Program 5B** (Bottom-Up DP, $\Theta(n^2)$): Both 5A and 5B exhibit quadratic growth, making them the most efficient options in this comparison. The slight edge of Program 5B over 5A reflects the benefits of an iterative bottom-up approach over recursive memoization in terms of both time and memory management.

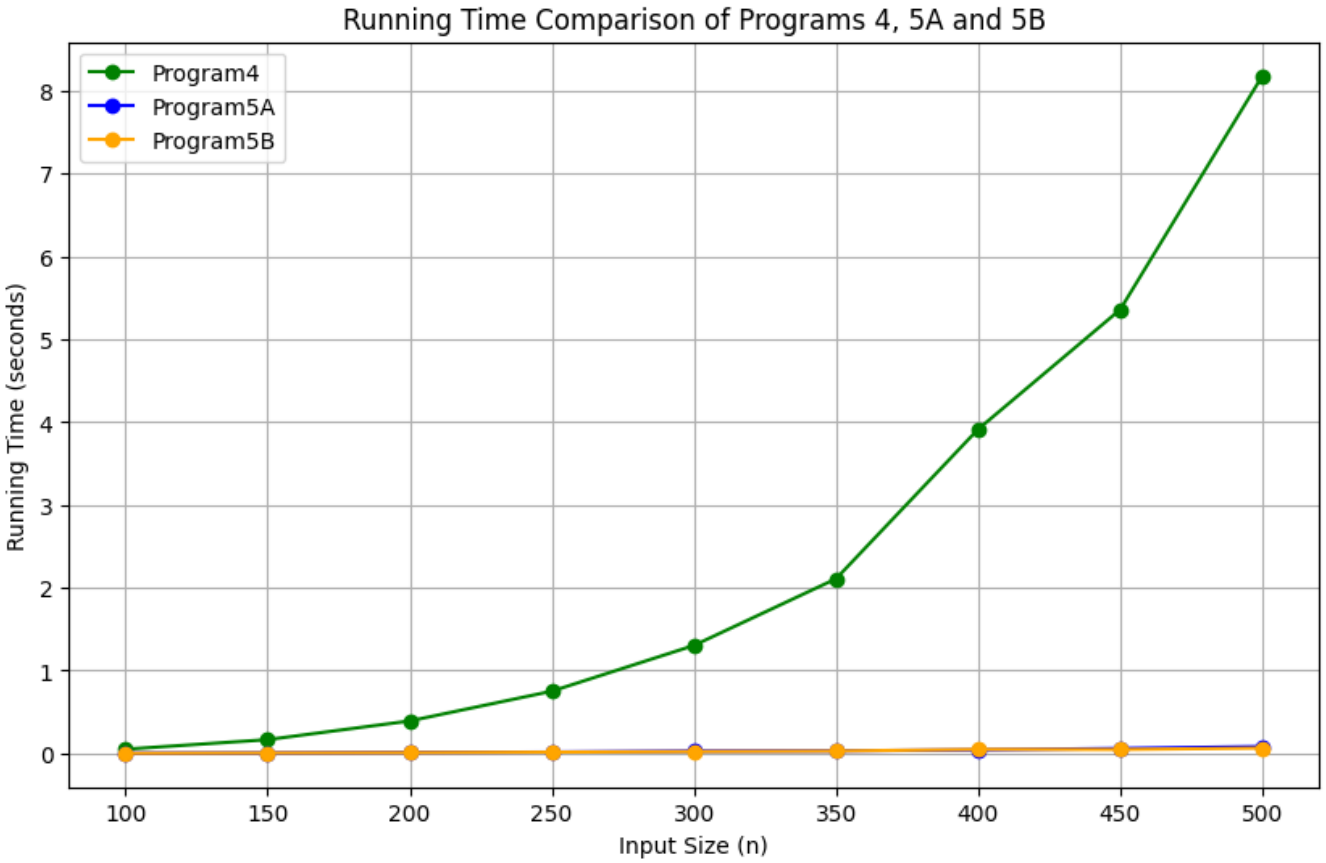
Overall, this comparison underscores the efficiency improvements gained by transitioning from naive to dynamic programming solutions and then optimizing further within DP approaches. For larger input sizes, the $\Theta(n^2)$ complexity of Programs 5A and 5B makes them the most practical choices, while Program 3's exponential growth renders it infeasible and Program 4's cubic growth limits its scalability.

***Plot 8**: This overlay plot contrasts the performance of **Programs 5A and 5B**. Both programs show similar quadratic growth, but Program 5B has a slight performance edge, demonstrating the benefits of a bottom-up iterative approach over the top-down recursive approach with memoization.*

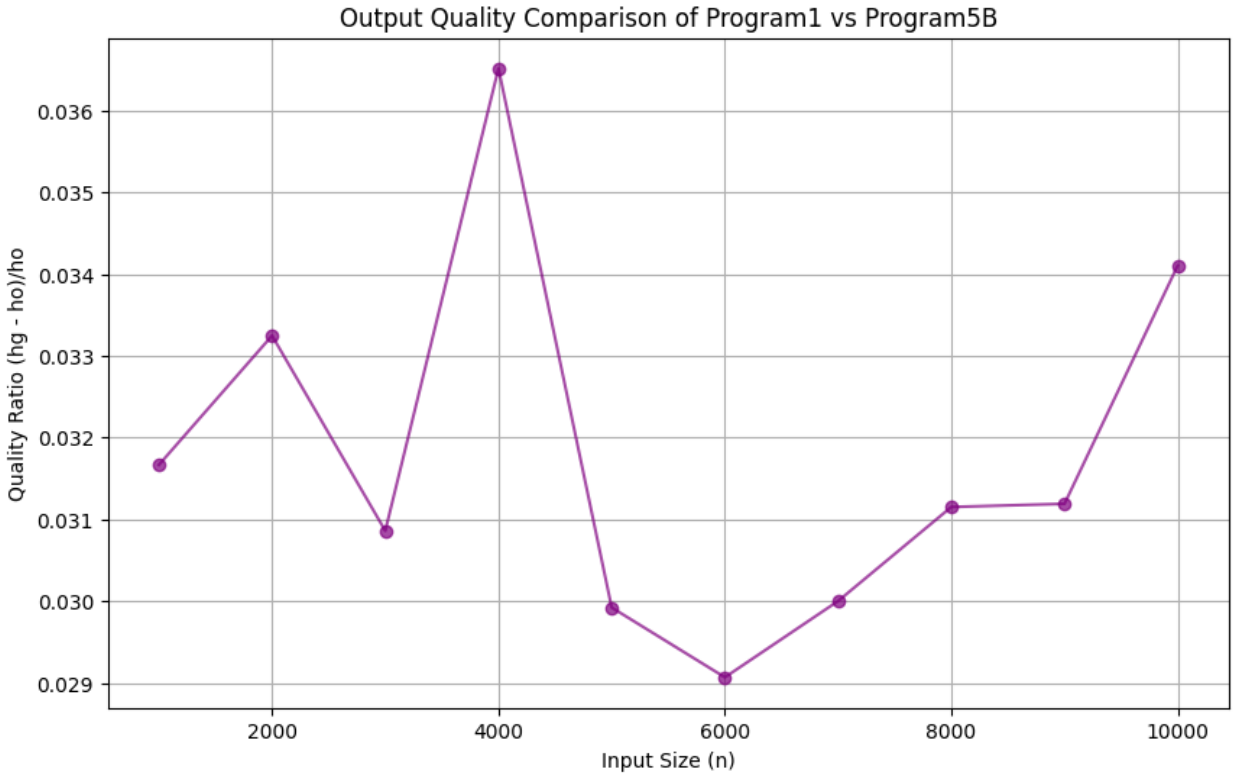


This overlay plot compares the execution times of Program 5A (Top-Down DP with Memoization) and Program 5B (Bottom-Up DP). Both algorithms exhibit similar quadratic growth, confirming their $\Theta(n^2)$ time complexity. However, Program 5B consistently shows a slight performance advantage over Program 5A as input size increases. This marginal edge highlights the efficiency gains of the bottom-up approach, which avoids the overhead of recursive calls and relies on an iterative structure. In larger datasets, Program 5B's iterative design makes it the more scalable and efficient choice between the two.


Running Time Comparison of Programs 5A and 5B
Program5A
Program5B
Running Time (seconds)
Input Size (n)
0.0
0.2
0.4
0.6
0.8
1.0
250
500
750
1000
1250
1500
1750
2000


Running Time Comparison of Programs 4, 5A and 5B
Program4
Program5A
Program5B
Running Time (seconds)
Input Size (n)
0
1
2
3
4
5
6
7
8
100
150
200
250
300
350
400
450
500

Plot9: *This plot shows the performance of Program 1, a greedy algorithm originally designed for a non-increasing dataset, against the optimal values provided by Program5B on randomly generated data. The comparison shows that Algorithm 1 produces significantly higher ratios of (hg - ho)/ho when contrasted with Algorithm 5B* (Bottom-Up Dynamic Programming) .



This plot shows the output quality comparison of Program 1 (Greedy Algorithm) versus Program 5B (Dynamic Programming-based Optimal Algorithm) across various input sizes (1000 to 10,000 sculptures). The quality ratio on the y-axis, calculated as (Hg - Ho) / Ho, measures the deviation of the greedy solution from the optimal one, with Hg representing the total height produced by Program 1 and Ho representing the optimal height from Program 5B.

For each input size n, random heights and widths were generated for the sculptures, with the following constraints:

- Heights: Random integers between 1 and 25, representing the heights of the sculptures.
- Widths: Random integers between 1 and 10, representing the widths of the sculptures.
- The platform width constraint W was set to 25 for consistency.

These random cases provide a general comparison between the algorithms but may not highlight specific performance trends across structured scenarios (e.g., non-increasing or unimodal patterns). As seen in the graph, the quality ratio varies slightly across input sizes, indicating that the greedy solution's deviation from the optimal solution is not consistent and can fluctuate due to the random nature of the input.

For the structured analysis of Program 1 (Greedy Algorithm) versus Program 5B (Dynamic Programming Optimal Solution), we conducted a series of controlled experiments across distinct scenarios. Each scenario was designed

to isolate specific input patterns that might influence the quality of results produced by the greedy approach when compared to the optimal dynamic programming solution.

Platform Width Constraint: The maximum allowable platform width for each scenario was set to 100.

Widths: For each sculpture, widths were randomly generated within the range 1 to 100.

Heights: Heights were generated differently depending on the scenario:

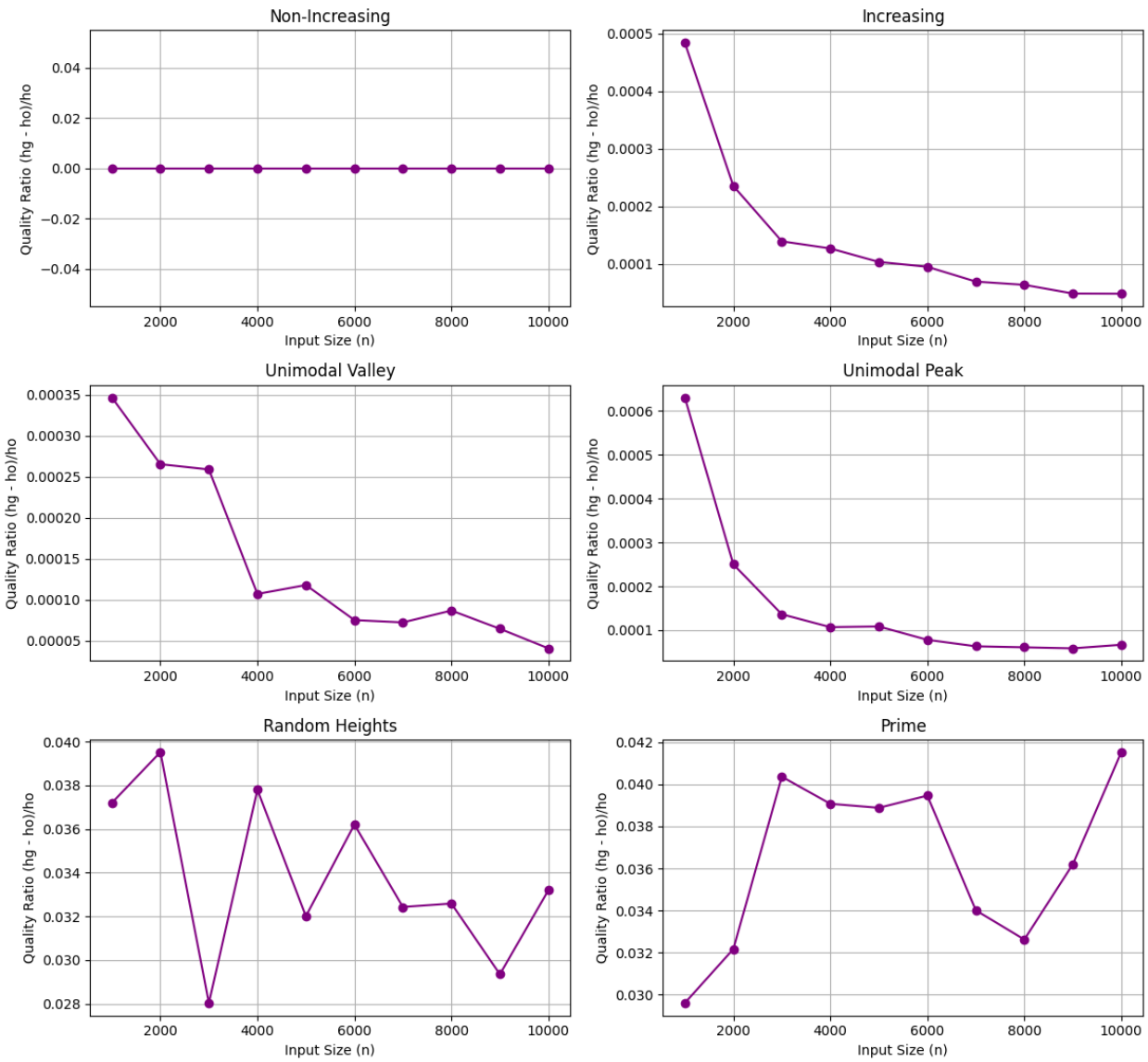
- **Non-Increasing**: Heights were sorted in descending order, with random values between 1 and 1000.
- **Increasing**: Heights were sorted in ascending order, with random values between 1 and 1000.
- **Random Heights**: Heights were randomly generated between 1 and 1000, without any specific ordering.
- **Unimodal Valley**: Heights formed a valley shape, where the left side (first half) decreased from a peak value, and the right side (second half) increased.
- **Unimodal Peak**: Heights formed a peak shape, where the left side (first half) increased to a peak, and the right side (second half) decreased from that peak.
- **Prime**: Heights were randomly selected from a list of prime numbers below 100, creating a unique distribution pattern.

Input Sizes: The input_sizes variable ranged from 1000 to 10,000 in increments of 1000, allowing the algorithms to be tested across a wide variety of input sizes.

Number of Samples per Scenario: For each scenario, 10 input sizes (from 1000 to 10,000, increments of 1000) were tested, giving 10 samples per scenario.

Observations:

- **Non-Increasing and Increasing Scenarios:** As anticipated, Program 1 performed well in the non-increasing scenario, with quality ratios being zero, indicating no deviation from optimal results. In the increasing scenario, however, the quality ratio fluctuated more significantly, suggesting that Program 1's design is less effective when the input is ordered in ascending heights.
- **Unimodal Valley and Unimodal Peak:** Both scenarios exhibited similar trends, with quality ratios initially higher for smaller inputs, then decreasing as input sizes increased. This observation suggests that while Program 1 struggles slightly with these structures, it approximates the optimal solution better as dataset size increases.
- **Random Heights:** The random scenario showed the most fluctuation in quality ratios across input sizes. This outcome reflects the unpredictability of unstructured data and the limitations of the greedy algorithm in approximating an optimal solution under random conditions.
- **Prime Numbers:** The quality ratio was consistently higher, revealing that the irregular intervals of prime numbers made it more challenging for the greedy algorithm to approximate the optimal solution effectively.



Building on the initial analysis, we expanded the set of controlled scenarios to include **Odd Heights**, **Even Heights**, and **Random Heights with Fixed Width** conditions. These additional scenarios were chosen to assess how specific constraints impact the quality of Program 1 (Greedy Algorithm) when compared to Program 5B (Optimal Dynamic Programming Algorithm).

Odd Heights and Even Heights:

- **Design:** Heights were restricted to either odd or even numbers within the same general range as previous experiments.
- **Observation:** These scenarios showed slightly more consistent quality ratios across input sizes, suggesting that Program 1 performs relatively stable when heights have a predictable parity (either all odd or all even).
- **Conclusion:** While the quality ratios were somewhat stable, they still reflected the limitations of Program 1 compared to Program 5B. This outcome highlights that while height uniformity in terms of parity may reduce extreme fluctuations, the greedy approach still shows deviation from optimal results.

Random Heights with Fixed Width:

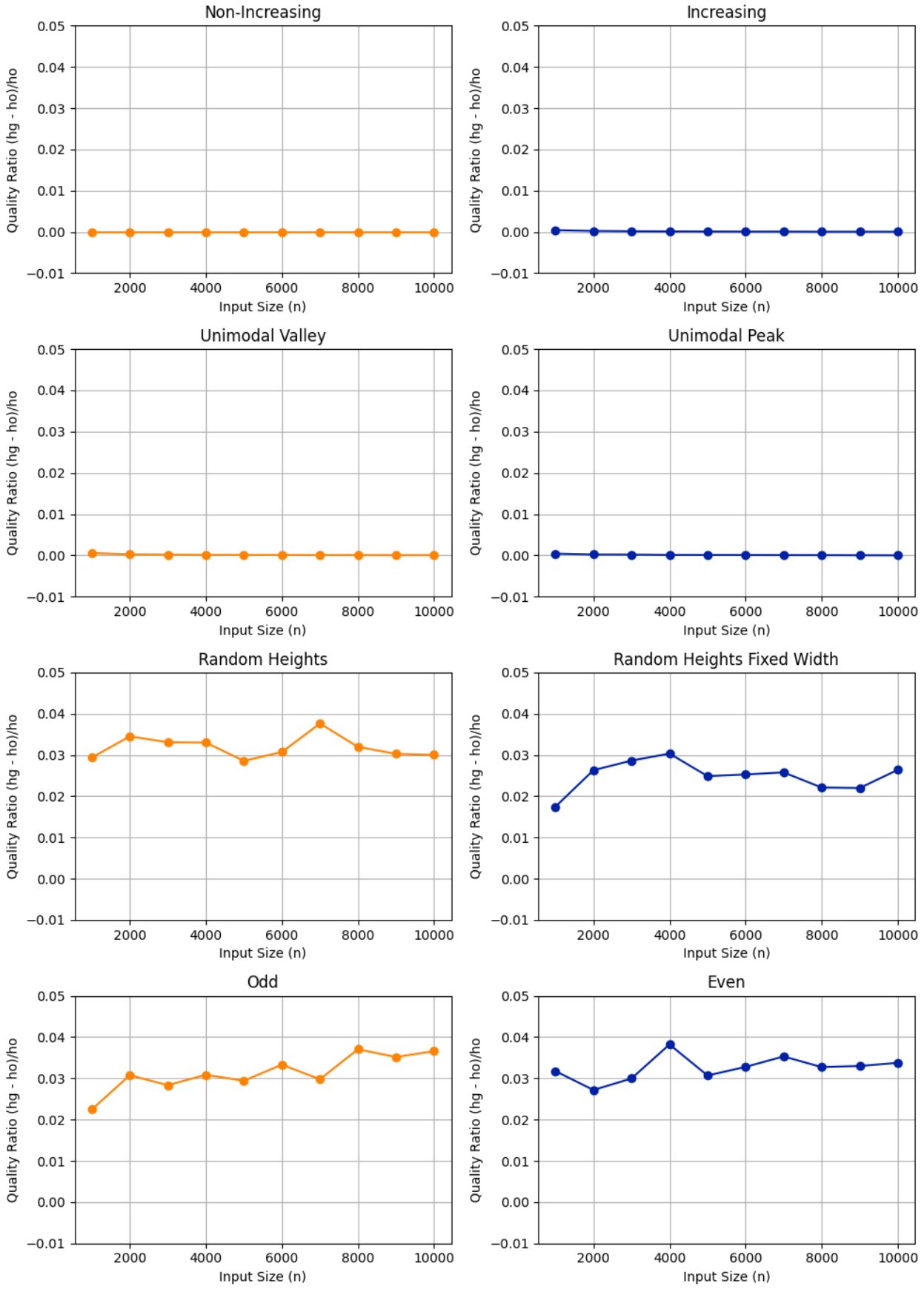
- **Design:** Heights were randomly generated, but each sculpture's width was kept constant (set to a fixed width of 10).
- **Observation:** The quality ratio showed lower variability compared to completely random heights and widths. By fixing the width, Program 1 had a more predictable structure to operate on, reducing the quality ratio slightly.
- **Conclusion:** Fixing widths provides Program 1 with a simplified structure, which helps it approximate the optimal solution more consistently. This setup underscores how reducing dimensional variability (e.g., holding width constant) can mitigate some of the greedy algorithm's limitations.

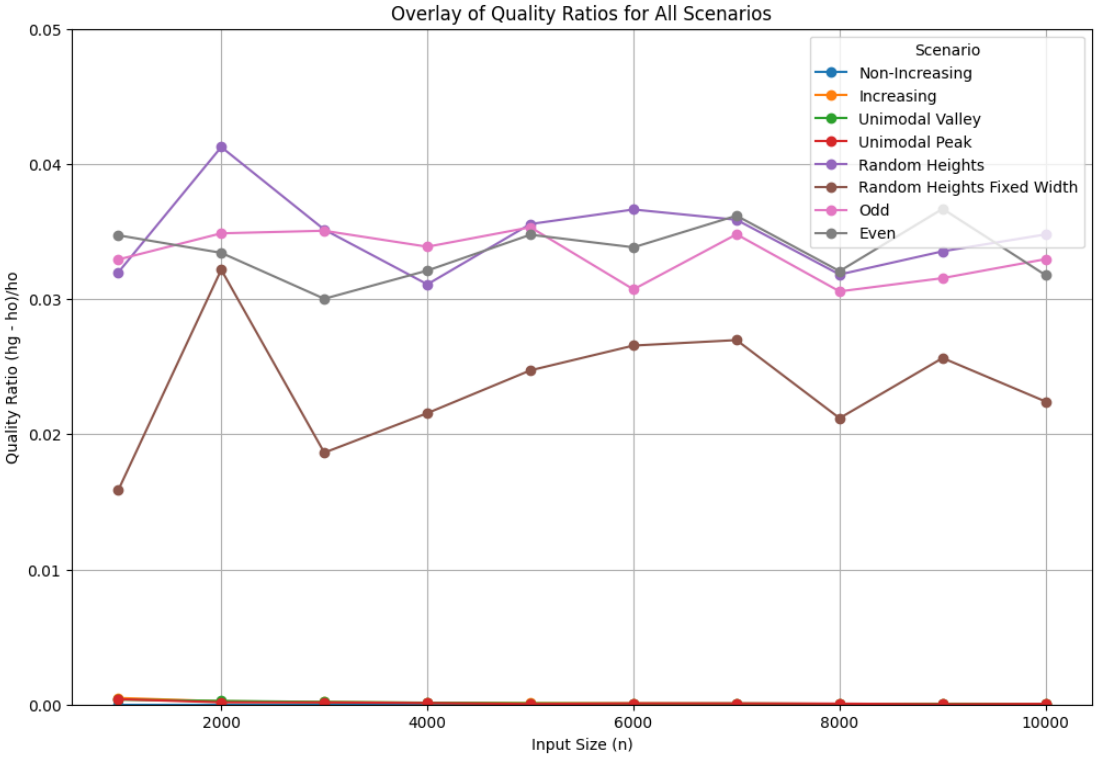
Standardizing Y-Axis for Cross-Scenario Comparisons

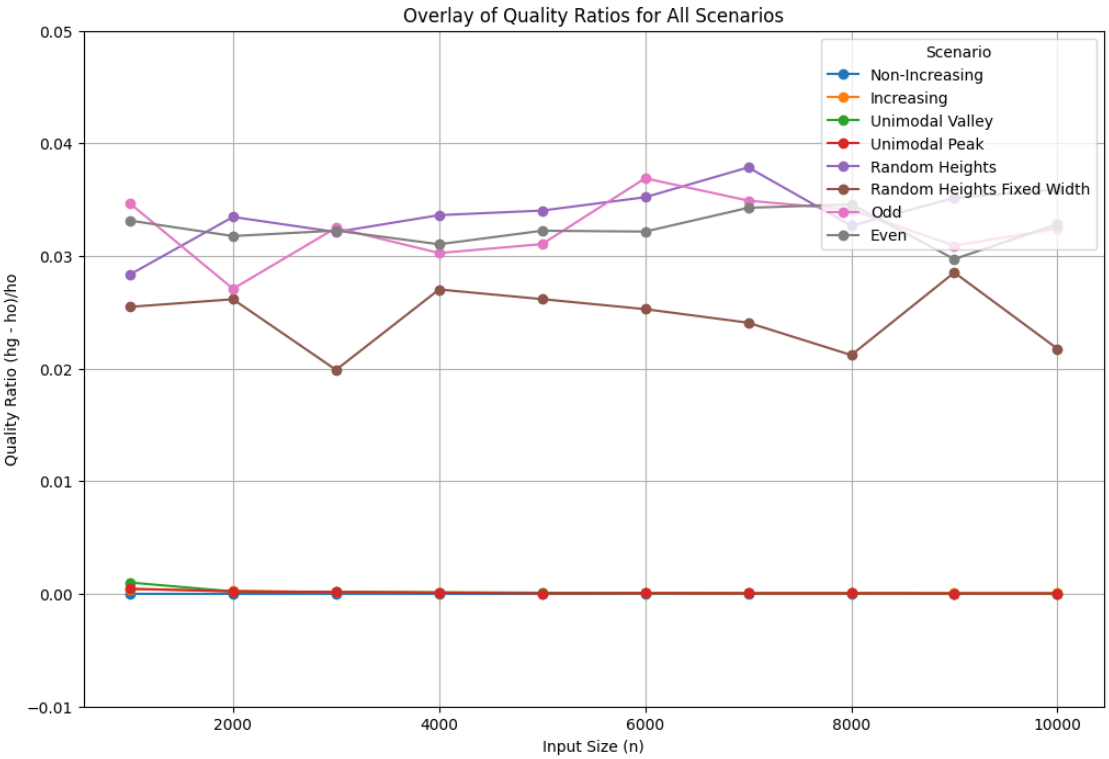
To facilitate direct comparison across all scenarios, we standardized the y-axis scale across each plot. This adjustment highlights the relative performance differences more clearly:

- **Comparison Benefits:** With a consistent y-axis, we can directly observe how Program 1's performance varies across scenarios. For instance, non-increasing and valley scenarios, with consistently low or near-zero quality ratios, stand out as best-case structures for Program 1. Meanwhile, the random scenarios (especially with varying widths) show higher ratios, confirming the limitations of the greedy algorithm in unstructured conditions.
- **Overlay Plot for Unified View:** We also introduced an overlay plot to visually compare all scenarios in a single graph. This visualization enables us to identify overarching trends and relationships between scenario structure and algorithm performance.

The standardized view and overlay plot reaffirm that while Program 1 performs effectively in structured scenarios like non-increasing or fixed widths, it shows notable deviations from optimal solutions in scenarios with more variability, such as random heights and widths or prime numbers.


Output Quality Comparison of Program1 vs Program5B by Scenario
Non-Increasing
Increasing
Unimodal Valley
Unimodal Peak
Random Heights
Random Heights Fixed Width
Odd
Even
Quality Ratio (hg - ho)/ho
Input Size (n)
0.05
0.04
0.03
0.02
0.01
0.00
−0.01
2000
4000
6000
8000
10000


Overlay of Quality Ratios for All Scenarios
Quality Ratio (hg - ho)/ho
Input Size (n)
Scenario
Non-Increasing
Increasing
Unimodal Valley
Unimodal Peak
Random Heights
Random Heights Fixed Width
Odd
Even
0.05
0.04
0.03
0.02
0.01
0.00
2000
4000
6000
8000
10000


Overlay of Quality Ratios for All Scenarios
Quality Ratio (hg - ho)/ho
Input Size (n)
Scenario
Non-Increasing
Increasing
Unimodal Valley
Unimodal Peak
Random Heights
Random Heights Fixed Width
Odd
Even
0.05
0.04
0.03
0.02
0.01
0.00
−0.01
2000
4000
6000
8000
10000

4. Conclusion

In this project, we implemented and analyzed a series of algorithms to tackle the sculpture arrangement problem with the goal of minimizing total platform height under various constraints. Our experiments revealed the performance and limitations of each approach, from the naive exponential algorithm (Program 3) to more sophisticated dynamic programming solutions (Programs 4, 5A, and 5B). Program 3's exponential time complexity made it impractical for large input sizes, emphasizing the inefficiencies of brute-force methods. In contrast, Programs 5A and 5B, with their quadratic time complexity, demonstrated a more balanced and scalable approach. Among these, the bottom-up approach of Program 5B had a slight performance edge, confirming the benefits of an iterative approach over recursion with memoization. Overall, the study reinforced the value of dynamic programming for optimization problems and highlighted the trade-offs between simplicity and efficiency in algorithm design.

5. Learning Experience and Technical Challenges:

This project deepened our understanding of algorithmic design, particularly in navigating the practical differences between naive, greedy, and dynamic programming strategies. Program 5A's recursive memoization approach posed challenges related to recursion depth and memory usage, while Program 5B's iterative design provided a clear efficiency advantage by avoiding recursion. Program 3, although straightforward to implement, quickly became unmanageable due to its exponential growth in runtime, underscoring the limitations of naive methods for larger datasets.

A key takeaway was appreciating the balance between runtime efficiency and solution quality. The greedy algorithm in Program 1, while fast, often produced suboptimal results, especially with unstructured data, as reflected in its (Hg - Ho) / Ho ratios compared to dynamic programming solutions. This highlighted the limitations of a greedy approach in complex optimization scenarios.

Overall, this project felt more like a collaborative study session, with frequent discussions on experimental setups, algorithms, outputs, and analysis. It was a rewarding experience that underscored the importance of teamwork, and it brought valuable insights into the trade-offs between algorithmic efficiency and accuracy, depending on problem requirements and constraints.