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 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(n2n-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 is returned upon completion of the algorithm. As mentioned previously, this algorithm runs in O(n2n-1) time. The number of possible groupings is a combinatorial problem, and there are 2n-1 possible groupings that will be iterated through. In every iteration, both the platform validity check and 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(n2n-1) total iterations, for a final run time of O(n2n-1).

Algorithm Design and Analysis – Algorithm Four

The 2nd milestone two algorithm solves the same general case as the first, but in O(n3) 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 partitions of the sculptures from

j to i (where sculptures j to i are placed on the same platform), 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.

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. Specifically, for each i, the algorithm considers all valid partitions of sculptures into groups from j to i, where sculptures j through i are placed on a single platform. 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:

The algorithm runs in O(n3) time complexity due to two nested loops, one for i and one for j, both of which run in O(n) time for a total of O(n2) iterations. And inside of each of these loops we spend O(n) time computing platform width and height.

Algorithm Design and Analysis – Algorithm Five

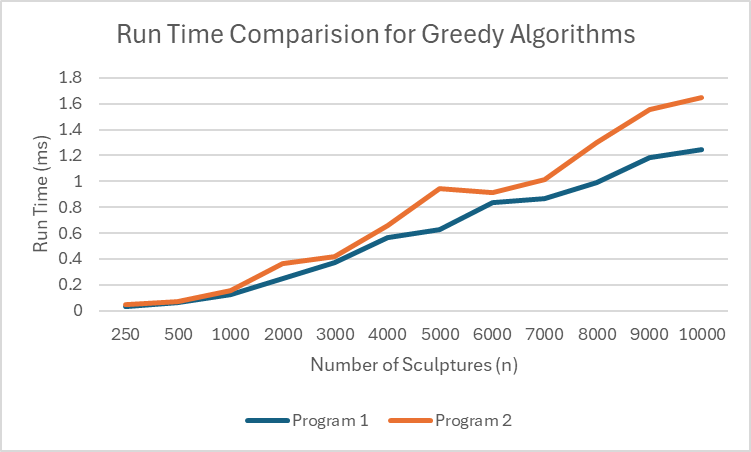
The final milestone two algorithm is very similar to the previous algorithm but runs in O(n3) time. First, it defines an array OPT, where OPT[i] represents the minimum total height needed to display sculptures 1 through i. to fill in this array, we consider all possible partitions of the sculptures from

j to i (where sculptures j to i are placed on the same platform), 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 of various partitions have been precalculated, so that only O(1) operations are performed inside of each loop.

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. Specifically, for each i, the algorithm considers all valid partitions of sculptures into groups from j to i, where sculptures j through i are placed on a single platform. 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:  
The algorithm runs in O(n2) time complexity due to two nested loops, one for i and one for j, both of Which run in O(n) time for a total of O(n2) 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 width and updating the platform height.

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 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 to 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"

    )