Programming Project Group 3, Milestone 1: COP4533 - Algorithms Abstraction and Design Fall '24

Elijah Johnson

- Repository, Code Standards, Report Structure and Documentation
- Algorithm 1 Design, Psuedocode, Implementation, Correctness Analysis
- Question 2
- Algorithm 2 Correctness Analysis
- Testing Implementations, Graphs, Experimental Study Report

Patrick Kallenbach

- Question 1
- Algorithm 2 Design, Psuedocode, Implementation
- Testing Implementations
- Gradescope Submissions

Nicholas Lindner

- Algorithm 1 Time Complexity Analysis
- Algorithm 2 Time Complexity Analysis
- Conclusions

Algorithm and Analysis 1

Problem S1

Given the heights $h_1, \cdots, h_n$, where $h_i \geq h_j, \forall i < j$, and the base widths $w_1, \cdots, w_n$ of n paintings, along with the width W of the display platform, find an arrangement of the paintings on platforms that minimizes the total height.

(Note: The heights of the paintings form a monotonically non-increasing sequence, as in EXAMPLE 1.)

Example 1

$n = 7, W = 10$
$h_i = [21, 19, 17, 16, 11, 5, 1]$
$w_i = [7, 1, 2, 3, 5, 8, 1]$

Solution 1

$Platform_1 = [s_1 \cdots s_3]$;
$Platform_2 = [s_4 \cdots s_5]$;
$Platform_3 = [s_6 \cdots s_7]$;
$cost = 21 + 16 + 5 = 42$

Algorithm 1: Greedy Left-Right Solution

- Let W represent max row width.
- Let w_i represent the width of painting p_i.
- Let $w(r_j)$ represent the width of a row, where each r_j is initialized with $w(r_j) = 0$.
- Let *cost* represent the total height of the platforms, initialized to $cost = 0$.
- Keep track of the paintings in the current row, and of the total number of rows.

Beginning with $j, i = 0$, repeat the following for each painting p_i:

- If $w(r_j) + w(p_i) \leq W$:
  - Add painting p_i to row r_j, and update $w(r_j)$.
  - Proceed to the next painting.
- Otherwise if the row is full or there are no paintings left:
  - Add the height of the first painting in r_j to *cost*, as it is the tallest.
  - Record r_j's length, begin a new row, and continue.

Return the number of rows, the total *cost*, and the list containing each platform's lengths.

Analysis

Time Complexity

Algorithm 1 has a time complexity of $O(n)$. Algorithm 1 proceeds through the list of paintings only one time, performing some $O(1)$ conditionals and $O(1)$ additions to auxiliary arrays. However, each painting must be checked and will be processed only once, terminating only when all paintings have been checked, so the total running time is $O(n)$.

Correctness

Consider the sequence P of n paintings, whose heights $h_i = [h_1, h_2, \cdots, h_n]$ are monotonically non-increasing. We will prove that the above greedy algorithm is correct and satisfies the following conditions:

- The order of the paintings is not changed.
- The combined widths of a row $w(r_j) \leq W$.
- For r rows, the total cost $\sum_0^{j=r} h_j$, where h_j is the first and tallest painting in a row, is minimized.

Due to the nature of a sequence P of n paintings, whose heights are monotonically non-increasing, this problem focuses more precisely on minimizing the number of rows, or maximizing the number of paintings per row, while preserving the original order. This is because the order of the paintings cannot be such that for two paintings p_i and p_j, where $\forall ij, i < j, h_i < h_j$. So, a random selection of paintings placed into rows that preserve the original order of the paintings would create an arrangement such that the first painting in any row is at least the tallest. The goal is that the number of rows must be minimized, or the number of paintings per row must be maximized, to minimize the total cost.

Proof: If paintings are selected in such a way that they maximize the number of paintings per row, then the height of the tallest painting in each row will be minimized. If there exists space in a row for a painting p_i in a row r_j, meaning $w(r_j) + w_i \leq W$, then placing it in the row will minimize the cost of the next row, since $\forall pq, p < q, h_p \geq h_q$, as per the Problem S1 definition. If p_i must be placed in a new row, then it will be the shortest possible painting that can be placed there. The shortest possible painting will be chosen to be placed first in each row, so Algorithm 1 will maximize the number of paintings per row, minimizes the total number of rows, and therefore minimizes the total cost.

Question 1

Give an input example showing that Algorithm 1 does not always solve Problem G.

Problem G (Generic Problem)

Given the heights $h_1, \cdots, h_n$ and the base widths $w_1, \cdots, w_n$ of n paintings, along with the width W of the display platform, find an arrangement of the paintings on platforms that minimizes the total height.

Solution

Consider the sequence P of n paintings, with heights $h_i = [2, 3, 1, 4, 3]$ and widths $w_i = [1, 1, 1, 1, 1]$, with $W = 2$. Algorithm 1 chooses as many paintings as can fit in a row while maintaining their original order. So, in this example, Algorithm 1 selects the following:

- $Platform_1 = [2, 3]$;
- $Platform_2 = [1, 4]$;
- $Platform_3 = [3]$;
- $cost = 3 + 4 + 3 = 10$

The ordering of the paintings' heights results in the minimum cost being found by Algorithm 1 as 6, since the cost of each row is decided by the first painting's height $(2 + 1 + 3)$. However, the true total height of the display would be 10, the sum of the maximum heights of each row.

The optimal solution for this setup actually has a height of 8. This arrangement maintains the original order of the paintings, but two of the rows are now shorter.

- $Platform_1 = [2, 3]$;
- $Platform_2 = [1]$;
- $Platform_3 = [4, 3]$;
- $cost = 3 + 1 + 4 = 8$

Hence, Algorithm 1 does not always produce a minimum-cost solution for Problem G.

Consider the sequence P', a permutation of P with paintings in decreasing order of height: $h_i = [4, 3, 3, 2, 1]$ and $w_i = [1, 1, 1, 1, 1]$. Then the solution produced by Algorithm 1 is the following:

- $Platform_1 = [4, 3]$;
- $Platform_2 = [3, 2]$;
- $Platform_3 = [1]$;
- $cost = 4 + 3 + 1 = 8$

Because this permutation adheres to the required monotonically non-decreasing ordering of paintings for Algorithm 1, it produces the arrangement of rows with minimum height.

Question 2

Give an input example showing that Algorithm 1 does not always solve Problem S2.

Problem S2

Given the heights $h_1, \cdots, h_n$, 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 $w_1, \cdots, w_n$ of n paintings, along with the width W of the display platform, find an arrangement of the paintings on platforms that minimizes the total height.

Solution

Consider a Problem S2 instance where $h_i = [4, 2, 1, 2, 4]$, and $w_i = [1, 1, 1, 1, 1]$, with $W = 2$. Algorithm 1's output is as follows:

- $Platform_1 = [4, 2]$;
- $Platform_2 = [1, 2]$;
- $Platform_3 = [4]$;
- $cost = 4 + 2 + 4 = 10$

An optimum output would resemble the following:

- $Platform_1 = [4, 2]$;
- $Platform_2 = [1]$;
- $Platform_3 = [2, 4]$;

- $cost = 4 + 1 + 4 = 9$

This is an optimum solution maintaining original order, as the minimum cost is 9, the lowest possible cost for the given ordering of the paintings. This demonstrates that Algorithm 1 does not always produce a minimum-cost solution for instances of Problem S2.

Algorithm and Analysis 2

Problem S2

Given the heights $h_1, \cdots, h_n$, 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 $w_1, \cdots, w_n$ of n paintings, along with the width W of the display platform, find an arrangement of the paintings on platforms that minimizes the total height.

(Note: The heights of the paintings follow a unimodal function with a single local minimum, as in EXAMPLE 3.)

Example 3

$n = 7, W = 10$
$h_i = [12, 10, 9, 7, 8, 10, 11]$
$w_i = [3, 2, 3, 4, 3, 2, 3]$

Solution 3

$Platform_1 = [s_1 \cdots s_3]$;
$Platform_2 = [s_4]$;
$Platform_3 = [s_5 \cdots s_7]$;
$cost = 12 + 7 + 11 = 30$

Algorithm 2: Greedy Split Solution

- Let W represent max row width.
- Let w_i represent the width of painting p_i.
- Let $w(r_j)$ represent the width of a row, where each r_j is initialized with $w(r_j) = 0$.
- Let *cost* represent the total height of the platforms, initialized to $cost = 0$.
- Let h_i represent the height of a painting p_i.
- Let $h(r_j)$ represent the height of a row.
- Keep track of the paintings in the current row r_j.
- Keep track of "top rows" T and "bottom rows" B, both initialized as empty sets.

Beginning with $j, i = 0$, repeat the following for each painting p_i until $minimumFound$ is $true$:

- If $w(r_j) + w_i \leq W$:
  - Append p_i to the *end* of r_j.
  - Update the current row's width by adding w_i to $w(r_j)$.
  - If p_i is the first painting in r_j, add h_i to *cost*.
- Otherwise if the row r_j is full:
  - Append r_j to the *end* of T.
  - Continue; return to the beginning of p_i's iteration.
- If the $h_{i+1} > h_i$:
  - Append r_j to the end of T.
  - Store the current index i as $minIndex$.
  - Set $minimumFound$ to *true*.
- Proceed to the next painting.

Now that $minimumFound$ is *true*, repeat the following for the remaining paintings, starting from p_n and working inward to, but not including, $p_{minIndex}$:

- If $w(r_j) + w_i \leq W$:
  - Append p_i to the *beginning* of r_j.
  - Add w_i to $w(r_j)$.
  - If p_i is the first painting added to r_j, add h_i to *cost*.
- Otherwise if there are no more paintings or if r_j is full:
  - Append r_j to the *beginning* of B.
  - Continue; return to the beginning of p_i's iteration.

When no paintings are left, add what remains of r_j to the *beginning* of B. Then:

If $w(T_{end}) + w(B_{start}) \leq W$:

- Merge T and B, combining T_{end} and B_{start} into one row.
- Subtract $min(h(T_{end}), h(B_{start}))$ from *cost*.

Otherwise:

- Merge T and B, leaving T_{end} and B_{start} as separate rows.

Return the number of rows, the total *cost*, and the list containing each platform's lengths.

Analysis

Time Complexity

Algorithm 2 has a time complexity of $\theta(n)$. Algorithm 2 proceeds through the list of paintings only one time, performing some $O(1)$ conditionals and $O(1)$ additions to auxiliary arrays. When the minimum height painting is found, it then proceeds from the other end toward the $minIndex$. Each painting must be checked and will be processed only once, terminating only when all paintings have been checked. At the end, two rows may be merged. If this is the case, it may take a maximum of n accesses to merge the rows into one, and then a $O(1)$ operation to get the minimum between their heights, but this still results

in a $\theta(n)$ total complexity, as every painting must be checked, but none will be checked more than twice.

Correctness

Consider the sequence P of n paintings, whose heights $h_i = [h_1, h_2, \cdots, h_n]$ follow a unimodal function with a single local minimum. We will prove that the above greedy algorithm is correct and satisfies the following conditions:

- The order of the paintings is not changed.
- The combined widths of a row $w(r_j) \leq W$.
- For r rows, the total cost $\sum_0^{j=r} h_j$, where h_j is the tallest painting in a row, is minimized.

An instance of Problem S2 can be understood as two Problem S1 instances, where the latter is reversed in order, and there exists one p_i whose height is the least of all paintings.

Algorithm 2 is constructed in a manner resembling that of two runs of Algorithm 1 with the latter in reverse order. The former proceeds until a minimum index is found, and the latter proceeds backward from the end until the minimum index is reached.

Using correctness of Algorithm 1, we can say that Algorithm 2 is also correct.

Proof: Let R represent the set of paintings from p_0 to p_m inclusive, where $\exists m$ such that $\forall i < j \leq m, h_i \geq h_j$, and $\forall m \leq i < j, h_i \leq h_j$. In this case, m represents the index of the minimum-height painting in P. R, then, has heights ordered in a monotonically non-increasing fashion, which can be understood as an instance of Problem S1. Let the remaining paintings S represent the reverse of another instance of Problem S1, since all paintings *after* R are monotonically non-decreasing. Therefore, an instance of Problem S2 can be treated as an instance of Problem S1 directly followed by another reversed instance.

Algorithm 2 follows the same logic as Algorithm 1, whose correctness we have already proved, but is designed to handle the now deconstructed Problem S2 structure. For the set of paintings R, A2 follows the same steps as A1, just performing an extra check to determine if, for painting p_i, whether $h_{i+1} > h_i$. If this is the case, the minimum value has been found, and its index is stored as $minIndex$. Along the way, paintings are added to the *ends* of rows of *decreasing* order in a set of rows T.

From there, A2 proceeds in a backwards fashion from the end of the list to $minIndex$, following the same steps as A1, only now in reverse order from the end of the list. In this case, paintings are added to the *beginnings* of rows of *increasing* order in a set of rows B. *(NOTE: A2 does not check $p_{minIndex}$ twice.)*

However, A2 is not finished after all paintings have been checked. The last row of T and the first row of B are of unknown widths, each with their own tallest paintings, and can be combined to minimize *cost*. A2 checks that their combined

widths $w(T_{end}) + w(B_{start})$ are less than or equal to W, and merges them if so. It then subtracts the minimum of T_{end}'s and B_{start}'s tallest paintings from the total cost, meaning only the height of the tallest painting in the new combined row is accounted for. If their combined widths exceed W, no further action is taken.

We have shown that R is assessed by T in A2, and S is assessed by B, and their (potential) overlap is addressed accordingly. Therefore, Algorithm 2 produces a correct output for an instance of Problem S2, satisfying the aforementioned conditions.

Experimental Comparative Study

Method

We have devised a system to easily test Algorithms 1 and 2 in *program1.py* and *program2.py*, using some constants and Python's CSV library to generate a CSV file. These tests will use an integer value in $\mathbb{Z}^+$, and generate sets of Problem S1 and Problem S2 instances for each algorithm, respectively. In *program1.py*, the value, say, x, will be used to generate x multiples of 1000 elements from 1 to x inclusive, whose elements begin at $x * 1000$ and decrease to 1. In *program2.py*, the elements are randomly generated, as well as the index of the minimum, but the output fits the problem description.

Each run of n elements is performed a specific number of times c, where c runs are then averaged to get the performance time for n elements.

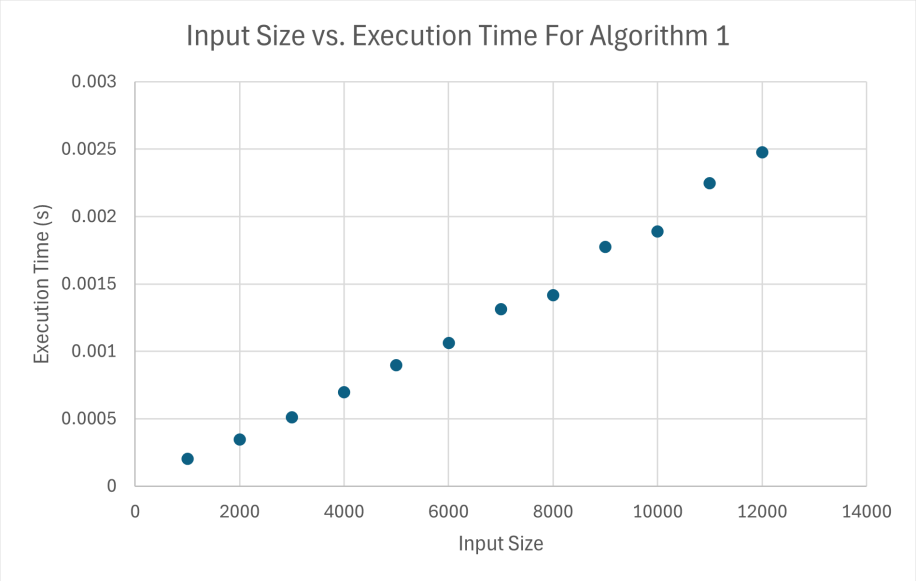
Performance Comparison

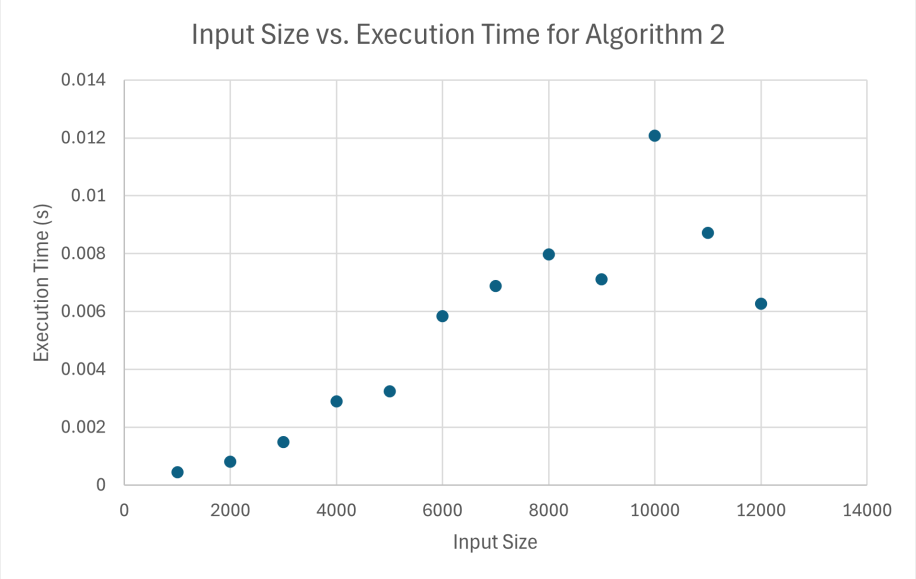
For our tests, we used twelve multiples of 1000 for our input sizes, meaning each algorithm was tested on $[1000, 2000, \cdots, 12000]$ elements. Algorithm 1 was tested on a randomly generated then sorted list of numbers from 1 to $n * 1000$.

For Algorithm 2, we used a randomly generated number from 2 to 1000 for each element in the first half, then sorted the first half in descending order. The midpoint was determined randomly, and was always set to a value of 1. The second half was generated the same way as the first, only now in ascending order. These sections were combined to generate a list of elements in the following pattern:

- For n elements, the test set could resemble $[1000, 448, 362, \cdots, 1, \cdots, 2, 992, 993]$.

Data





Algorithm 1		Algorithm 2	
Input Size	**Execution Time (s)**	**Input Size**	**Execution Time (s)**
1000	0.000193	1000	0.000343
2000	0.000364	2000	0.000812
3000	0.000541	3000	0.001047
4000	0.000715	4000	0.001566

Algorithm 1		Algorithm 2	
5000	0.000877	5000	0.003689
6000	0.001098	6000	0.004294
7000	0.001276	7000	0.005829
8000	0.001484	8000	0.004366
9000	0.001673	9000	0.003972
10000	0.001980	10000	0.008214
11000	0.002158	11000	0.009492
12000	0.002307	12000	0.012766

Analysis

It is evident that Algorithm 1 closely follows an $O(n)$ performance time as input size scales linearly. Algorithm 2 does the same, however it seems to be more prone to performance fluctuations, as outliers become more common with larger input sizes. That being said, both algorithms perform very quickly compared to the time of human actions, with the longest performance time still being hardly longer than $\frac{1}{100}^{th}$ of a second at 12000 elements. Nonetheless, it is clear our algorithms are consistent and could be used in practical applications suitable for the problem description with negligible performance overhead.

Conclusions

Summary

In Milestone 1, we learned how and when to apply greedy algorithms to specific types of problems. Through our efforts, we found that different variations of a similar problem may sometimes require varying implementations of a greedy algorithm, but similar algorithmic design paradigms can easily be implemented across similar problems.

Upon designing and implementing Algorithm 1, we initially believed we would need to create an entirely new algorithm for Problem S2. However, after attempting from scratch using a left->right scan of the paintings, we discovered that extending the logic of Algorithm 1 was sufficient to solve Problem S2 efficiently. We discovered the novelty and importance of applying algorithmic principles between similar problems, bolstering both correctness and performance.

Problem S1

A lot of care went into understanding the problem and its nuances, and through careful analysis, we were able to craft highly-detailed and well-structured psuedocode for our Problem S1 solution. This allowed us to break down the problem logically and into well-detailed steps, enabling the easy implementation of our algorithm using Python.

Furthermore, Problem S1 only requires a basic greedy algorithm, so it was fairly straightforward regardless. Our original implementation worked as intended immediately, and the only changes made were ones that optimized the algorithm's performance.

Technical challenges were few and far between. After our initial optimization changes, we found that the algorithm unfortunately was performing worse than before. Thankfully this was resolved quickly with some careful reading and problem-solving skills. We were also fortunate to have a team member experienced with Python CSV, making it easy to generate test results.

Problem S2

While attacking Problem S2, we invested more time than with Problem S1 into fully understanding the nature of the challenge, as it was expected to be more difficult. Because of this, we were able to identify edge cases quickly, and more importantly, correct careless mistakes that were made when designing Algorithm 2. We initially created an entirely new greedy algorithm that attempted to read the list of paintings from left to right, but we discovered that applying Algorithm 1's strategy to two separate sub-problems was more effective (and performant).

One of the main challenges we faced with Problem S2 was simply achieving correct output. Even though we knew that Algorithm 1 could be the foundation, and modified to solve Problem S2, it was difficult to get it to work for all inputs. A significant amount of debugging and logic fixes were necessary to arrive at our final implementation. Even still, inputs with large sizes (roughly > 9000) seem to cause widly varying performance times, however the trends show that our time complexity analyses are still correct, and that Algorithm 2 still performs well.