COP4533-UFO
Algorithm Abstraction & Design
Programming Project
Milestone One

Kyle Lund

Introduction

Milestone One required the creation of two greedy algorithms to solve variants 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. This first algorithm is designed to solve the narrow case where sculptures are in decreasing order based on their height. The second algorithm is designed to solve a case where the sculpture heights are unimodal. Meaning that one sculpture in the series serves as a local minimum, where the heights of the sculptures prior to this one are decreasing, and the heights of the sculptures afterwards are increasing. Both algorithms use a greedy approach and run in O(N) time.

Algorithm Design and Analysis – Algorithm One

The first algorithm solves the case where the sculptures are presented in a decreasing series based on their height, as stated here:

> *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 minimizes the total height.*

This has important implications for the design of the algorithm. This begins by placing the first sculpture on a platform, and then looks at every sculpture in order, and at every step decides between two possible outcomes. The first option is to place the sculpture under consideration on the same platform as the previous sculpture. This option is taken whenever there is room on the platform for the new sculpture. Whenever there is no room on the platform, the second option is taken, and the sculpture gets placed on a new platform.

Proof of correctness is derived via induction. In the base case with a single sculpture, this sculpture is placed on its own platform and this solution is optimal. In the inductive case, at each step the algorithm can make one of two choices. If the sculpture under consideration can fit on the same platform as the previous sculpture, it is placed there, and the final cost of the solution is unchanged. This is guaranteed due to the decreasing height of the sculptures as they are presented to the algorithm. If the sculpture cannot fit on the same platform as the previous sculpture, it is placed on a new sculpture. The final cost of the algorithm will increase in this case, but as it is not possible to place the sculpture elsewhere due to the requirement that the sculptures remain in order, this remains the optimal solution. Therefore, at each step the algorithm makes the optimal choice which results in outputting the optimal solution.

This algorithm is a narrow one, designed only to solve the decreasing case. It cannot solve the general case of the algorithm, where there is no strict ordering of the input, as stated here:

> *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.*

For proof via counterexample, consider an input where the sculpture heights $h_1,\ldots,h_n$ are [1, 4, 5], the sculpture widths $w_1,\ldots,w_n$ are [4, 5, 2], and the maximum platform width is 10. The algorithm as stated above will place the first two sculptures on the first platform, with a maximum height of 4, and the last sculpture on its own platform, with a height of 5. This gives a solution with a total height of 9. However, the optimal solution would put the first sculpture on its own platform and the last two on their own platform. This would result in an optimal solution of 6

This algorithm is also incapable of solving a different narrow case, which is solved by the second algorithm, as stated here:

> *Given the heights $h_1,\ldots,h_n$, where $\exists k$ such that $\forall i < j \leq k$, $h_i \geq h_j$ and $\forall k \leq i < j$, $h_i \leq h_j$, 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.*

For proof via counterexample, consider an input where the sculpture heights $h_1,\ldots,h_n$ are [5, 3, 1, 4, 8], the sculpture widths $w_1,\ldots,w_n$ are [5, 4, 3, 6, 3], and the maximum platform width is 10. The algorithm as stated above will place the first two sculptures on the first platform, with a maximum height of 5, the next two sculptures on the second platform with a maximum height of 4, and the last sculpture on its own platform with a height of 8. This results in a solution with a total cost of 17. The optimal solution, however, is one where the middle sculpture is on its own platform, and the last two sculptures are on the same platform. This gives a solution with three platforms which have heights of 5, 1, and 8, for a total cost of 14.

Algorithm Design and Analysis – Algorithm Two

The second algorithm solves the case where the sculptures are presented in a unimodal order, where the sculptures are decreasing until a sculpture with a minimum height is processed, then from this point the sculpture heights are increasing, as stated here:

> *Given the heights $h_1,\ldots,h_n$, where $\exists k$ such that $\forall i < j \leq k$, $h_i \geq h_j$ and $\forall k \leq i < j$, $h_i \leq h_j$, 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.*

This algorithm begins the same way as the previous one, greedily assigning sculptures to platforms where it can, and adding them to new platforms when they do not fit onto the current platform. This algorithm diverges from the previous one once the sculpture heights begin to increase. At this point, the algorithm begins to process the remaining sculptures starting from the end of the list towards the previously found local minimum sculpture. This effectively transforms what would have been an input with increasing heights back to one with decreasing heights. Then, once all sculptures have been processed, the algorithm determines whether it can combine the last platform from the

forward looking first half of the algorithm, with the first platform from the backwards looking second half of the algorithm.

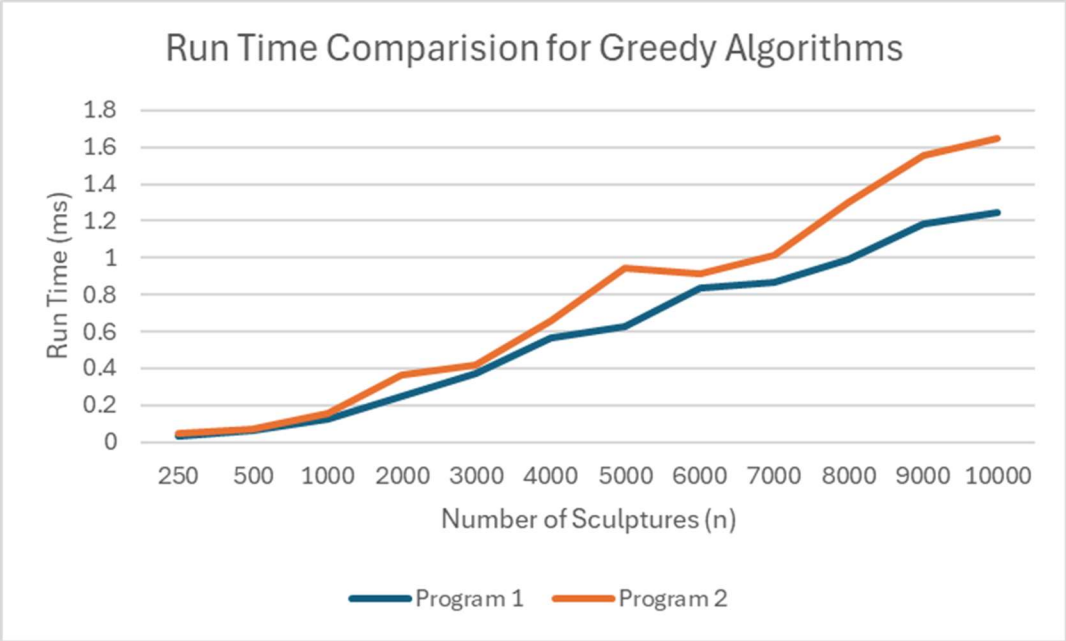
Proof of correctness is derived from the proof for the first algorithm, which proved the correctness of the greedy approach for an input with a decreasing height. The forward looking first half of the algorithm is exactly the same as the algorithm proved to be correct previously. Once the local minimum is discovered in the input, the algorithm begins its backward looking second half. This algorithm again processes the input in order of decreasing height, and the greedy approach to this problem has already been shown to be correct. The only new component of this algorithm is the combine step, where the solutions of the two steps are combined. There are only two theoretical options for this step:

Case 1: the last platform generated by the first half can be combined with the first platform generated by the second half. In this case, the algorithm combines the two platforms producing the optimal output.
Case 2: the last platform generated by the first half cannot be combined with the first platform generated by the second half. In this case, there is nothing the algorithm can do, and the solution is already optimal.

In both cases, it may be possible to shift sculptures which occur after the minimal sculpture forward, however, this will either have no effect or will increase the total cost. Because the sculptures at this point in the input are increasing in height, the sculpture under consideration for moving to a previous platform will either have the same height as the previous sculpture, in which case the total cost of the solution will remain unchanged, or the height of the sculpture is higher, in which case the total cost of the solution will increase. In either case, moving a sculpture forward is never optimal.

Experimental Study

The graph included below shows the results of the runtime analysis performed on the two algorithms.



The testing code is included below. Tests were performed for n values of 250, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, and 10000. For each test, lists of heights and widths were randomly generated, and the heights list was altered to be unimodal for testing of program two. Each input was run 10 times, and the runtimes were averaged to get the final result. Variations in the runtime occurred due to the nature of the timing module used and its inability to account for process switching. The results show a clear linear runtime for both algorithms, with algorithm two having a slightly higher constant multiplier due to the additional work involved in the combining step.

Conclusion

In conclusion, the development and analysis of two greedy algorithms to address specific variants of the bin-packing problem demonstrate the effectiveness of tailored approaches in solving constrained optimization challenges. The first algorithm efficiently handles sculptures in a strictly decreasing height order, ensuring optimal packing while minimizing total height on display platforms. However, its limitations highlight the necessity for specialized solutions in varying scenarios. The second algorithm extends the framework to unimodal height arrangements, successfully incorporating a dual-pass strategy that capitalizes on both decreasing and increasing sequences, ultimately leading to improved solutions. The empirical results affirm that both algorithms operate within a linear time complexity, with the second algorithm introducing slight overhead due to its more complex operations. This exercise offers useful insights into designing algorithms for the bin-packing problem and opens the door for more general solutions that can be used in other situations.

Code

```
def program1(
    n: int, W: int, heights: List[int], widths: List[int]
) -> Tuple[int, int, List[int]]:
    """
    Solution to Program 1

    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 loop parameters
    curr_width = widths[0]
    total_height = heights[0]
    platforms = [1]
```

```
    # Loop through all scultpures
    for index in range(1, n):
        # If can fit on current platform
        if curr_width + widths[index] <= W:
            # Add to current platform
            curr_width += widths[index]
            platforms[-1] += 1

        # If can't fit on current platform
        else:
            # Initialize new platform
            curr_width = widths[index]
            total_height += heights[index]
            platforms.append(1)

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

```
def program2(
    n: int, W: int, heights: List[int], widths: List[int]
) -> Tuple[int, int, List[int]]:
    """
    Solution to Program 2

    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 loop parameters
    curr_width = widths[0]
    platform_heights = [heights[0]]
    platforms = [1]
    unimodal = False

    # Loop through all scultpures
    for index in range(1, n):
        # If minima passed
```

```
        if heights[index] > heights[index - 1]:
            # Start part two
            unimodal = True
            break

        # If can fit on current platform
        if curr_width + widths[index] <= W:
            # Add to current platform
            curr_width += widths[index]
            platforms[-1] += 1

        # If can't fit on current platform
        else:
            # Initialize new platform
            curr_width = widths[index]
            platform_heights.append(heights[index])
            platforms.append(1)

    # Part two, start from end of input and go backwards
    if unimodal:
        # Initialize loop parameters
        reverse_curr_width = widths[-1]
        reverse_platform_heights = [heights[-1]]
        reverse_platforms = [1]

        # Loop through remaining unplaced scultpures, from the back
        for reverse_index in range(-2, index - n - 1, -1):
            # If can fit on current platform
            if reverse_curr_width + widths[reverse_index] <= W:
                # Add to current platform
                reverse_curr_width += widths[reverse_index]
                reverse_platforms[-1] += 1

            # If can't fit on current platform
            else:
                # Initialize new platform
                reverse_curr_width = widths[reverse_index]
                reverse_platform_heights.append(heights[reverse_index])
                reverse_platforms.append(1)

        # Reverse reverse_platforms (will now be in front to back order)
        reverse_platforms.reverse()

        # Check if last normal and first reverse platforms can be combined
        # AKA, the last platform on the forward part and the first platform
```

```
        # on the now reversed part can fit onto a single platform
        if curr_width + reverse_curr_width <= W:
            # Update platform count
            platforms[-1] += reverse_platforms[0]
            # Keep max height
            platform_heights[-1] = reverse_platform_heights[0]
            # Delete combined platform
            del reverse_platforms[0]
            del reverse_platform_heights[0]

        # Combine
        platforms.extend(reverse_platforms)
        platform_heights.extend(reverse_platform_heights)

    return len(platforms), sum(platform_heights), platforms
```

```
from timeit import Timer
from random import randint, choice

from program1 import program1
from program2 import program2

W = 10
Ns = [250, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]
REPEATS = 10

for n in Ns:
    # Initialize variables
    heights = set()
    while len(heights) < n:
        heights.add(randint(1, n * 5))

    heights = sorted(list(heights))
    heights.reverse()
    widths = [randint(1, W) for _ in range(n)]

    # Run programs
    print(
        f"Program 1, N={n}: {Timer(lambda: program1(n, W, heights,
widths)).timeit(REPEATS) * 1000 / REPEATS} ms"
    )
    # Rearrange heights to be unimodal
    minima = choice(range(int(n / 20), int(n * 19 / 20)))
    heights[minima:] = heights[n - 1 : minima : -1]
    print(
```

```
        f"Program 2, N={n}: {Timer(lambda: program2(n, W, heights,
widths)).timeit(REPEATS) * 1000 / REPEATS} ms"
    )
```