COP4533-UFO
Algorithm Abstraction & Design
Programming Project
Milestone Two

Kyle Lund

Introduction

Milestone Two required the creation of one brute force and two dynamic programming algorithms to solve a variant of the popular bin-packing problem. This problem involves placing a series of sculptures while maintaining their order, onto a series of platforms. Each platform has a maximum width, and each sculpture has both a width, which takes up space on these platforms, and a height. Any resultant solution is judged based on the sum of the maximum sculpture heights on each platform used. In milestone one we solved specific cases of this problem suitable for greedy algorithms. With milestone two we will solve the general case of the problem as specified here:

Given the heights h1,...,hn and the base widths w1,...,wn 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.

Algorithm Design and Analysis – Algorithm Three

The third algorithm for this project and the first for this milestone solves the general case as stated in the introduction. This is a brute force algorithm that runs in $O(n2^{n-1})$ time. This algorithm iterates through every possible grouping of statues, determines whether this is a valid grouping based on the allowed platform width, calculates the cost of this grouping (the total of the maximum sculpture heights of each platform) and returns the best platform grouping.

Correctness is accomplished through brute force. Every possible valid combination of sculptures is processed, and the best solution is returned upon completion of the algorithm. As mentioned previously, this algorithm runs in $O(n2^{n-1})$ time. The number of possible groupings is a combinatorial problem, and there are 2^{n-1} possible groupings that will be iterated through. In every iteration, both the platform validity check and the calculation of the total cost of the platform grouping is accomplished in O(n) times. The final result is that each loop iteration takes O(n) time and there are $O(n2^{n-1})$ total iterations, for a final run time of $O(n2^{n-1})$.

Algorithm Design and Analysis – Algorithm Four

The 2^{nd} milestone two algorithm solves the same general case as the first, but in $O(n^3)$ time. First, it defines an array OPT, where OPT[i] represents the minimum total height needed to display sculptures 1 through i. To fill in this array, we consider all possible valid partitions of the sculptures from j to i (where j ranges from i to the beginning of the array), ensuring that the total width of sculptures from j to i fits within the platform's width W. Then, the algorithm proceeds back through the array and reconstructs the final solution.

Correctness is accomplished by utilizing optimal subproblems. The key idea for this algorithm is that the solution for arranging sculptures 1 through i can be built from the optimal solutions of subproblems for the sculptures 1 through j -1, where j ≤ i. To maintain correctness, the algorithm ensures that the total width of sculptures from j to i does not exceed the platform's width W. For each valid partition, the height of the platform is determined by the tallest sculpture on the platform. The recurrence relation updates OPT[i] as the minimum height of all possible sculpture combinations from the valid partitions as shown below:

$$OPT[i] = \min_{1 \le j \le i}(OPT[j-1] + \max(h_j, h_{j+1}, \dots, h_i)$$

The algorithm runs in $O(n^3)$ time complexity due to two nested loops, one for i and one for j for a run time of $O(n^2)$ for the iterations. The calculations occurring inside these loops run in O(n) time, resulting in a total run time of $O(n^3)$.

Algorithm Design and Analysis – Algorithm Five

The final milestone two algorithm is very similar to the previous algorithm but runs in $O(n^2)$ time. First, it defines an array OPT, where OPT[i] represents the minimum total height needed to display sculptures 1 through i. To fill in this array, we again consider all possible valid partitions of the sculptures from j to i (where j ranges from i to the beginning of the array), checking if the total width of sculptures from j to i fits within the platform's width W. Then, the algorithm proceeds back through the array and reconstructs the final solution. The main difference is that total widths and possible maximum heights of the various partitions have been precalculated, so that only O(1) operations are performed inside of the nested loops.

Correctness does not change from the previous algorithm and is accomplished by utilizing optimal subproblems. The key idea for this algorithm is that the solution for arranging sculptures 1 through i can be built from the optimal solutions of subproblems for the sculptures 1 through j -1, where j ≤ i. To maintain correctness, the algorithm ensures that the total width of sculptures from j to i does not exceed the platform's width W. For each valid partition, the height of the platform is determined by the tallest sculpture on the platform. The recurrence relation updates OPT[i] as the minimum of all possible heights from the valid partitions:

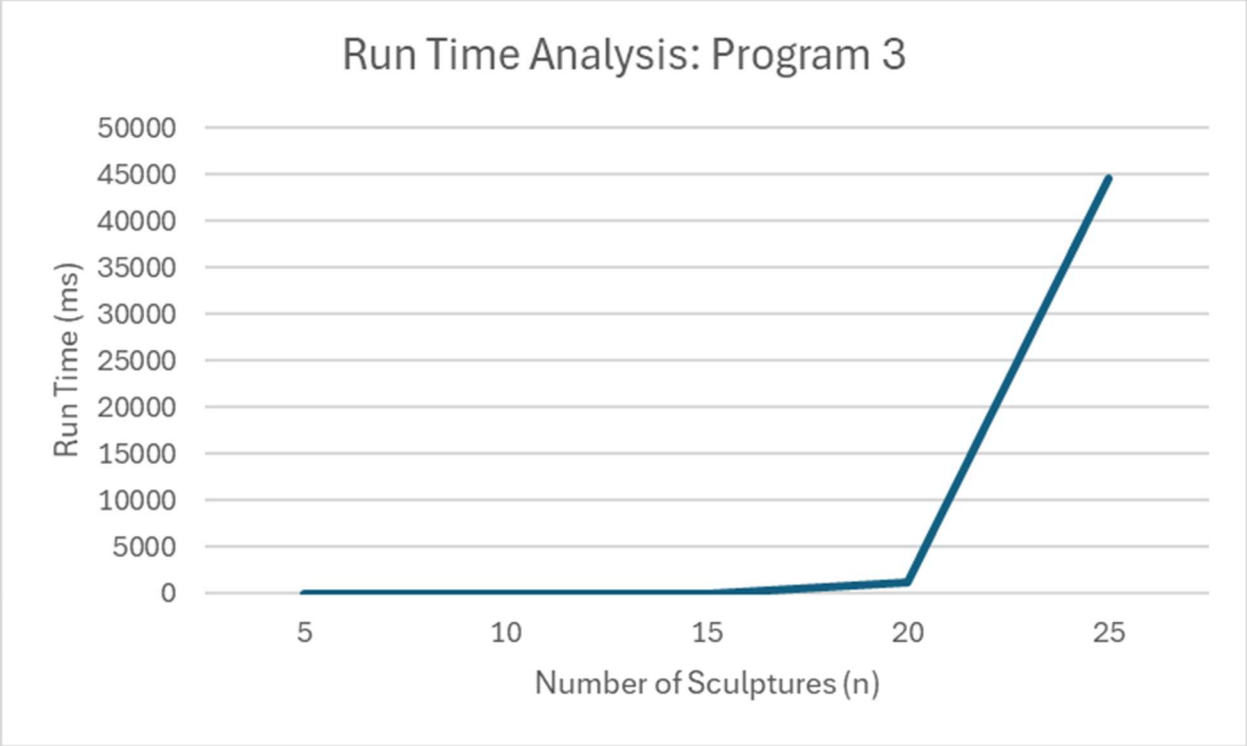
$$OPT[i] = \min_{1 \le j \le i}(OPT[j-1] + \max(h_j, h_{j+1}, \dots, h_i)$$

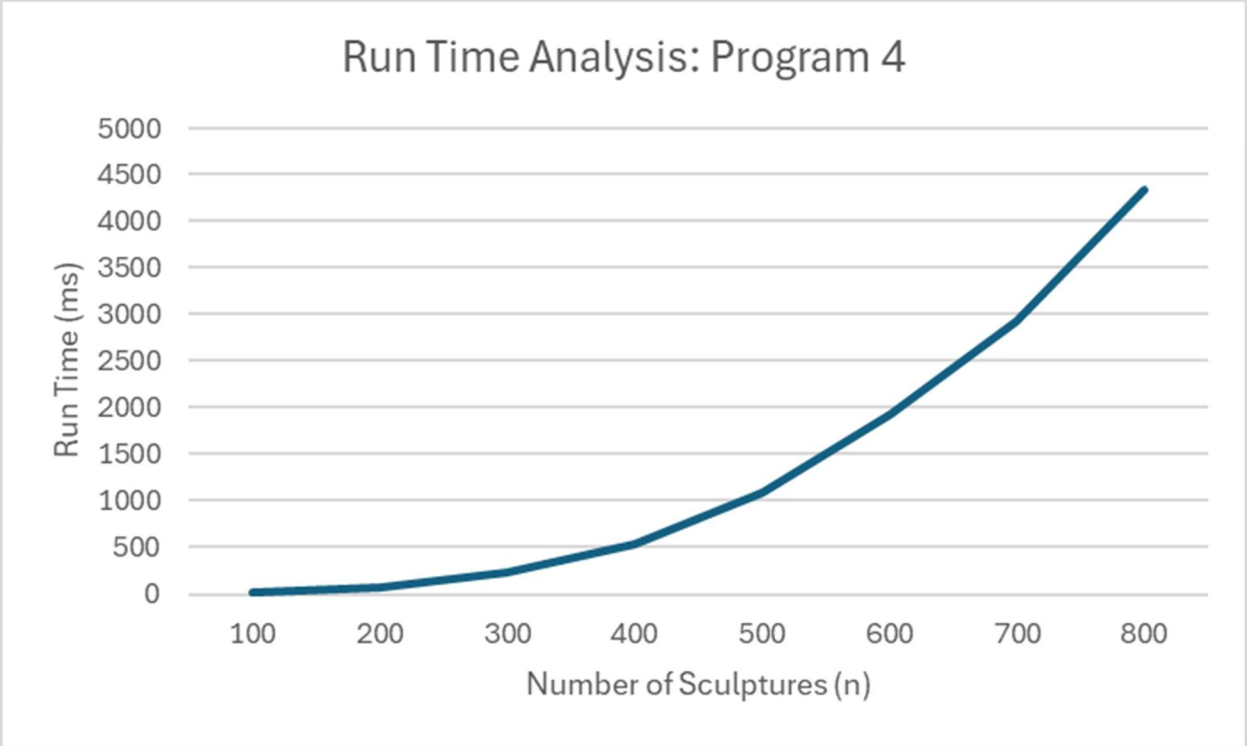
The algorithm runs in $O(n^2)$ time complexity due to two nested loops, one for i and one for j, for a run time of $O(n^2)$ for the iterations. The main difference from the previous algorithm is that inside of each of these loops we spend only O(1) time retrieving previously calculated platform widths and either updating the platform height or retrieving it from the precalculated values.

There are two variations of algorithm five. The first is a top-down, recursive implementation. This utilizes a sub-function in place of the outer loop described above to update the OPT array and determine the optimal results. The second variation is a bottom-up, iterative implementation, which is described above.

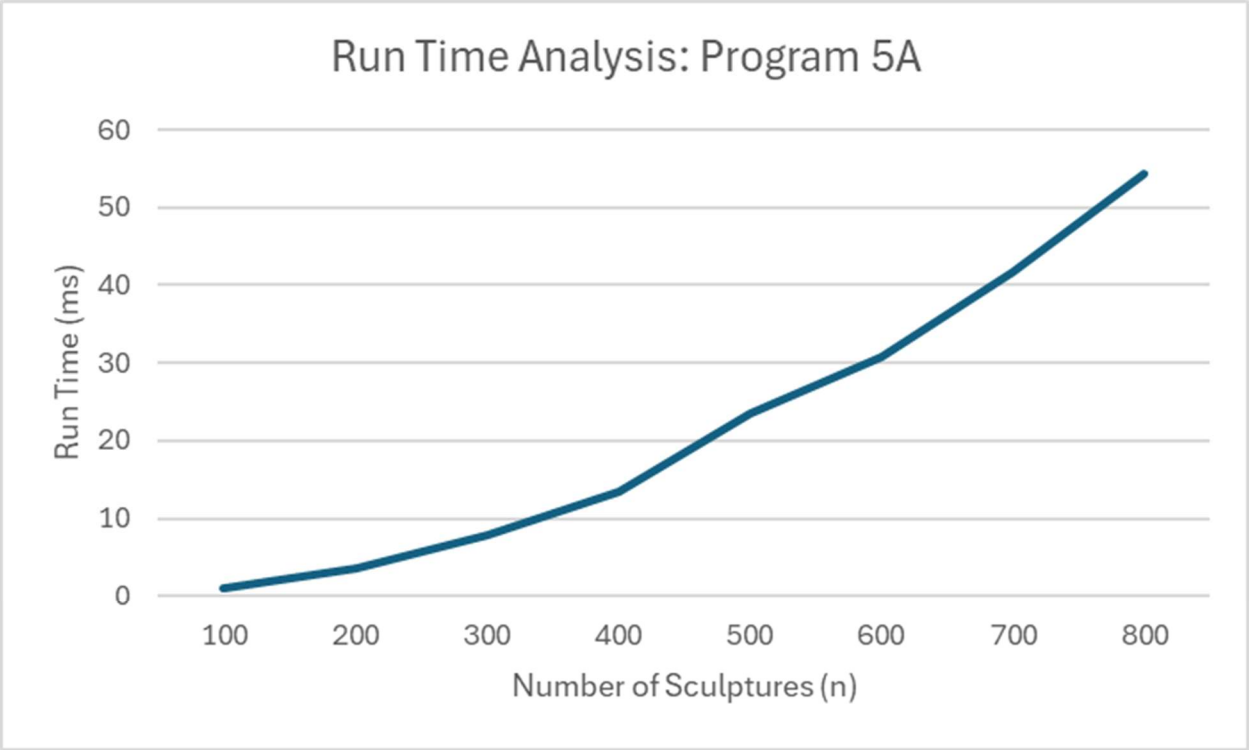
Experimental Study

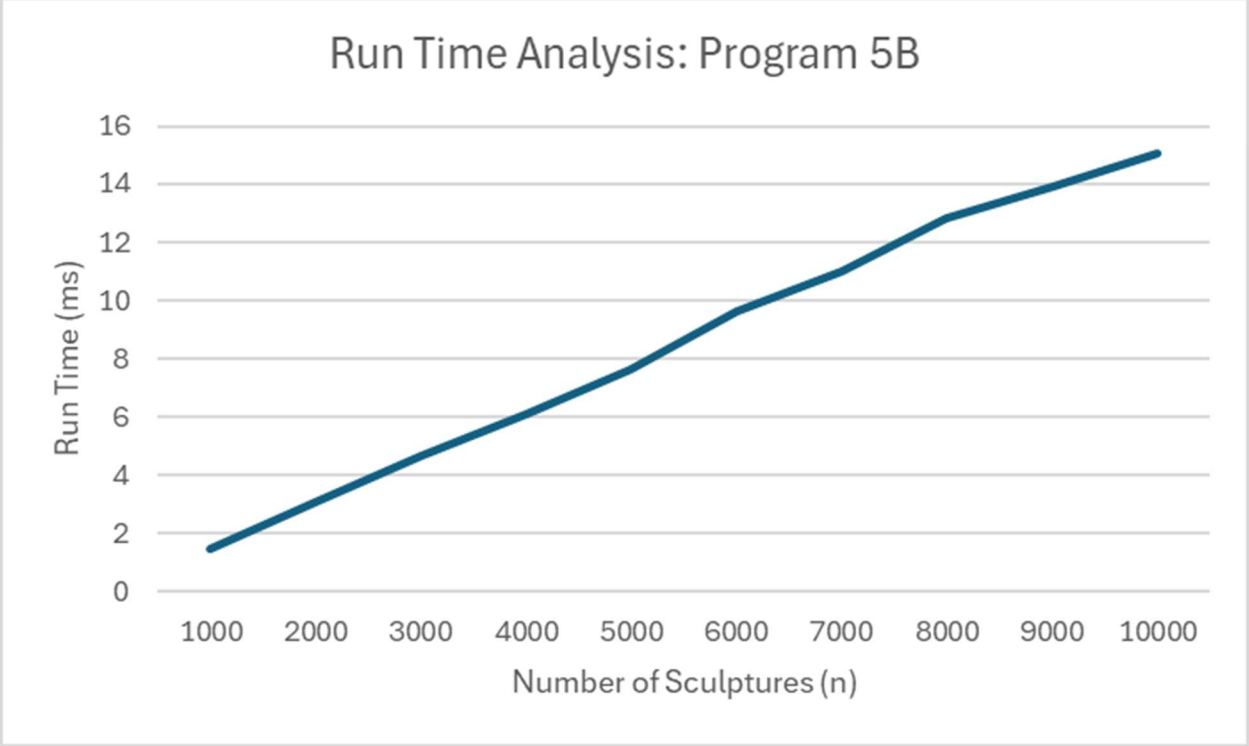
The graphs included below shows the results of the runtime analysis performed on the algorithms. Tests for program 3 were performed for n values of 5, 10, 15, 20, and 25. Values higher than this were unusable due to the exponentially increasing run times.





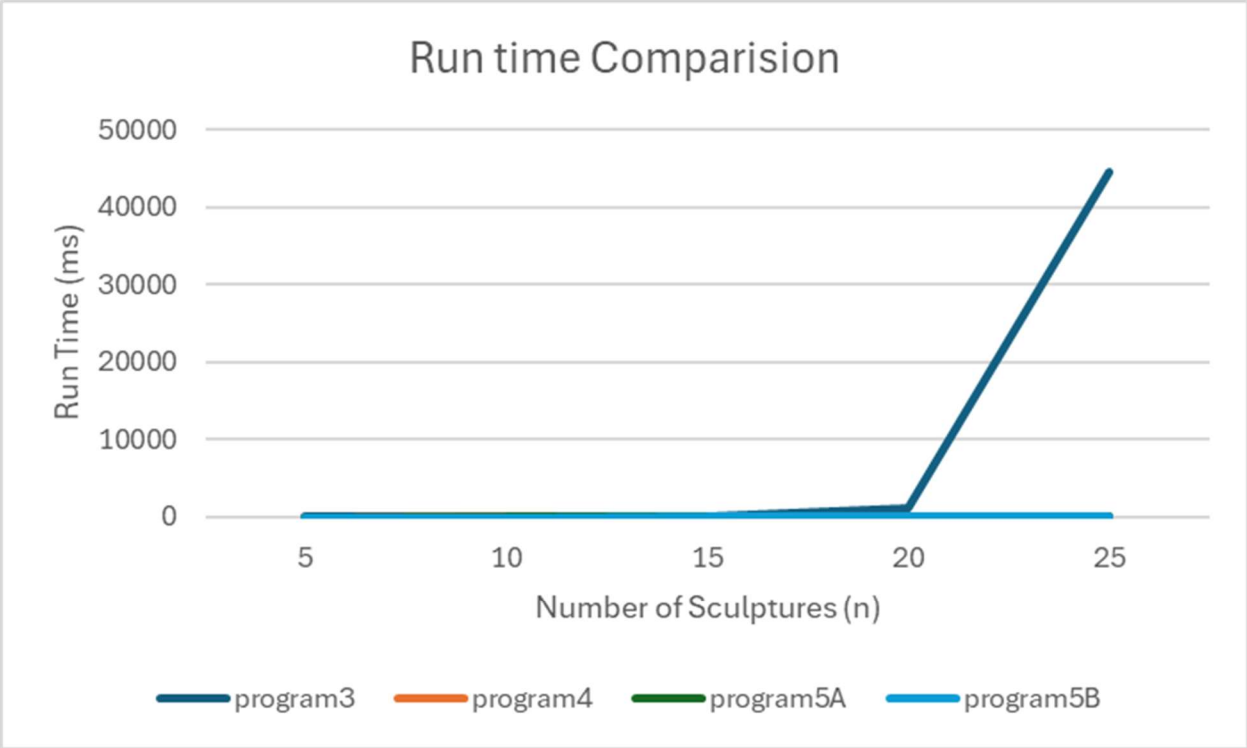
Tests were performed for programs 4, and 5A with n values of 100, 200, 300, 400, 500, 600, 700, 800. Values above this were unusable due to reaching the recursion limit.

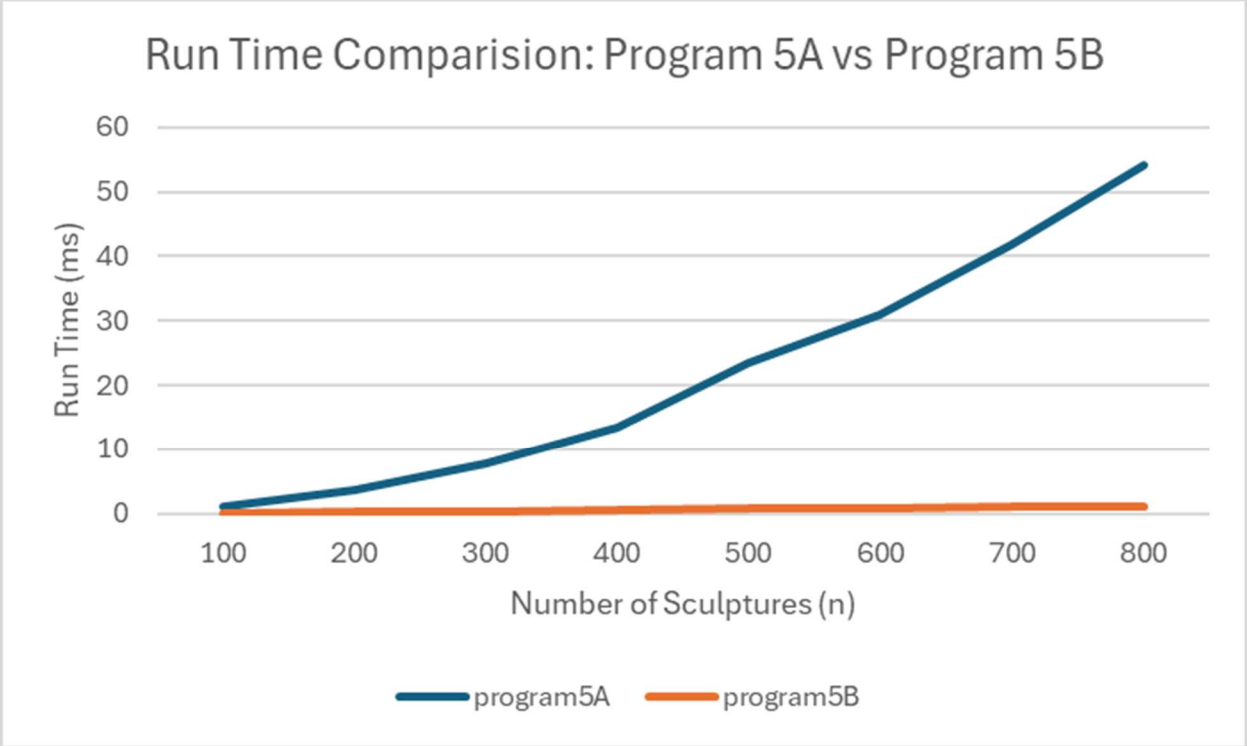




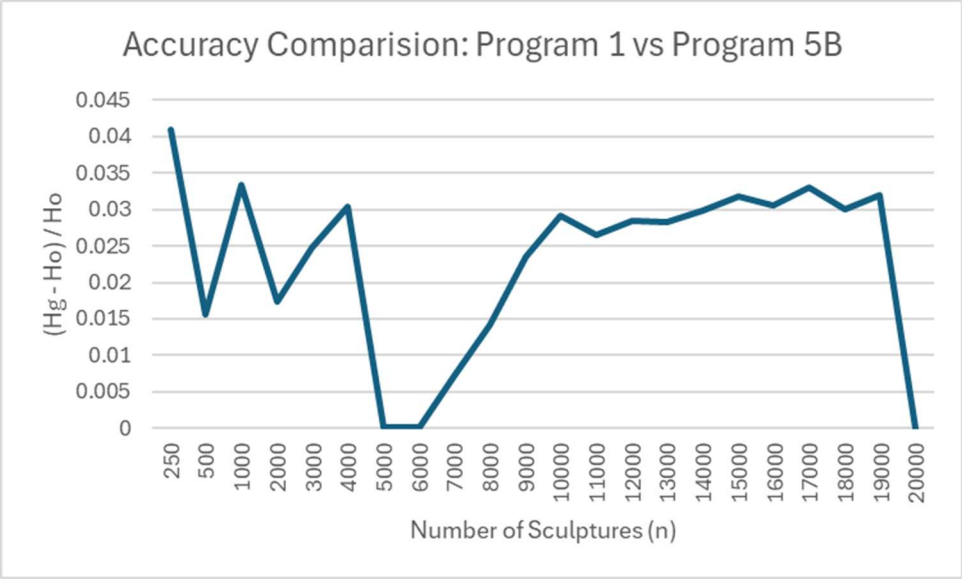
For all tests, lists of heights and widths were randomly generated and each program was run 10 times. The running times were then averaged to get the final result. One noticeable potential discrepancy is the difference in run times between program 5A and 5B. There are two potential contributors to this. First, while both are written to be $O(n^2)$ time complexity, the recursive calls and

time spent doing memory management, particularly when using python, pushed the run time of program 5A higher than expected. The second is that due to the way the two loops in algorithm 5B work, the inner loop is potentially much smaller than the outer loop, depending on the value of W. The overlayed charts, one run on the input that worked for program 3 comparing all four programs, and another comparing programs 5A and 5B are below.





There was less information to be gleamed from the results comparison to the greedy approach, as plotted below. The results were fairly random with respect to the input size and the average error represented around a 3% difference in reported optimal heights, though there were multiple cases with near 0 error. However, it should be noted that the overall error rate is, in my opinion, relatively low, given the simple implementation and speed up in run time gained by running the greedy algorithm.



Conclusion

In conclusion, the development and analysis of algorithms to address general case of the presented problem demonstrate the additional complexity, both algorithmically and run-time, involved in solving non-cooperative challenges. The first algorithm effectively demonstrates both the advantages, guaranteed success and relative simplicity of the algorithm, and the disadvantages, long run-times, associated with brute force solutions. However, its limitations highlight the necessity and opportunity for more advanced algorithm paradigms. The second algorithm offers us the first step in this direction, utilizing memorization and dynamic programming to solve the problem in a more efficient, though still non-optimal, manner. The final two algorithms take this solution to its final state, utilizing two different methods, top-down recursion and bottom-up iteration, of obtaining far faster run-time on the general case of this problem. Finally, the comparison to the previous greedy solution shows that even though an algorithm may be non-optimal, the quickly produced solution can be close enough to optimal in order to be useful.

Code

```
#Program 3
from typing import List, Tuple
from itertools import chain, combinations
from math import inf


def all_combinations(indexes):
    for n in range(1, len(indexes)):
        for splits in combinations(range(1, len(indexes)), n):
```

```
            result = []
            prev = None
            for split in chain(splits, [None]):
                result.append(indexes[prev:split])
                prev = split
            yield result


def is_valid_platform(widths, platforms, W):
    for platform in platforms:
        if sum([widths[i] for i in platform]) > W:
            return False
    return True


def calc_height(heights, platforms):
    height_platforms = [[heights[i1] for i1 in platform] for platform in
platforms]
    return sum([max(platform) for platform in height_platforms])


def program3(
    n: int, W: int, heights: List[int], widths: List[int]
) -> Tuple[int, int, List[int]]:
    """
    Solution to Program 3

    Parameters:
    n (int): number of sculptures
    W (int): width of the platform
    heights (List[int]): heights of the sculptures
    widths (List[int]): widths of the sculptures

    Returns:
    int: number of platforms used
    int: optimal total height
    List[int]: number of statues on each platform
    """

    best_height = inf
    best_platforms = None

    # Check every permutation of statues
    for platforms in all_combinations(list(range(n))):
        # Check if valid
```

```
        if is_valid_platform(widths, platforms, W):
            # Check if best
            height = calc_height(heights, platforms)
            if height < best_height:
                best_height = height
                best_platforms = platforms

    return (
        len(best_platforms),
        best_height,
        [len(platform) for platform in best_platforms],
    )
```

```
#Program 4
from typing import List, Tuple


def program4(
    n: int, W: int, heights: List[int], widths: List[int]
) -> Tuple[int, int, List[int]]:
    """
    Solution to Program 4

    Parameters:
    n (int): number of sculptures
    W (int): width of the platform
    heights (List[int]): heights of the sculptures
    widths (List[int]): widths of the sculptures

    Returns:
    int: number of platforms used
    int: optimal total height
    List[int]: number of statues on each platform
    """

    # Initialize memo and set base case to 0
    memo = [float("inf")] * (n + 1)
    memo[0] = 0

    # Track the solution for reconstructing the platform allocation
    partition = [-1] * (n + 1)

    # DP loop
    for i in range(1, n + 1):
        for j in range(i, 0, -1):
```

```
            total_width = sum(widths[index] for index in range(j - 1, i))
            if total_width <= W:
                max_height_on_platform = max(heights[k] for k in range(j - 1, i))
                if memo[j - 1] + max_height_on_platform < memo[i]:
                    memo[i] = memo[j - 1] + max_height_on_platform
                    partition[i] = j - 1

    # Reconstruct solution
    platforms = []
    sculpture_indices = n
    while sculpture_indices > 0:
        prev_sculpture_index = partition[sculpture_indices]
        platforms.append(sculpture_indices - prev_sculpture_index)
        sculpture_indices = prev_sculpture_index

    platforms.reverse()

    return len(platforms), memo[n], platforms
```

```
#Program 5A
from typing import List, Tuple


def program5A(
    n: int, W: int, heights: List[int], widths: List[int]
) -> Tuple[int, int, List[int]]:
    """
    Solution to Program 5A

    Parameters:
    n (int): number of sculptures
    W (int): width of the platform
    heights (List[int]): heights of the sculptures
    widths (List[int]): widths of the sculptures

    Returns:
    int: number of platforms used
    int: optimal total height
    List[int]: number of statues on each platform
    """

    # Initialize memo and set base case to 0
    memo = [float("inf")] * (n + 1)
    memo[0] = 0
```

```
    # Precalculate running sums for width
    width_sum = [0] * (n + 1)
    for i in range(1, n + 1):
        width_sum[i] = width_sum[i - 1] + widths[i - 1]

    # Recursive call
    def dp(i):
        # If already computed
        if i in memo:
            return memo[i]

        # Base case
        if i == 0:
            return 0

        # Explore partitions
        min_height = float("inf")

        # Try all partitions from j to i
        max_height = 0
        for j in range(i, 0, -1):
            total_width = width_sum[i] - width_sum[j - 1]

            # Break if doesn't fit
            if total_width > W:
                break

            # Update max height
            max_height = max(max_height, heights[j - 1])

            # Recursively calculate height for partition
            height_for_this_partition = dp(j - 1) + max_height

            # Minimize the total height
            min_height = min(min_height, height_for_this_partition)

        # Update memo and return
        memo[i] = min_height
        return min_height

    # Get recursion started
    total_height = dp(n)

    # Reconstruct solution
    platforms = []
```

```
    i = n
    while i > 0:
        for j in range(i, 0, -1):
            total_width = width_sum[i] - width_sum[j - 1]
            if total_width <= W and dp(i) == dp(j - 1) + max(heights[j - 1 : i]):
                platforms.append(i - j + 1)
                i = j - 1
                break

    platforms.reverse()

    return len(platforms), total_height, platforms
```

```
#Program 5B
from typing import List, Tuple


def program5B(
    n: int, W: int, heights: List[int], widths: List[int]
) -> Tuple[int, int, List[int]]:
    """
    Solution to Program 5B

    Parameters:
    n (int): number of sculptures
    W (int): width of the platform
    heights (List[int]): heights of the sculptures
    widths (List[int]): widths of the sculptures

    Returns:
    int: number of platforms used
    int: optimal total height
    List[int]: number of statues on each platform
    """

    # Initialize memo and set base case to 0
    memo = [float("inf")] * (n + 1)
    memo[0] = 0

    # Precalculate running sums for width
    width_sum = [0] * (n + 1)
    for i in range(1, n + 1):
        width_sum[i] = width_sum[i - 1] + widths[i - 1]

    # DP loop
```

```
    for i in range(1, n + 1):
        max_height = 0
        for j in range(i, 0, -1):
            total_width = width_sum[i] - width_sum[j - 1]
            if total_width <= W:
                # For partition j to i, update the maximum height and memo
                max_height = max(max_height, heights[j - 1])
                memo[i] = min(memo[i], memo[j - 1] + max_height)
            else:
                break

    # Reconstruct solution
    platforms = []
    i = n
    while i > 0:
        for j in range(i, 0, -1):
            total_width = width_sum[i] - width_sum[j - 1]
            if total_width <= W and memo[i] == memo[j - 1] + max(heights[j - 1 :
i]):
                platforms.append(i - j + 1)
                i = j - 1
                break

    platforms.reverse()

    return len(platforms), memo[n], platforms
```