Group 3 - Milestone 2: COP4233 - Algorithms Abstraction and Design Fall '24

Elijah Johnson

- Repository, Code Standards, Report Structure and Documentation
- Algorithm 3 Backtracking, Time Complexity Analysis, Implementation
- Algorithm 5 Design, Pseudocode, Backtracking, Time Complexity Analysis, Correctness Analysis
- Testing Implementations, Graphs, Experimental Study Report
- Gradescope Submissions

Patrick Kallenbach

- Algorithm 3 Design, Correctness Analysis
- Algorithm 5A, 5B Implementations
- Conclusion

Nicholas Lindner

- Algorithm 4 Design, Backtracking, Time Complexity Analysis, Correctness Analysis, Implementation
- Conclusion

Algorithm and Analysis 3

Problem G

Given the heights $h_1, \cdots, h_n$ and the base widths $w_1, \cdots, w_n$ 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.

Algorithm 3: Naive Solution

Definition

$OPT(j)$ = The minimized *cost* of arranging j paintings into rows, where j is the *rightmost* painting being examined. In this formulation, $cost = \sum_1^j (\max_{h_k} r_j)$, and h_k is the height of a painting $p_k \in r_j$.

Goal

$OPT(n)$ = the minimized cost of arranging n paintings into rows.

Computing $OPT(j)$

Let C_{ij}, where $1 \leq i \leq j$, represent the cost of a row consisting of all paintings between and including paintings p_i and p_j. We allow i and j to be equivalent because it is possible that a row may have only one painting in an optimum solution. C_{ij} is represented by:

$$C_{ij} = \begin{cases} \infty & \text{if } \sum_{k=i}^{j} w_k > W \\ \max_{i \leq k \leq j}(h_k) & \text{otherwise} \end{cases}$$

To compute $OPT(j)$, we take the minimum sum over i for $1 \leq i \leq j$ of C_{ij} and $OPT(i-1)$. In this case, $i-1$ represents the *last* painting in the row preceding painting p_i. The following Bellman Equation describes $OPT(j)$:

$$OPT(j) = \begin{cases} 0 & \text{if } j < 1 \\ \min_{1 \leq i \leq j}(C_{ij} + OPT(i-1)) & \text{otherwise} \end{cases}$$

Backtracking to Determine Painting Attendance

To determine which paintings are present in a given row is not complex, but requires careful attention. At each OPT step, one must find the difference $j - i + 1$, which represents the length of the now-calculated row. This length will be appended to the end of a row-tracking list, as in our C++ implementation. This way, as recursive calls return upward, the list is generated from the front of the list of paintings to the back. Afterward, one can simply examine the list of paintings, counting the corresponding number for a given row to determine which paintings belong in that row.

Analysis

Time Complexity

We begin by examining the calculation of all C_{ij} values. Beginning at $j = n$ and $i = 1$, with j proceeding backward and i proceeding forward, to and including j in each iteration, we end up with a number of comparisons between painting heights totaling the following: $\Theta(n + (n-1) + (n-2) + \cdots + 1) = \Theta(\frac{n(n+1)}{2}) = \Theta(n^2)$.

Algorithm 3 recursively chooses the index i for which the sum $C_{ij}+OPT(i-1)$ is minimized. Given that this index may be arbitrary and all paintings may, in the worst case, optimally belong on each of their own rows, the number of recursive calls can be quite high. However, as soon as a painting is placed in a row, the number of remaining paintings must decrease by at least 1. Thus, there are $O(2^{n-1})$ possibilites at each recursive step, followed by an $O(n)$ comparison to determine which produces the minimum cost. In total, we have $O(n2^{n-1}+n^2) = O(n2^{n-1})$.

This solution does not memoize or store optimal values from previously-computed subproblem solutions, so it can be no slower than the C_{ij} computation time added to the minimum time needed to recursively calculate all possible values. All possible values must be considered and compared in order to compute an optimum solution, so we arrive at a total complexity of $\Theta(n2^{n-1})$.

Correctness

We begin by establishing the following assumption, that no painting is, by itself, too wide to fit on a platform:

$$\forall 1 \leq k \leq n, \nexists p_k \text{ where } w_k > W.$$

We will now examine the construction of C_{ij} values. These values represent, at index $C[i][j]$, the cost of a row containing paintings $[p_i, \cdots, p_j]$ for $1 \leq i \leq j$, calculated as the height of the tallest painting on the platform. Thus, any value of C_{ij} can be used to determine the relative *worth* of row $[p_i, \cdots, p_j]$. These values are used to determine the best combination of rows, since rows that cannot exist are marked as $C_{ij} = \infty$ and will therefore never be chosen over a smaller value. We now establish the following invariant:

I1: At the end of each iteration of the inner loop of C_{ij} computation, $C[i][j]$ correctly stores the maximum height of a row containing paintings p_i to p_j, as long as the combined widths of the paintings do not exceed W.

From our initial assumption, we also establish the second invariant:

I2: There always exists an arrangement of n paintings $p_1, \cdots, p_n$ into rows, as every painting is permitted to sit on its own row, whose cost will be $C[k][k] = h_k, \forall 1 \leq k \leq n$.

Following the computation of all C_{ij}, OPT(n) is called. This algorithm is recursive, solving smaller subproblems of the original input. We will use induction to prove the correctness of Algorithm 3.

Proof: For OPT(j) where $j = 1$, the case is simple. A set of one painting has only itself to be placed in a row, so the maximum height of the row is $C[1][1]+0 = h_1$, that of the singular painting, following and upholding I2.

Moving forward, when $j > 1$, OPT(j) computes the value of the minimum cost over i for $C_{ij} + \text{OPT}(i-1)$. This step will recursively calculate the minimum possible cost of a row being created whose rightmost painting is p_j, and calculate the cost of the next row that begins at p_{i-1}. The value of OPT(j) is minimized at every recursive step using solutions to smaller subproblems, each of which is calculated without changing any C values and therefore upholding I1. By using the optimal substructure property of Problem G, Algorithm 3 properly constructs a correct solution, outputting the minimized cost of a set of paintings

by finding the minimum cost of a row, over all possible rows. This completes our proof, and shows that Algorithm 3 correctly computes the minimum-cost arrangement of paintings in every recursive call from OPT(j) to OPT(1).

Algorithm and Analysis 4

Algorithm 4: Inefficient Dynamic Programming Solution

Definition

$OPT(j)$ = The minimized *cost* of arranging j paintings into platforms, where j is the *rightmost* painting being considered. In this formulation, the cost of a platform is equal to the height of the tallest painting on the platform, and the total cost is the sum of the costs of all platforms. Mathematically, $cost = \sum_1^j (\max_{h_k} r_j)$, and h_k is the height of a painting $p_k \in r_j$.

$OPT(i)$ returns the value $M[i]$, where $M[i]$ represents the minimum combined height required to arrange paintings $p_1, \cdots, p_i$ into rows. This solution uses a cost matrix C_{ij} to precompute the cost of valid platform arrangements.

Goal

$OPT(n) = M[n]$ = the minimized cost of arranging n paintings into rows.

Computing $OPT(j)$

Let C_{ij} represent a cost matrix, providing the height of the tallest painting in the platform $[s_i, \cdots, s_j]$, where a valid row is one such that the sum of the widths of its constituent paintings does not exceed W. C_{ij} is represented by:

$$C_{ij} = \begin{cases} \infty & \text{if } \sum_{k=i}^{j} w_k > W \\ \max_{i \le k \le j}(h_k) & \text{otherwise} \end{cases}$$

We can represent $M[i]$ using the following recurrence relation:

$$M[i] = \begin{cases} 0 & \text{if } i = 0 \\ \min_{1 \le j \le i}(M[j] + C_{j,i-1}) & \text{if } i > 0 \text{ and } C_{j,i-1} \neq \infty \\ \infty & \text{otherwise} \end{cases}$$

Backtracking to Determine Painting Attendance

We can use backtracking to determine the exact paintings present in each platform. This is found through the minimum-cost path, which is calculated using the dynamic programming array M and supplementary array P. We begin at $i = n$. At each $OPT(i)$ step we take, we add $M[i]$ to our total cost, and trace

backward using $P[i]$, where $P[i]$ represents the location of the painting that sits at the *end* of the preceding platform. This provides the starting index of the platform with the smallest cost for arranging the first i paintings. The number of paintings on this platform is equal to $i - P[i]$, so by tracing $i = n$ to 0 in P, we compute lengths of the platforms, but in reversed order. To correct this, we simply reverse the sequence.

Analysis

Time Complexity

To compute the cost matrix C_{ij}, the outer loop over i runs n times, the inner loop over j runs from i to $n-1$ for each i, requiring n iterations, and the innermost loop over s executes $\Theta(n)$ times in the worst case for each (i, j) pair. In total, computing the cost matrix is $\Theta(n^3)$.

Calculating the minimum cost for every position using C_{ij} necessitates dynamic programming. To fill the M and P arrays, the outer loop over i runs n times, and the inner loop over j runs 0 to $i-1$ times. So, the time complexity for this is $\Theta(n^2)$.

Finally, reconstructing the solution uses backtracking, and the loop to backtrack runs once from $i = n$ to 0, requiring $\Theta(n)$ time.

$\Theta(n + (n-1) + (n-2) + \cdots + 1) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^3)$

$\Theta(n^3)$ dominates our time complexity calculation, so this is our result.

Correctness

We begin by establishing the following assumption, that no painting is, by itself, too wide to fit on a platform:

$$\forall 1 \leq k \leq n, \nexists p_k \text{ where } w_k > W.$$

We will now examine the construction of C_{ij} values. These values represent, at index $C[i][j]$, the cost of a row containing paintings $[p_i, \cdots, p_j]$ for $1 \leq i \leq j$, calculated as the height of the tallest painting on the platform. Thus, any value of C_{ij} can be used to determine the relative *worth* of row $[p_i, \cdots, p_j]$. These values are used to determine the best combination of rows, since rows that cannot exist are marked as $C_{ij} = \infty$ and will therefore never be chosen over a smaller value. We now establish the following invariant:

I1: At the end of each iteration of the inner loop of C_{ij} computation, $C[i][j]$ correctly stores the maximum height of a row containing paintings p_i to p_j, as long as the combined widths of the paintings do not exceed W.

From our initial assumption, we also establish the second invariant:

I2: There always exists an arrangement of n paintings $p_1, \cdots, p_n$ into rows, as every painting is permitted to sit on its own row, whose cost will be $C[k][k] = h_k, \forall 1 \leq k \leq n$.

This guarantees feasibility.

We will now prove that Algorithm 4 produces correct, minimum-cost output using the optimal substructure property.

Proof: This part resembles Algorithm 3. Let $OPT(j)$ represent the optimal solution for the first j paintings. Then, for any $P[i]$ where $1 \leq i \leq j$, $OPT(j) = \min_{1 \leq ileqj}(C_{ij} + PT(i-1))$. We can say that this holds because if we have computed $M[i-1]$ and the cost C_{ij} of the row starting at i and ending at j, then the optimal solution must select the split point $P[i]$ that minimizes the sum $M[i-1] + C_{ij}$.

By I1, we know that all valid platform arrangements will be considered, and by I2 we know that a feasible solution exists. As we previously stated, we know that Algorithm 4 must check all possible points in P, so all valid solutions will be considered.

Last, we prove that Algorithm 4 produces the minimum cost. For all values in M, we know that $M[k]$ will store the minimum cost for paintings 1 through k. The recurrence relation defined above always takes the minimum over all possible P, so no better solution can exist, as it would contradict the optimal substructure property. Thus, we have proven that Algorithm 4 produces a correct, minimum-cost solution.

Algorithm and Analysis 5

Algorithm 5: Efficient Dynamic Programming Solution

Definition

This algorithm closely follows Algorithm 3. To design a $\Theta(n^2)$ algorithm, we can employ memoization to avoid recomputing solutions to subproblems.

$OPT(j)$ = The minimized *cost* of arranging j paintings into rows, where j is the *rightmost* painting being examined. In this formulation, $cost = \sum_1^j (\max_{h_k} r_j)$, and h_k is the height of a painting $p_k \in r_j$.

Goal

$OPT(n)$ = the minimized *cost* of arranging n paintings into rows.

Computing $OPT(j)$

Let C_{ij}, where $1 \leq i \leq j$, represent the cost of a row consisting of all paintings between and including paintings p_i and p_j. We allow i and j to be equivalent

because it is possible that a row may have only one painting in an optimum solution. C_{ij} is represented by:

$$C_{ij} = \begin{cases} \infty & \text{if } \sum_{k=i}^{j} w_k > W \\ \max_{i \le k \le j}(h_k) & \text{otherwise} \end{cases}$$

To compute $OPT(j)$, we take the minimum sum over i for $1 \le i \le j$ of C_{ij} and $OPT(i-1)$. In this case, $i-1$ represents the *last* painting in the row preceding painting p_i. The following Bellman Equation describes $OPT(j)$:

$$OPT(j) = \begin{cases} 0 & \text{if } j < 1 \\ \min_{1 \le i \le j}(C_{ij} + OPT(i-1)) & \text{otherwise} \end{cases}$$

Backtracking to Determine Painting Attendance

Determining painting attendance in rows follows the exact strategy present in Algorithm 3. At each OPT step, one must find the difference $j - i + 1$, which represents the length of the now-calculated row. This length should be added to the end of a list of row lengths, so that recursive calls create the row list in the correct order as they return upward. Afterward, one can simply examine the list of paintings from left to right, counting the corresponding number for a given row to determine which paintings belong in that row.

5A: Top-Down Recursive Implementation

Algorithm 5A is an implementation of Algorithm 5 that uses memoization and recursion to construct optimal solutions to subproblems of $OPT(j)$ as needed.

Top-Down-OPT$(n, W, h_1, \cdots, h_n, w_1, \cdots, w_n)$:

Precompute all C_{ij} for $C[n][n]$ (one-based indexing):

For $j = n$ to $j = 1$:

For $i = 1$ to $i = j$:

if $(\sum_{k=i}^{j} w_k \leq W)$:

$C[i][j] \leftarrow \max_{i \leq k \leq j}(h_k)$.

else:

$C[i][j] \leftarrow \infty$.

Initialize OPT array $M[n]$.

$M[0] \leftarrow 0$.

Return TD-OPT(n).

TD-OPT(j):

if ($M[j]$ is uninitialized):

$M[j] \leftarrow \min_{1 \leq i \leq j}(C[i][j] + \text{TD-OPT}(i-1))$.

Return $M[j]$.

5B: Bottom-Up Iterative Implementation

Algorithm 5B is an implementation of Algorithm 5 that uses memoization and iteration to construct optimal solutions to subproblems of $OPT(j)$, starting at $OPT(1)$ and working up to $OPT(j)$.

Bottom-Up-OPT$(n, W, h_1, \cdots, h_n, w_1, \cdots, w_n)$:

Precompute all C_{ij} for $C[n][n]$ (one-based indexing):

For $j = n$ to 1:

For $i = 1$ to j:

if $(\sum_{k=i}^{j} w_k \leq W)$:

$C[i][j] \leftarrow \max_{i \leq k \leq j}(h_k)$.

else:

$C[i][j] \leftarrow \infty$.

Initialize OPT array $M[n]$.

$M[0] \leftarrow 0$.

For $k = 1$ to n:

$M[k] = \min_{1 \leq i \leq k}(C_{ik} + M[i-1])$.

Return $M[n]$.

Analysis

Time Complexity

5A: Top-Down Algorithm 5A precomputes all values of C_{ij}, taking $\Theta(n^2)$ time to do so, as all n paintings are checked no fewer and no more than n times each. Afterward, all n values of the OPT-array M require n comparisons against the rest of the painting list to determine the minimum possible value. This takes $O(n^2)$ time. Our total is now $O(n^2 + n^2) = O(2n^2) = \Theta(n^2)$.

5B: Bottom-Up Algorithm 5B, just like 5A, precomputes all C_{ij} values in $\Theta(n^2)$ time. The difference is now that we iterate from 1 to n, making n comparisons at each step, traking $O(n^2)$ time. Our total is now $O(n^2 + n^2) = O(2n^2) = \Theta(n^2)$.

Correctness

We begin by establishing the following assumption, that no painting is, by itself, too wide to fit on a platform:

$$\forall 1 \leq k \leq n, \nexists p_k \text{ where } w_k > W.$$

We will now examine the construction of C_{ij} values. These values represent, at index $C[i][j]$, the cost of a row containing paintings $[p_i, \cdots, p_j]$ for $1 \leq i \leq j$,

calculated as the height of the tallest painting on the platform. Thus, any value of C_{ij} can be used to determine the relative *worth* of row $[p_i, \cdots, p_j]$. These values are used to determine the best combination of rows, since rows that cannot exist are marked as $C_{ij} = \infty$ and will therefore never be chosen over a smaller value. We now establish the following invariant:

I1: At the end of each iteration of the inner loop of C_{ij} computation, $C[i][j]$ correctly stores the maximum height of a row containing paintings p_i to p_j, as long as the combined widths of the paintings do not exceed W.

From our initial assumption, we also establish the second invariant:

I2: There always exists an arrangement of n paintings $p_1, \cdots, p_n$ into rows, as every painting is permitted to sit on its own row, whose cost will be $C[k][k] = h_k, \forall 1 \leq k \leq n$.

The proof of Algorithm 5 strongly resembles that of Algorithm 3, and the use of memoization dramatically reduces the running time. We will now prove using induction that Algorithm 5 produces the correct, optimal, minimized cost of organizing the paintings into rows.

Proof: For OPT(j) where $j = 1$, the case is simple. A set of one painting has only itself to be placed in a row, so the maximum height of the row is $C[1][1]+0 = h_1$, that of the singular painting, following and upholding I2.

Moving forward, when $j > 1$, Algorithm 5 checks for the existence of $M[j]$, the minimum cost of placing j paintings into rows. If $M[j]$ has not been initialized, then OPT(j) computes the value of the minimum cost over i for C_{ij}+OPT($i-1$), recursively calculating the minimum possible cost of a row being created whose rightmost painting is p_j. For OPT($i-1$), Algorithm 5 follows the same check-and-recurse pattern for returning the cost of the next row that ends at p_{i-1}. Because we do not modify values in C, I1 is upheld, and we are guaranteed to compute the minimum cost for all $M[k]$ for $1 \leq k \leq j$. The value of OPT(j) is minimized at every recursive step, whose solution is calculated upon the existence of optimal solutions to previous subproblems. By using the optimal substructure property of Problem G, Algorithm 5 properly constructs a correct solution, outputting the minimized cost of a set of paintings by finding the minimum cost of a row, over all possible rows ending on a painting p_j, over all $1 \leq j \leq n$. This completes our proof, and shows that Algorithm 5 correctly computes the minimum-cost arrangement of paintings in every recursive call from OPT(j) to OPT(1).

Experimental Comparative Study

Method

For Milestone 2, we had Program 3 written in C++, and programs 4, 5A, and 5B written in Python. As such, Program 3 was tested individually, but all the tests followed the same format.

We used five tests for each program, with varying numbers of elements for each. The heights in these tests use a random set of integers in $\mathbb{Z}^+$, where the length of the test set is equivalent to 5 increasing multiples of 15 for Program 3, 5 increasing multiples of 200 due to execution time constraints for Program 4, and 5 increasing multiples of 1000 for 5A and 5B. The difference in the multiple sizes is due to the necessary execution time, as Program 3 is exponential and a minimum of 1000 elements would have a worst-case time complexity of $1000 * 2^{999}$. We simply do not have enough time as a species to test this way.

Widths are generated as a random number 1 through 10 for the same length as the list of heights. The maximum row width is constant throughout our tests.

Each test was run five times and averaged to get the performance time for the specified number of elements, and then the dataset size and average time were ouput in a single row in a CSV file for that program, with the number of data rows equaling the number of multiples.

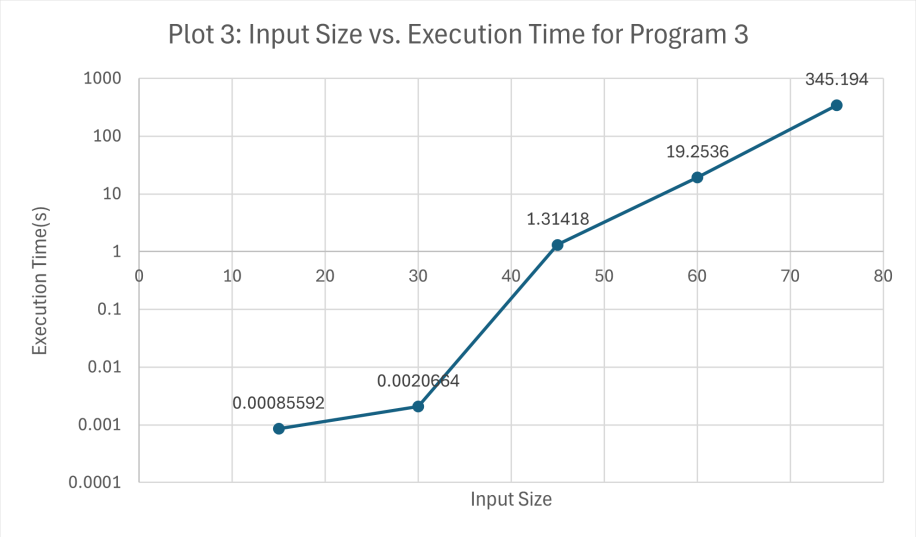
Performance Comparison

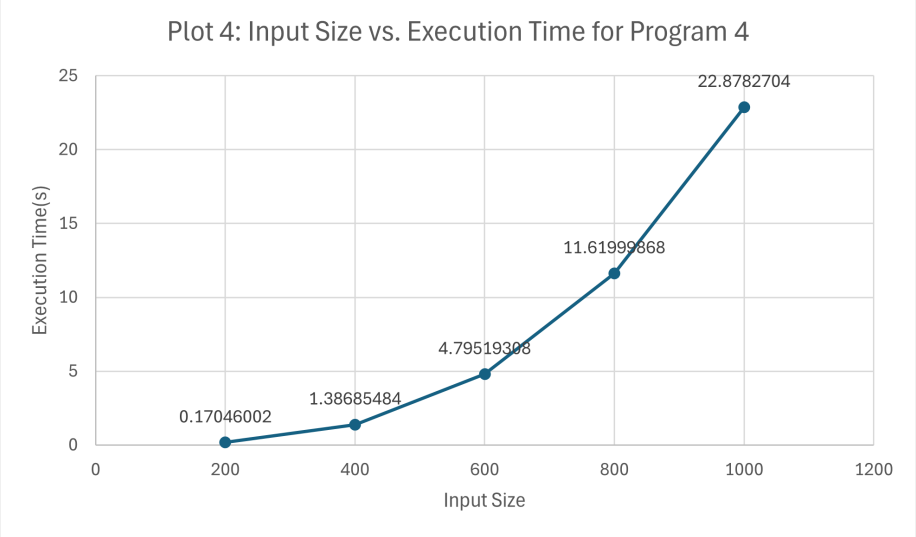
For our tests, we used five multiples of 15 for Program 3, five multiples of 200 for Program 4, and five multiples of 1000 for programs 5A and 5B. We decided this difference was necessary while waiting excessively for test results from Program 3. Test set sizes for Program 3 include: $[15, 30, 45, 60, 75]$. Test set sizes for program 4 include: $[200, 400, 600, 800, 1000]$ and 5A and 5B include $[1000, 2000, 3000, 4000, 5000]$.

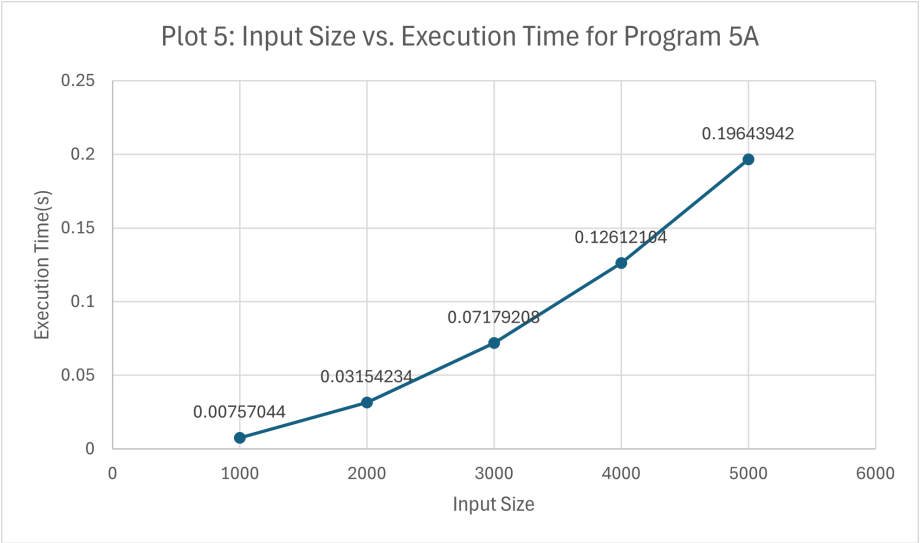
Again, heights are randomly generated for a list whose length is equal to each of the aforementioned sizes, and widths are randomly generated 1 through 10 for the same size as each height list. Each size has *one* corresponding randomly generated list, which is tested five times in this case, since we wanted the average of five runs.

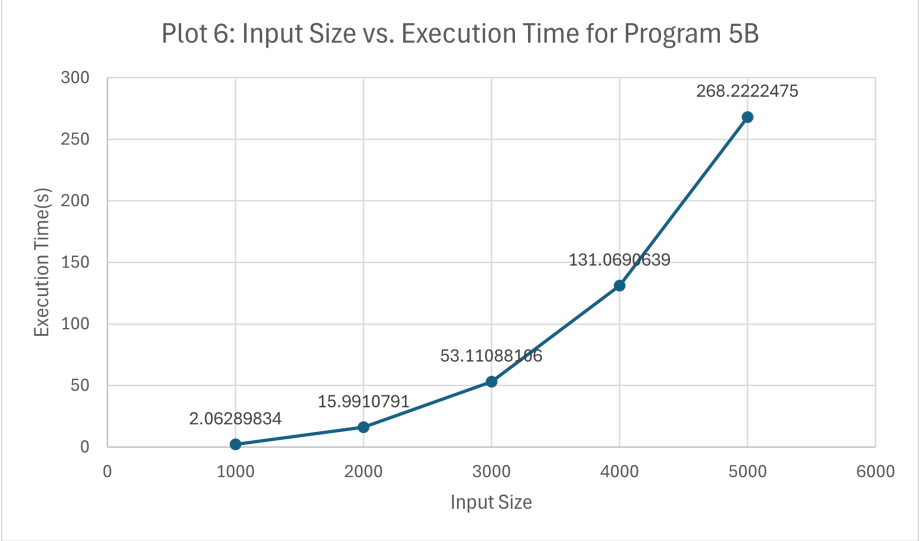
Executions of programs 4, 5A, and 5B must occur separately since one test file was written for modularity, and so operate on the same list sizes, but not the same list contents.

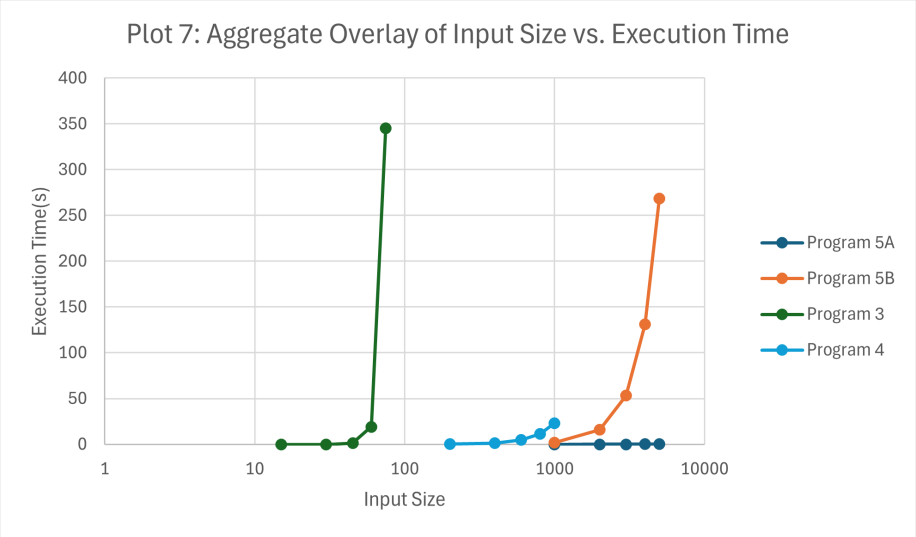
Data

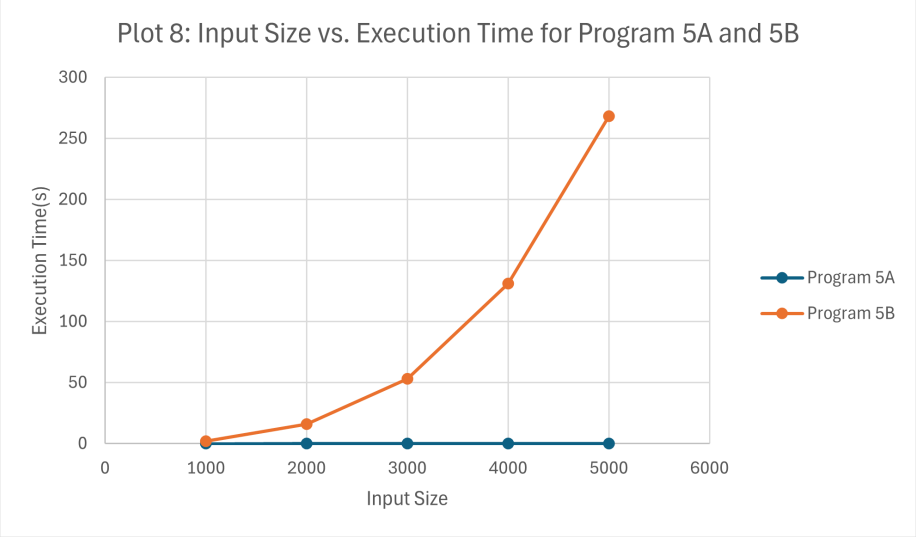

Plot 3: Input Size vs. Execution Time for Program 3
Execution Time(s)
1000
100
10
1
0.1
0.01
0.001
0.0001
0
10
20
30
40
50
60
70
80
345.194
19.2536
1.31418
0.0020664
0.00085592
Input Size


Plot 4: Input Size vs. Execution Time for Program 4
Execution Time(s)
25
20
15
10
5
0
0
200
400
600
800
1000
1200
22.8782704
11.61999868
4.79519308
1.38685484
0.17046002
Input Size


Plot 5: Input Size vs. Execution Time for Program 5A
Execution Time(s)
0.25
0.2
0.15
0.1
0.05
0
0.19643942
0.12612104
0.07179208
0.03154234
0.00757044
0
1000
2000
3000
4000
5000
6000
Input Size


Plot 6: Input Size vs. Execution Time for Program 5B
Execution Time(s)
300
250
200
150
100
50
0
268.2222475
131.0690639
53.11088106
15.9910791
2.06289834
0
1000
2000
3000
4000
5000
6000
Input Size





NOTE: Plot 3 is logarithmic in base 10 for Execution Time, and Plot 7 is logarithmic in base 10 for Input Size.

Program 3		Program 4	
Input Size	**Execution Time (s)**	**Input Size**	**Execution Time (s)**
15	0.000856	200	0.17046
30	0.002066	400	1.386855
45	1.31418	600	4.795193
60	19.2536	800	11.62
75	345.194	1000	22.87827

Program 3	Program 4

Program 5A		Program 5B	
Input Size	**Execution Time (s)**	**Input Size**	**Execution Time (s)**
1000	0.00757	1000	2.062898
2000	0.031542	2000	15.99108
3000	0.071792	3000	53.11088
4000	0.126121	4000	131.0691
5000	0.196439	5000	268.2222

Analysis

There are a few major points of interest. To begin, Program 3 is set on a logarithmic curve in base 10 to better illustrate how its performance scales with input size. It climbs extremely rapidly, even for a C++ implementation, as compared to the Python versions for the others. Nonetheless, Program 3 obviously has the worst performance of all the algorithms, as made evident by its exponential time complexity. This was to be expected, and it was quite difficult to test the algorithm and determine realistic values for input sizes.

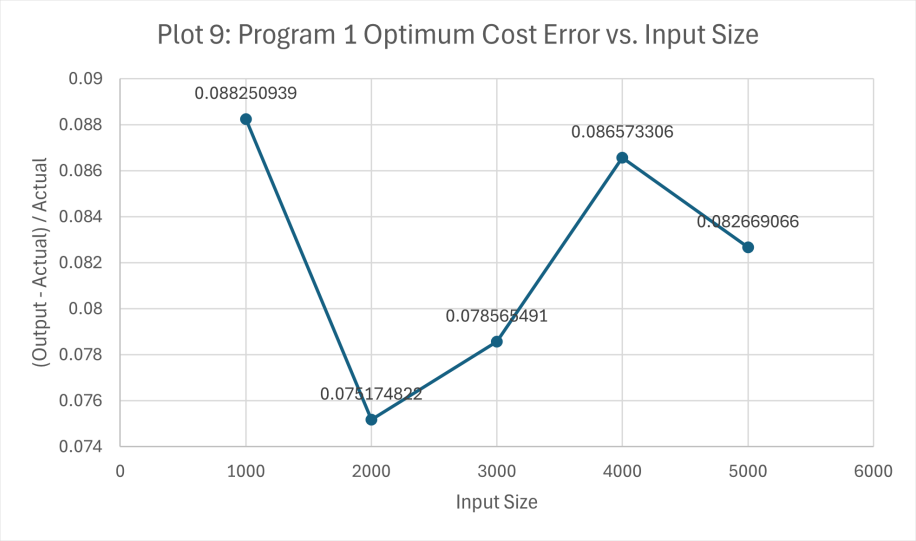
We also note that Program 4 took too long to test at the same input sizes used for programs 5A and 5B, so we had to trim the input sizes. Nonetheless, we can see that its necessary execution time climbs faster than 5A and 5B, but still slower than 3, as expected for a time complexity between these. Most of Program 4's time is spent calculating costs, hence its $\Theta(n^3)$ execution time.

Programs 5A and 5B do follow relatively the same rate of growth against input size, but when placed next to each other, it becomes evident that 5A, the top-down implementation of Algorithm 5, performs substantially better than 5B. We believe this is because iteratively computing all possible values OPT(1) to OPT(n) is likely to be slower than recursively computing them *as-needed*. 5A also has the best performance of all the programs, which was the rationale for its use in the following comparison.

Program 1 vs. Program 5A Accuracy Comparison

To test how accurate Program 1 output is compared to an optimal solution, we used the same testing strategies as before, except the data being examined is $\frac{(h_g - h_o)}{h_o}$, where h_o is the optimal height of Program 5A, and h_g is the greedy height of Program 1. We used randomly generated heights in lists of size $[1000, 2000, 3000, 4000, 5000]$, with a constant max width of 10 and randomly generated painting widths from 1 to 10.

Data



NOTE: $\frac{(h_g - h_o)}{h_o}$ **is represented as** $\frac{(\textbf{Output - Actual})}{Actual}$**.**

Input Size	Program 1 Optimum Cost Error	Program 1 Optimum Cost Error %
1000	0.088251	8.8251
2000	0.075175	7.5175
3000	0.078565	7.8565
4000	0.086573	8.6573
5000	0.082669	8.2669

Analysis Examining the data, it is evident that the greedy algorithm overshoots the minimum every time. Its output *must* be a greater cost than that of the optimum solution, obtained from Program 5A, our top-down implementation of Algorithm 5. Program 1, and therefore Algorithm 1, is not sufficient to solve Problem G, as it does not cover all possible solutions. It does not have an "undo" mechanism to allow for decisions, it simply takes the best possible option at every step.

To determine exactly how far off Program 1 is, we can multiply its error values by 100 to convert them into percents, as seen in the table.

We observe that, on average, Program 1 is off by $\approx 8.2\%$ from the actual value. The actual value of its inaccuracy tends to vary, but in all five cases it did not correctly determine the optimal minimum cost arrangement of paintings.

Conclusions

Summary

For this milestone, we were tasked with creating three new algorithms for solving the General form of our original problem definition. In our first milestone, we designed a greedy algorithm to solve special cases of the problem, in which the paintings were arranged in decending order of height, or were organized with heights descending toward one central painting. However, for this milestone we were tasked with solving the general case where paintings can be arranged in any height. We achieved this by designing three different Dynamic Programming algorithms that solve the problem with three different time complexities.

The first algorithm solves the problem using a "Brute Force" algorithm, which uses a running recursion tree to solve every subproblem manually, without storing the output of each iteration. The second variation of the algorithm creates a cost matrix and precomputes the costs associated with every possible painting. Lastly, the third variation performs the same process as the first algorithm, storing the result of each call to save on computation. This algorithm was implemented using both a top-down and bottom-up approach.

Implementation Challenges

Program 3

Program3 was difficult because an exponential-time algorithm is basically guaranteed to hit Python's recursion limit, and attempting to find an upper limit that can handle the recursion is difficult. C++ was chosen instead, as it would (hopefully) be faster, at the expense of being harder to write. C++ is pickier with typing, and mutability and indexing between the two languages can feel like a guessing game.

Aside from language challenges, Algorithm 3 was not exceedingly difficult to implement into code, it just required some hack-y solutions and careful, thorough debugging to resolve rampant indexing issues. Given that the description of Algorithm 3 is relatively generic and lacks pseudocode, program3 required mental acrobatics for designing a recursive solution. For example, it would have been asinine to attempt to create needed values on the fly when determining the minimum solution in the **options** vector, and devising a lambda function to compare the second values of all the tuples was not expected to show up on the to-do list.

There was also some confusion related to determining what value of C_{ij} to use in the case that a row could not exist, as well as a redefinition of bounds in the original algorithm that was not initially accounted for, that would allow individual paintings to sit on their own rows. Algorithm 3 was the first and possibly most difficult algorithm we completed for this Milestone, and it was just as difficult to implement correctly.

Program 4

Although algorithm 4 is slower and has a larger time complexity ($\Theta(n^3)$) than algorithms 5A and 5B, it was exceptionally difficult to program. Creating a structure of code that runs in $\Theta(n)$ time within the already functioning $\Theta(n^2)$ algorithm (since we decided creating the $\Theta(n^2)$ first and then extrapolating it from there would be the easiest), turned out to be more difficult than anticipated. To make it work we ended up having to alter the dynamic programming approach to construct a cost matrix. Unlike our initial $\Theta(n^2)$ time algorithm, we had to explicitly recomputate all possible groupings of paintings so that we could evalueate their total widths and max heights.

This required a triple-nested loop, where the first loop iterates through all potential starting paintings, the second loop iterates through all possible ending paintings for the current start point, and the third loop gets the total width and max height for the group of paintings from i to j.

Another difficulty of algorithm 4 was that backtracking became a lot more complicated, since now we could no longer just follow the recursive calls, but instead traverse the P array and reconstruct the optimal assignment only using the cost matrix. So overall, it seems that algorithm 4 was more complicated to create than expected.

Program 5A

Because this algorithm was heavily dependent on the structure of Algorithm 3, the implementation was not too much of a challenge. The algorithm needed to store each of the calls to a subproblem OPT(j), which were kept in an array passed through each recursive function call. By storing these values, redundant calls to OPT(j) can be stored and retrieved, which significantly reduces the complexity of the algorithm. There were only some slight issues with accidentally referencing variables rather than creating true copies, but there were no other challenges aside from that minor hurdle.

Program 5B

With the Pseudocode done, converting the Bottom-Up approach into functioning code was not too difficult. There were some complications in converting the Pseudocode to working code in a way that was readable and easy to follow, but it generally resulted in code that was simpler and more straigforward than the Top-Down approach.