Algorithm Abstraction & Design

Programming Project

Milestone 1

Jessica Lourenco, Ozlem Polat, Zhengxiao Wang

COP4533

University of Florida

October 2024

1. Team Member Contributions

All team members contributed in some capacity to every part of the project in milestone 1 (discussing designs, writing code/tests, writing this report, or reviewing). Ozlem led the efforts for solving Problem1 and generating the experimental study charts, Zhengxiao led the same for Problem2, and Jessica led the analysis and report efforts.

2. Greedy Algorithms - Design & Analysis

2.1. Algorithm 1

2.1.1 ProblemS1

Given the heights $h_1, \ldots, h_n$, where $h_i \geq h_j \ \forall i < j$, 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 minimize the total height.

2.1.2 Θ(n) Time Greedy Algorithm Design

To design a greedy algorithm that solves *ProblemS1*, we take advantage of the fact that, in this particular case of the problem, we are given the heights of sculptures in decreasing order (i.e., monotonically non-increasing sequence). Therefore, the main idea of the algorithm is to iterate over the sculptures and place them in order in the platforms as long as they fit (do not exceed the width W). To ensure this, we keep track of the current width of the platform being arranged and make decisions based on how it compares to W.

At each step, the algorithm makes a greedy choice: it places the next sculpture on the current platform if the width constraint is satisfied. If adding the next sculpture would exceed the platform's width, the platform is finalized, and a new one is started.

Since the heights are monotonically non-increasing, placing the tallest sculptures on platforms earlier guarantees that we are always minimizing the total height for each platform.

Time Complexity

The time complexity of Algorithm1 is Θ(n), where n is the number of given sculptures.

The main loop of the algorithm iterates over each structure, consistently executing *n* times. The other two loops, used to calculate the total height cost of arrangements and count the number of sculptures, iterate over a *p* number of platforms generated by the algorithm. However, *p* can be at most *n* – we can still consider n iterations.

The remaining operations (including those within iterations) take Θ(1) time to complete. Note that for this project, we assume array operations such as appending to be Θ(1) since their analysis is not relevant in our case study.

Therefore, the algorithm takes Θ(n) time to complete.

2.1.3 Proof of Correctness

We will prove the correctness of the algorithm using a proof by induction.

Base Case (n = 1):

When only one sculpture is placed on a single platform, the algorithm correctly assigns the platform's height to the sculpture's. Since there is only one sculpture, the width constraint is trivially satisfied. Thus, the algorithm is correct for n = 1.

Inductive Hypothesis:

Assume that the algorithm is correct for n = k sculptures. That is, for k sculptures, the algorithm correctly arranges the sculptures on platforms such that the width of each platform does not exceed W, and the total height is minimized.

Inductive Step (n = k+1):

Consider the case with n = k+1 sculptures. The algorithm processes the first k sculptures correctly using the inductive hypothesis. When processing the (k+1)-th sculpture:

- If the current platform has enough width remaining, the sculpture is added.
- If the current platform is not wide enough, it is finalized, and a new one is started.

By placing the taller sculptures first and ensuring that platforms are finalized as soon as the width exceeds W, the algorithm guarantees that no sculptures can be placed more optimally. Any other arrangement would either exceed the width constraint or result in a larger total height, violating the solution's optimality.

2.1.4 Counter Examples

Algorithm1 does not always solve *ProblemG*.

We can prove this by counter-example using the following as input:

Counter 1: n = 4, W = 10,
h_i = [5, 7, 8, 4]
w_i = [3, 2, 7, 3]

Algorithm1 outputs Solution S:

Solution S: $Platform_1$: [s_1, s_2]
$Platform_2$: [s_3, s_4]
Cost = 7 + 8 = 15

But a solution with a minimized cost would be:

Solution S': $Platform_1$: [s_1]
$Platform_2$: [s_2…s_4]
Cost = 5 + 8 = 13

Algorithm1 does not always solve *ProblemS2*.

We can prove this by counter-example using the following as input:

Counter 2: n = 6, W = 10,
h_i = [35, 15, 10, 5, 75, 85]
w_i = [3, 3, 2, 1, 1, 1]

Algorithm1 outputs Solution S:
Solution S: Platform$_1$: [s_1…s_5]
Platform$_2$: [s_6]
Cost = 75 + 85 = 160

But a solution with a minimized cost would be:
Solution S': Platform$_1$: [s_1…s_4]
Platform$_2$: [s_5, s_6]
Cost = 35 + 85 = 120

2.2. Algorithm 2

2.2.1 ProblemS2

Given the heights $h_1, \ldots, h_n$, where $\exists$ k such that $\forall$ i < j ≤ k, hi ≥ hj and $\forall$ k ≤ i < j, $h_i \leq h_j$, 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.

2.2.2 Θ(n) Time Greedy Algorithm Design

In this special case of the problem, the heights follow an unimodal function with a single local minimum. Therefore, we design a greedy strategy that takes advantage of this fact using a partitioning approach. We name this local minimum a “valley.”

The algorithm tries to partition the sculptures in parallel from two directions: from the front (starting at the beginning of the list) and from the back (starting at the end of the list), both aiming toward the valley. It works greedily by adding sculptures to a platform until the total width of sculptures on that platform exceeds the given width limit (W). When this happens, the current platform is finalized, and the next platform begins. Each traversal continues until the valley is reached. The sculptures between the two traversal points—the ones that have not yet been assigned to either the front or back platform—form what we refer to as the *ambiguous region*. This region is called ambiguous because it could potentially be assigned to either the front or back platforms, depending on which side offers a better fit based on the heights and available width.

Once the ambiguous region is identified, the algorithm decides how to allocate the sculptures in this region by comparing the heights at the boundaries of the front and back platforms. If the front platform has taller sculptures at its boundary, the algorithm extends the front platform to include more sculptures from the ambiguous region. Otherwise, the back platform takes these sculptures. This decision-making process ensures that the total height of all platforms is minimized.

Finally, the two sets of platforms (from the front and back) are merged.

Time Complexity

The time complexity of Algorithm2 is Θ(n), where n is the number of given sculptures.

To find the valley, the program iterates once over the list of heights, taking Θ(n) time to complete.

Then, it performs the loop of sculptures accessing both the front and the back of the list. Each traversal iterates through the list of sculptures up to the valley, and together, they cover the entire list of sculptures only once – taking Θ(n) time to complete.

The ambiguous region (if any) is processed by iterating through a subset of sculptures, which can range from 0 to *n* in the worst case. It doesn't add extra time beyond *n*.

In conclusion, the dominant operations in this algorithm all occur in linear time. Furthermore, the program always runs in linear time, meaning there is no situation where the time complexity drops below linear.

Therefore, Algorithm2 takes Θ(n) time to complete.

2.2.3 Proof of Correctness

We will prove the correctness of the algorithm using a proof by induction

Base Case (n = 1):

When n = 1, there is only one sculpture. The algorithm will place this single sculpture on a platform, and the total height will be the height of that sculpture. The width constraint is trivially satisfied, as a single sculpture will always fit on the platform.Thus, the algorithm is correct for n = 1.

Inductive Hypothesis:

Assume that the algorithm is correct for n = k sculptures. Specifically, it places the sculptures on platforms such that the total width on each platform does not exceed W and the total height is minimized by always grouping as many tall sculptures as possible on each platform.

Inductive Step (n = k+1):

Consider the case with n = k+1 sculptures. By the inductive hypothesis, the algorithm correctly processes the first k sculptures and places them optimally on platforms. Now, we need to show that adding the (k+1)-th sculpture maintains the correctness of the solution.

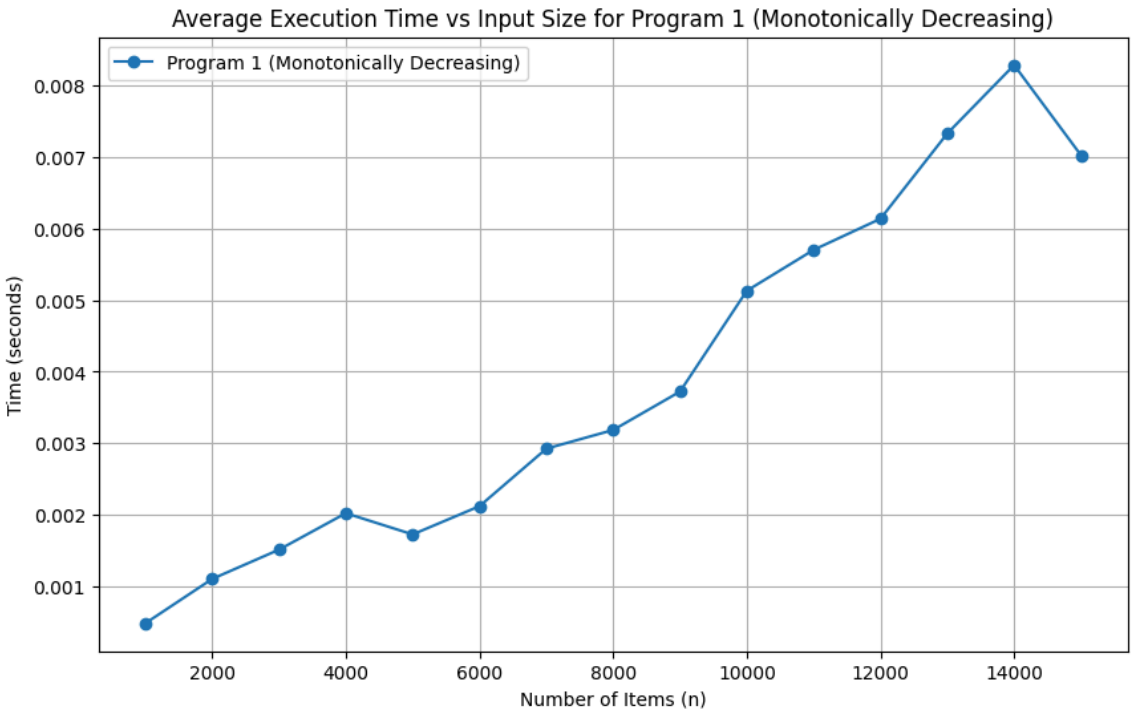
1. Process both ends at same time: The algorithm starts from the edges and attempts to group sculptures into a platform based on the width constraint W on each end. If the algorithm reaches the ambiguous region, it evaluates both ends of the region and

prioritizes the taller end to minimize the total height on subsequent platforms. Based on the induction hypothesis, the algorithm has already arranged the sculptures optimally for the first k sculptures. When adding the k+1th sculpture, it will be placed at the end of the heights sequence with a height greater or equal to k, maintaining this special case's constraint of an unimodal function structure. The greedy decision will select it first since it's the tallest on the right end and select as many as it can fit on its platform to ensure that the total height remains minimized. This decision guarantees that the platform arrangement is locally optimal because we maximize the number of "tall" sculptures per platform while trying to keep the tallest sculptures together from one end to minimize height.

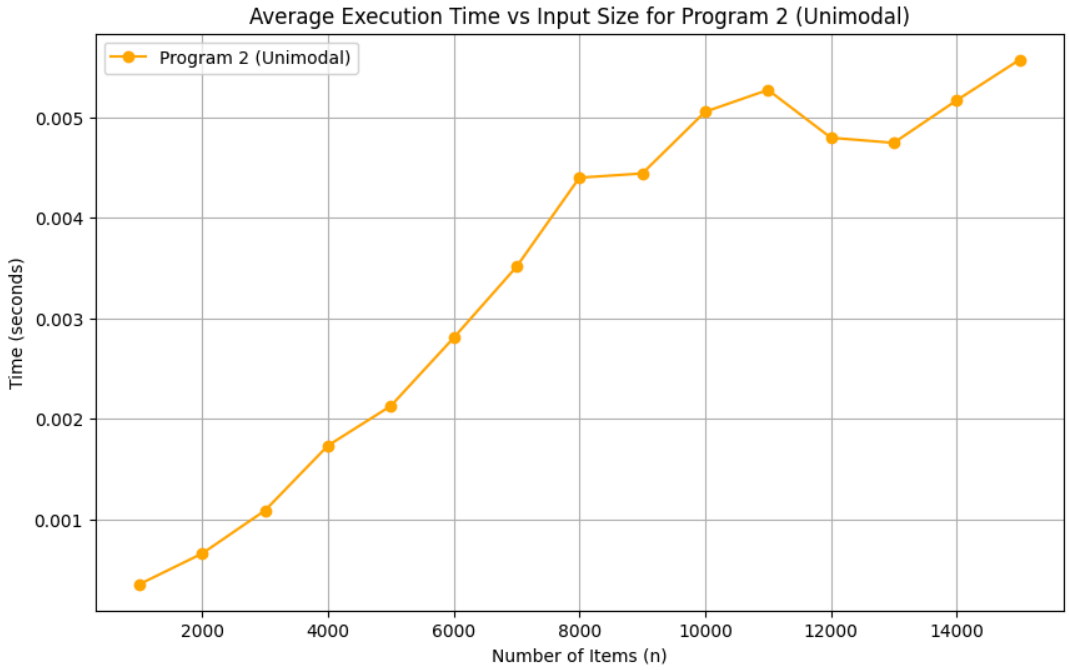
2. Handling the Ambiguous Region: In the ambiguous region around the valley (or at the valley point), we compare the heights of the sculptures on both ends of the region. The key observation is that the taller sculpture is prioritized because it dictates the overall height of that platform. The greedy choice to assign more sculptures to the side with the taller sculpture (as long as W is not exceeded) minimizes the total platform height, as placing sculptures on the shorter side would only unnecessarily increase the total height.

By induction, the algorithm correctly places all n sculptures on platforms such that the total width on each platform does not exceed W and the total height is minimized. Therefore, the algorithm is **correct**.

3. Experimental Comparative Study



Plot 1: Running time of Program1 with respect to varying input size



Plot 2: Running time of Program2 with respect to varying input size

Both programs show a steady general growth trend as the input increases. There are fluctuations, but the overall trend is an upward trajectory – which suggests the Θ(n) time complexity of both programs.

These fluctuations can possibly be attributed to the testing data, which are random and not evenly distributed. The impact of this might be more noticeable in *Program2* because its code is much more complex than that of *Program1*, with significantly more branch logic that could be more sensitive to this characteristic.

4. Conclusion

As a group, we found the project to be challenging and a good learning experience.

Program2 was significantly more challenging to implement than Program1. The latter was a more intuitive and straightforward case. Program2 took many days of discussions and brainstorming, and we worked with different implementations until we found the correct one. Our main problem was trying to find a way of grouping structures in the ambiguous region previously mentioned while maintaining the greedy strategy and defined time complexity. We also had to consider and write many more testing cases to account for edge scenarios, which was another challenge on its own. We did not encounter any technical difficulties as all three members were familiar with the chosen programming language (Python).

Proving the correctness of both algorithms was also challenging. Writing the formal proof is an intricate part of the class in general, but writing more test scenarios and performing the experimental comparative study with random inputs helped improve our confidence in the provided solutions.