Synthesis Assignment 3
Erdun E
November 10, 2024
Dr. Alan Jamieson

CS 5800: Algorithm
Synthesis Assignment 3

6 Greedy Algorithms Continued

6.1 Encodings Huffman's Encoding Algorithm

6.11 Explanation

Huffman's Encoding Algorithm is a greedy approach used for data compression, where symbols are encoded based on their frequencies. This algorithm prioritizes symbols that appear more frequently by assigning them shorter binary codes, while less frequent symbols receive longer codes. By doing so, Huffman encoding efficiently reduces the average code length, making it ideal for storage and transmission applications.

Consider a text file where the letters "E", "R", "D", "U",and "N" appear most frequently. Huffman encoding assigns shorter binary codes to these commonly occurring letters, while rarer letters receive longer codes. In my understanding, Huffman encoding is similar to how we arrange frequently used items within easy reach, and less commonly used items further away. Just as this organization reduces the time needed to find important items, Huffman encoding minimizes the space needed by assigning shorter codes to frequently used symbols.

Algorithm 1 Huffman's Encoding Algorithm

Input : A list of characters with their corresponding frequencies
Output: A Huffman tree with binary encodings for each character
Create a priority queue and insert all characters based on their frequencies
while *there is more than one node in the priority queue* **do**
    Remove the two nodes with the lowest frequencies
    Create a new node with these two nodes as children, with its frequency as the sum of their frequencies
    Insert the new node back into the priority queue
The last remaining node is the root of the Huffman tree
Traverse the tree to assign binary codes to each character, assigning '0' to the left branch and '1' to the right branch

Step-by-Step Explanation of the Pseudocode

- Begin by creating a priority queue, where each character is stored along with its frequency. Characters with lower frequencies will be dequeued before those with higher frequencies, ensuring that less common characters end up with longer codes.
- Building the Huffman Tree while there is more than one node in the queue, repeat the following steps:
  - Remove the two nodes with the lowest frequencies from the queue. These nodes will represent the least common characters or groups of characters.
  - Create a new internal node by combining these two nodes as children. This node's frequency will be the sum of the two nodes' frequencies, representing a cumulative weight.
  - Insert this newly created node back into the priority queue, ordered by its frequency.

- When only one node remains in the priority queue, it becomes the root of the Huffman tree. This root node now connects all characters based on their frequencies, forming the complete Huffman tree.
- Starting from the root node, traverse the tree to assign a binary code to each character:
  - Assign "0" for the left branch and "1" for the right branch as you descend each level of the tree.
  - Continue assigning binary codes until every character has a unique code determined by its position in the tree.

6.12 Exercise

Given the following characters with their frequencies, execute Huffman's Encoding Algorithm step-by-step to build the Huffman tree and determine each character's binary code.

- Characters: A, B, C, D, E, F
- Frequencies: 5, 9, 12, 13, 16, 45

6.13 Solution

Step 1: Initialize Priority Queue

- Insert each character into the priority queue based on frequency.

$$\text{Priority Queue: } (A:5), (B:9), (C:12), (D:13), (E:16), (F:45)$$

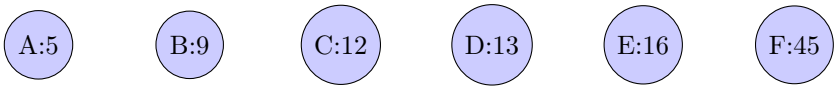

Figure 1: Initial Priority Queue with Characters and Frequencies

Step 2: First Iteration

- Remove nodes A (5) and B (9).
- Create a new node with frequency $5 + 9 = 14$.
- Insert the new node back into the priority queue.

$$\text{Priority Queue: } (C:12), (D:13), (E:16), (NewNode:14), (F:45)$$

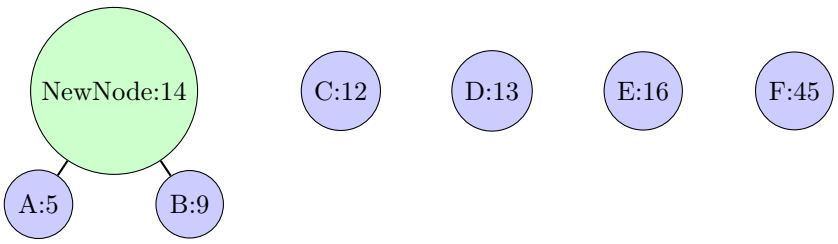

Figure 2: Merging A and B to create New Node with frequency 14

Step 3: Second Iteration

- Remove nodes C (12) and D (13).
- Create a new node with frequency $12 + 13 = 25$.
- Insert the new node back into the priority queue.

$$\text{Priority Queue: } (E : 16), (NewNode : 14), (NewNode : 25), (F : 45)$$

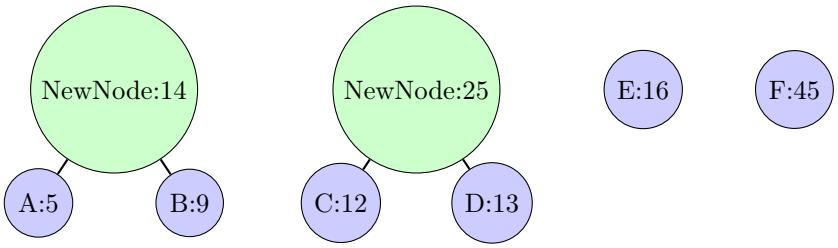

Figure 3: Merging C and D to create New Node with frequency 25

Step 4: Third Iteration

- Remove nodes E (16) and the first NewNode (14).
- Create a new node with frequency $16 + 14 = 30$.
- Insert the new node back into the priority queue.

$$\text{Priority Queue: } (NewNode : 25), (NewNode : 30), (F : 45)$$

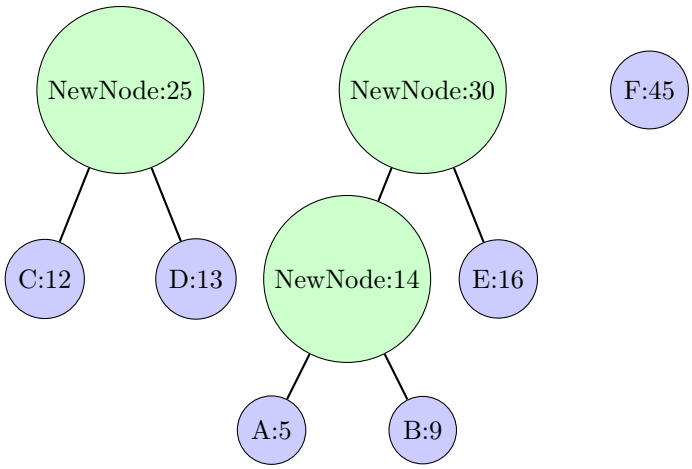

Figure 4: Merging E and Node (14) to create New Node with frequency 30

Step 5: Fourth Iteration

- Remove nodes NewNode (25) and NewNode (30).
- Create a new node with frequency $25 + 30 = 55$.
- Insert the new node back into the priority queue.

Priority Queue: $(F : 45), (NewNode : 55)$

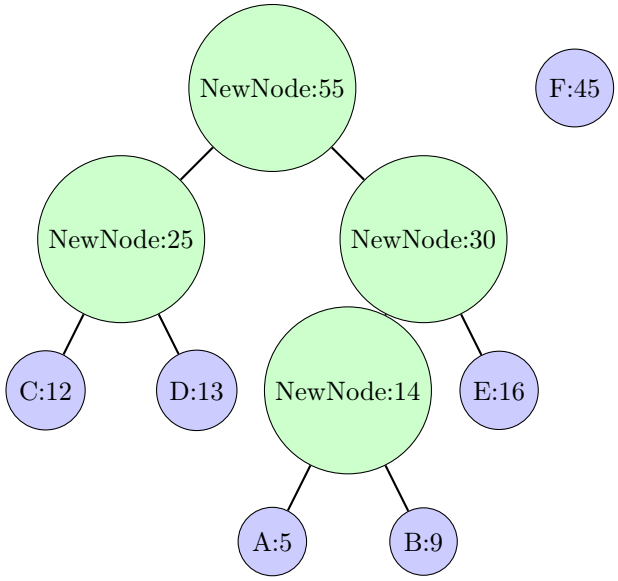

Figure 5: Merging Nodes 25 and 30 to create New Node with frequency 55

Step 6: Final Iteration

- Remove nodes F (45) and NewNode (55).
- Create a new root node with frequency $45 + 55 = 100$, completing the Huffman tree.

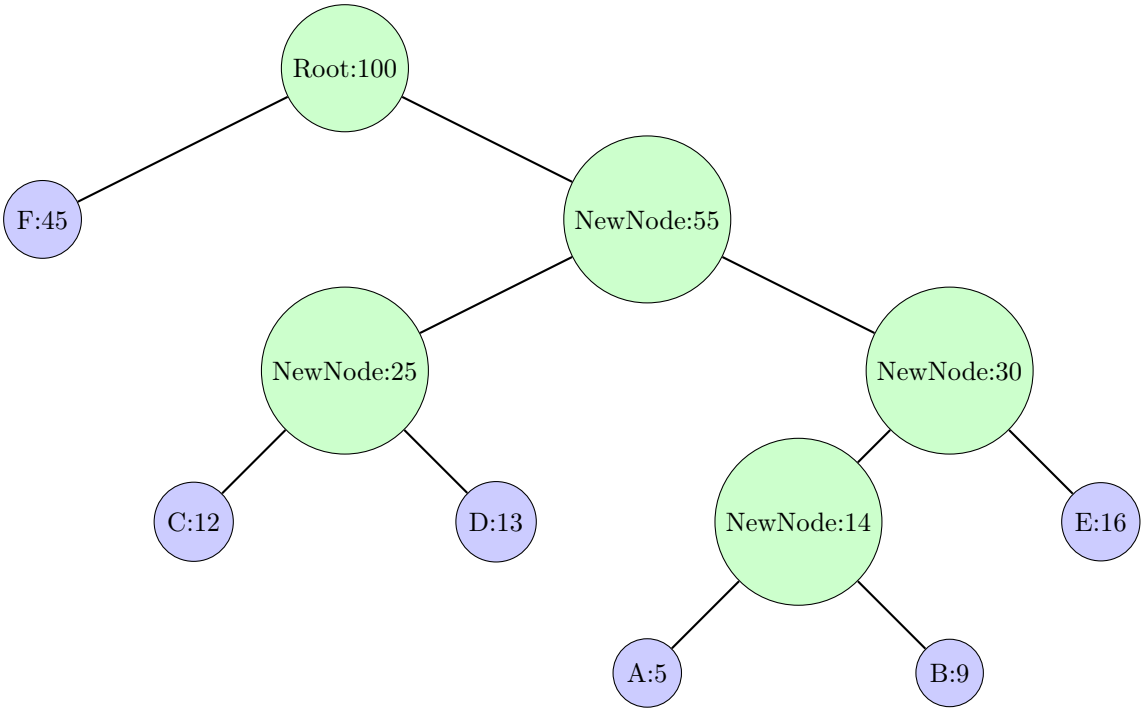

Figure 6: Final Huffman Tree with Root Node (100)

Step 7: Traverse the tree and assign a number

- Traverse the tree, assigning "0" to the left branches and "1" to the right branches

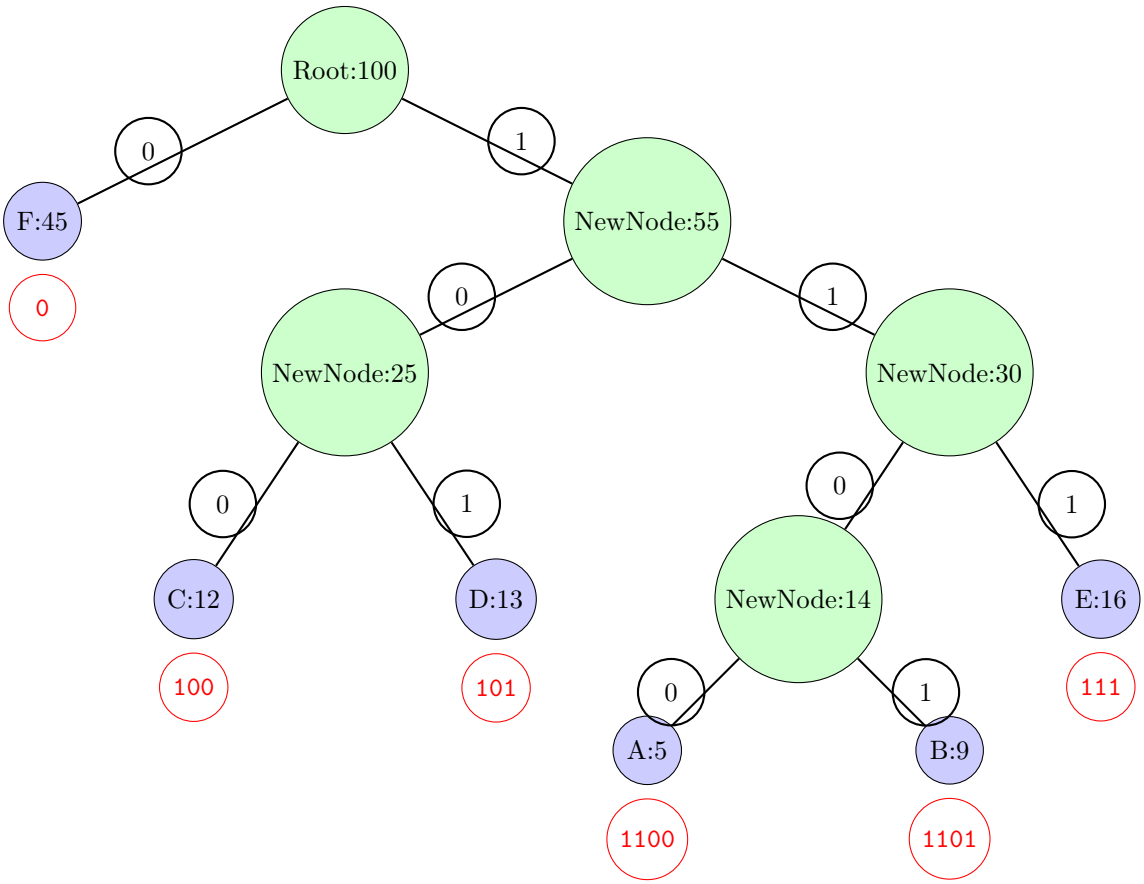

Figure 7: Final Huffman Tree with Binary Code Assignments

Step 8: Final Result

- F: 0
- C: 100
- D: 101
- E: 111
- A: 1100
- B: 1101

6.2 Greedy Algorithm Design

6.21 Problem

Given an array of non-negative integers, where each element represents the height of a vertical line on a coordinate plane, find two lines that together with the x-axis form a container that holds the most water. The container's height is determined by the shorter of the two lines, and its width is the distance between them.

For example, given the array [1, 8, 6, 2, 5, 4, 8, 3, 7], where each element represents the height of a bar, our task is to identify two bars that, when combined, form the container with the largest possible area.

The array [1, 8, 6, 2, 5, 4, 8, 3, 7] can be visualized as a set of vertical bars with heights as follows:

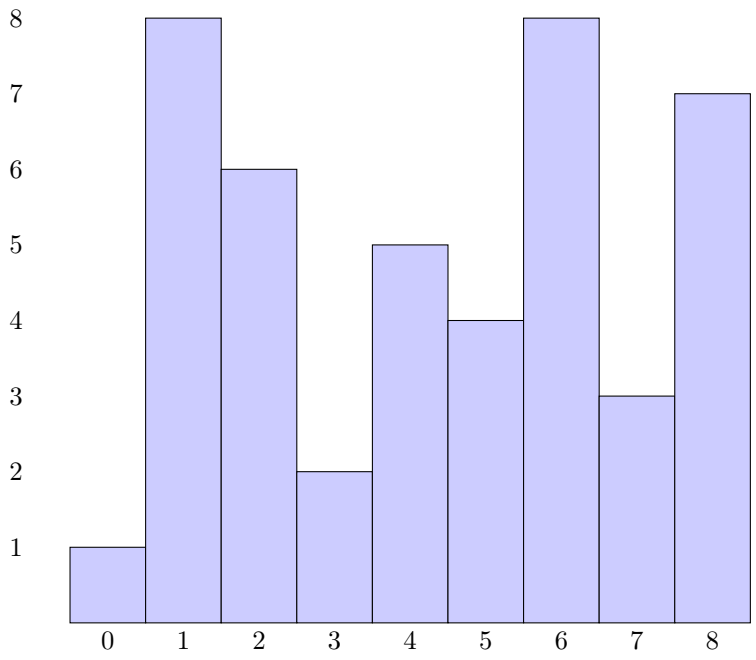

Figure 8: Initial Bar Heights for the Container Problem

In this setup:

- The x-axis represents the positions of the bars.
- The y-axis represents the height of each bar.

The objective is to find two bars that maximize the area of the container they form. The area of the container is calculated as Area = width × min(height[left], height[right]) where width is the distance between the two chosen bars, and height[left] and height[right] represent their respective heights.

6.22 Pseudocode and Explanation

Algorithm 2 MaxWaterContainer

Input : Array of integers *height*, representing heights of vertical lines
Output: Maximum area of water container formed by any two lines
Initialize *left* pointer at 0
Initialize *right* pointer at *length(height) - 1*
Initialize *max_ area* as 0
while *left* < *right* **do**
    Calculate *width* as *right - left*
    Calculate *current_ area* as *width* × *min(height[left], height[right])*
    Update *max_ area* to be the maximum of *max_ area* and *current_ area*
    if *height[left]* < *height[right]* **then**
        Move the *left* pointer to the right by 1
    else
        Move the *right* pointer to the left by 1
return *max_ area*

Step-by-Step Explanation of the Pseudocode

- The left pointer starts at the beginning of the array, and the right pointer starts at the end of the array (length - 1). These represent the two bars that could form a container.
- Start with maxarea set to 0, which will store the maximum area encountered.
- Continue the loop until left and right meet, maximizing the area by adjusting the pointers.
- Compute width as the distance between the left and right pointers.
- Calculate the area of the current container by multiplying the width by the shorter of the two heights at left and right.
- If the height[left] is smaller than the height[right], move the left pointer to the right to find a taller bar.
- If height[right] is smaller or equal to height[left], move the right pointer to the left.
- Return maxarea: After the loop, maxarea holds the largest area found.

6.23 Solution

Step 1: Initial Set up

- Array: [1, 8, 6, 2, 5, 4, 8, 3, 7]
- Left pointer: 0 as the height 1
- Right pointer: 8 as the height 7
- Max Area: 0

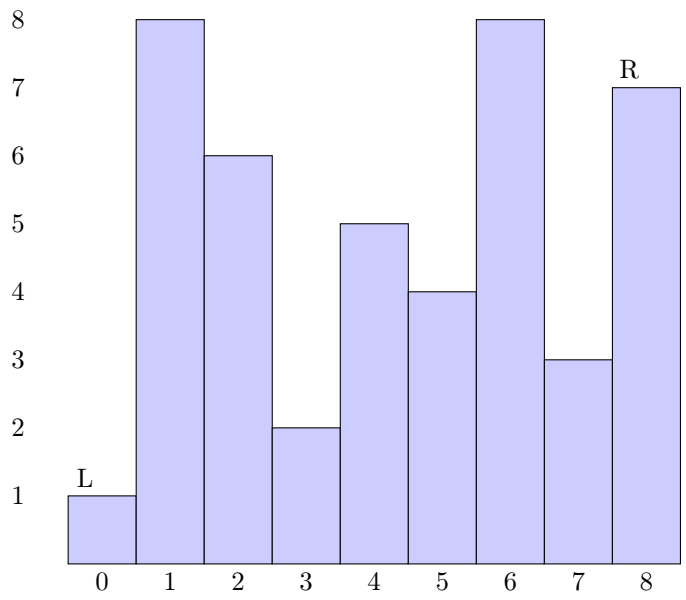

Figure 9: Step 1: Initial Setup with Left and Right Pointers

Step 2

- Calculate width: right - left = 8 - 0 = 0
- Calculate area: currentarea = width × min(height[left],height[right]) = 8 × min(1,7) = 8
- Update Max Area: maxarea = max(0, 8) = 8
- Move Pointers: Because height[left] < hetigh[right] so left pointer move right 1.

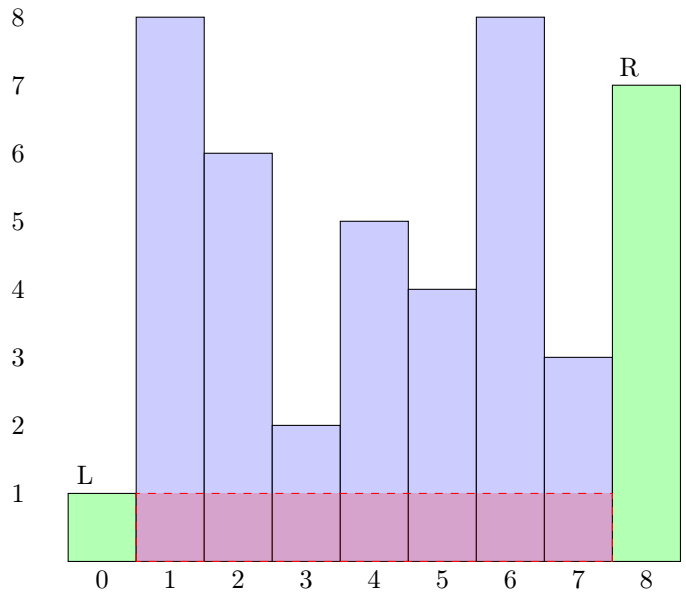

Figure 10: Step 2: Calculated Area = 8

Step 3

- Calculate width: right - left = 8 - 1 = 7
- Calculate area: current_area = width × min(height[left], height[right]) = 7 × min(8, 7) = 49
- Update Max Area: max_area = max(8, 49) = 49
- Move Pointers: Since height[left] > height[right], move right pointer left by 1.

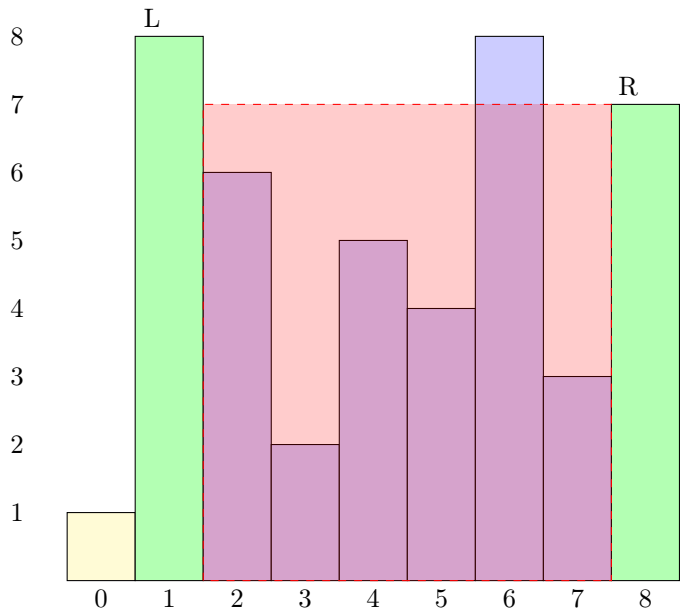

Figure 11: Step 3: Calculated Area = 49

Step 4

- Calculate width: right - left = 7 - 1 = 6
- Calculate area: current_area = width × min(height[left], height[right]) = 6 × min(8, 3) = 18
- Max Area remains 49
- Move Pointers: Since height[left] > height[right], move right pointer to the left by 1.

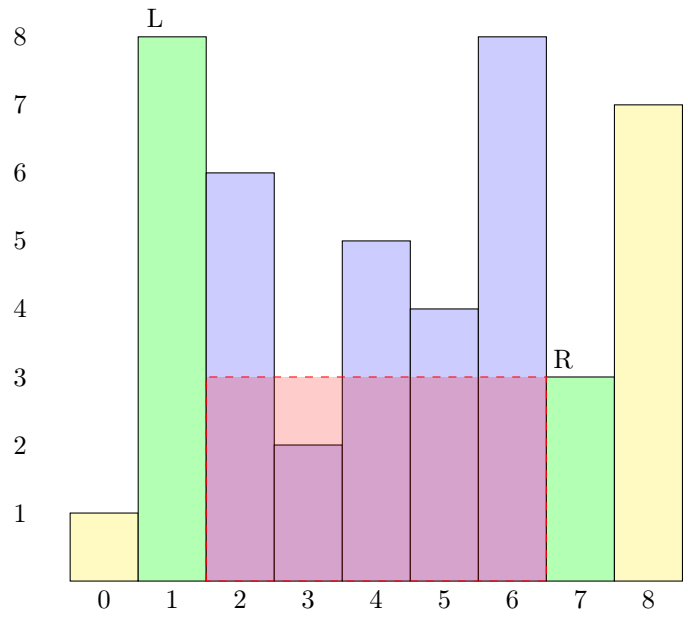

Figure 12: Step 4: Calculated Area = 18

Step 5

- Calculate width: right - left = 6 - 1 = 5
- Calculate area: current_area = width × min(height[left], height[right]) = 5 × min(8, 8) = 40
- Max Area remains 49
- Move Pointers: Since height[left] > height[right], move right pointer to the left by 1.

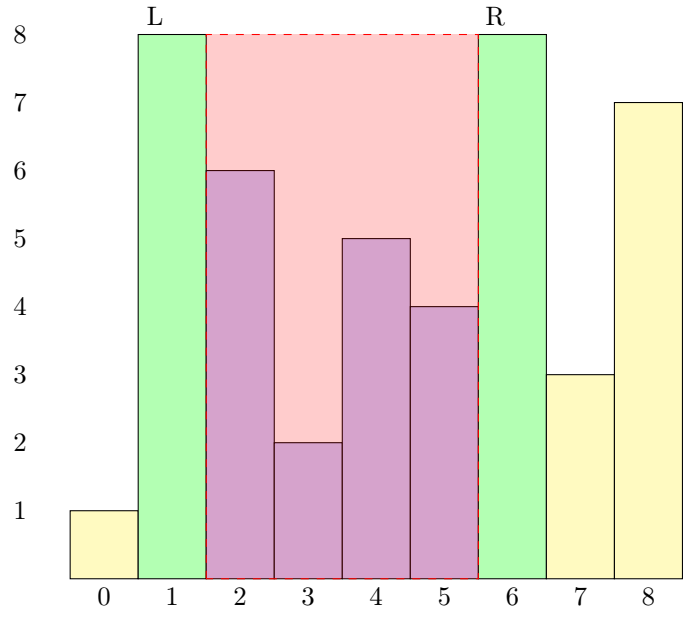

Figure 13: Step 5: Calculated Area = 40

Step 6

- Calculate width: right - left = 5 - 1 = 4
- Calculate area: current_area = width × min(height[left], height[right]) = 4 × min(8, 4) = 16
- Max Area remains 49
- Move Pointers: Since height[left] > height[right], move right pointer to the left by 1.

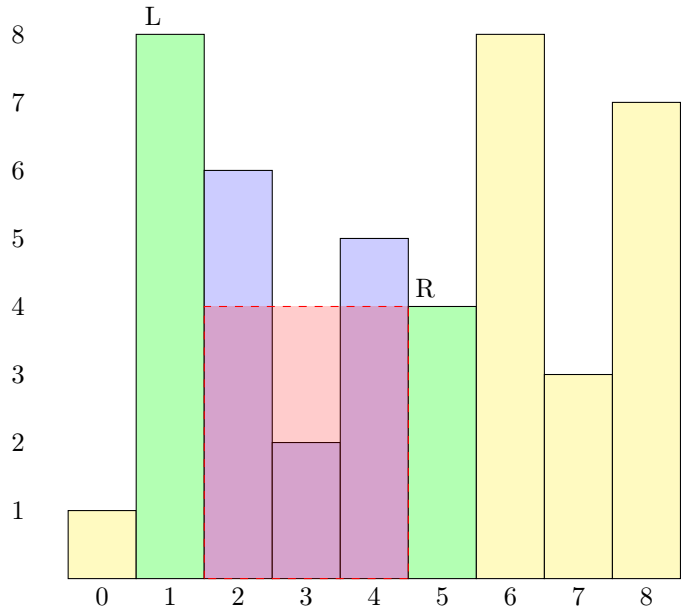

Figure 14: Step 6: Calculated Area = 16

Step 7

- Calculate width: right - left = 4 - 1 = 3
- Calculate area: current_area = width × min(height[left], height[right]) = 3 × min(8, 5) = 15
- Max Area remains 49
- Move Pointers: Since height[left] > height[right], move right pointer to the left by 1.

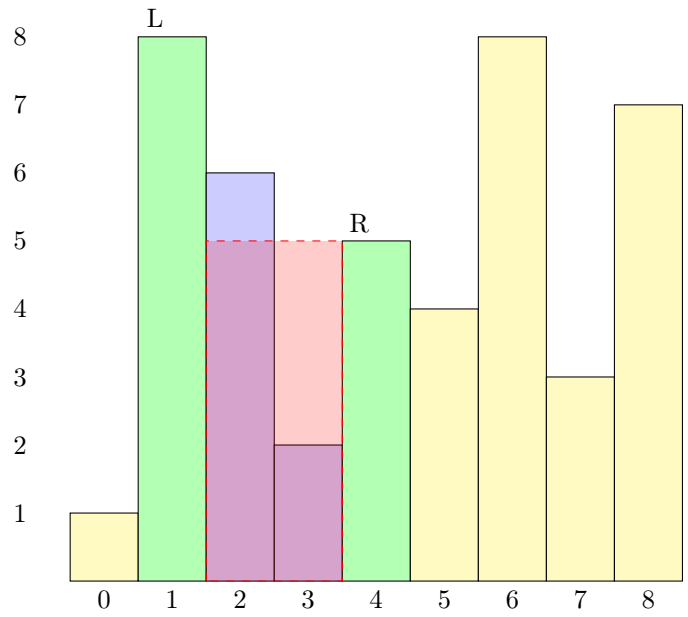

Figure 15: Step 7: Calculated Area = 15

Step 8

- Calculate width: right - left = 3 - 1 = 2
- Calculate area: current_area = width × min(height[left], height[right]) = 2 × min(8, 2) = 4
- Max Area remains 49
- Move Pointers: Since height[left] > height[right], move right pointer to the left by 1.

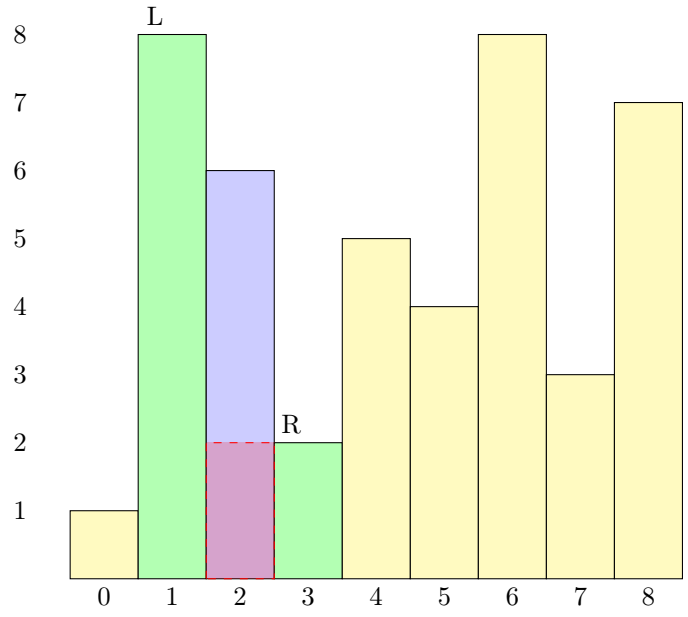

Figure 16: Step 8: Calculated Area = 4

Step 9

- Calculate width: right - lef = 2 - 1 = 1
- Calculate area: current_area = width × min(height[left], height[right]) = 1 × min(8, 6) = 6
- Max Area remains 49
- Move Pointers: Since height[left] > height[right], move right pointer to the left by 1.

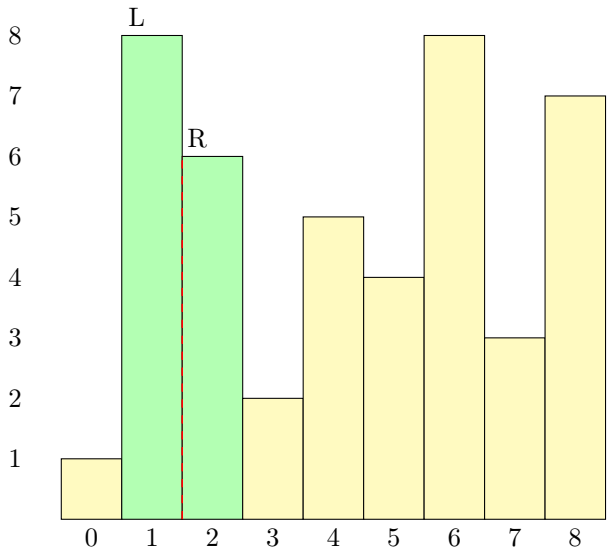

Figure 17: Step 9: Calculated Area = 6

Step 10: Final Result

- Left pointer meets right pointer, loop finish
- And the max area is 49

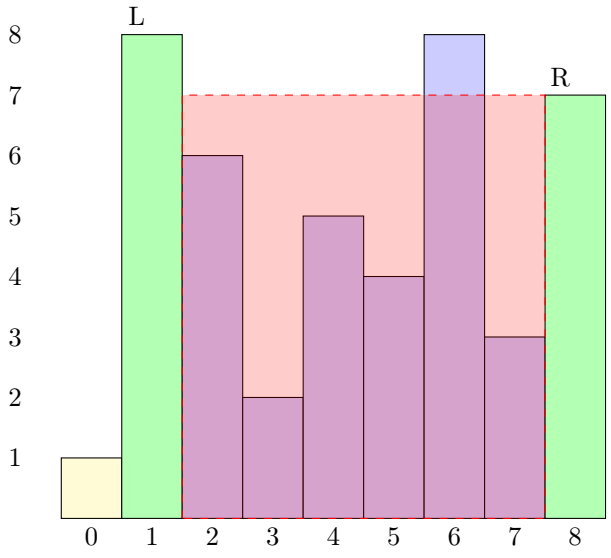

Figure 18: Step 10: Final Result = 49

6.24 Correctness

I use a two-pointer approach to solve the Container With Most Water problem. Starting with the pointers at the ends of the array, my idea was to gradually move the pointers inside to maximize the area between the two lines.

1. I realize that the shorter line always constrains the area. So, I moved the pointer to the shorter line, hoping to find a taller line to increase the area. Moving the taller line wouldn't help here, as it wouldn't remove the height limitation caused by the shorter line.

2. By using this method, I ensure that I don't miss any possible maximum areas. If the initial lines produce the largest area, I capture that immediately. As the pointers progress inside, I consider every pair of lines that could give a greater area.

3. Moving the shorter line's pointer also helps me eliminate pairs that aren't likely to produce the maximum area. This allows me to zero in on promising configurations without wasting time on less effective choices.

Through this approach, I'm confident that by the time the two pointers meet, I've considered all possible situations and found the maximum area. This makes the algorithm both correct and efficient.

6.25 Time Complexity

I chose a two-pointer approach for the Container With Most Water problem due to its efficiency.

Time Complexity:

- I move each pointer inside only once, making a single pass through the array. This gives an O(n) time complexity.

Space Complexity:

- I only use a few variables, so the space complexity is O(1), requiring no extra memory.

7 Dynamic Programming

7.1 Technique Definition

7.11 Basic Definition

7.12 Exercise

7.13 Solution

7.2 Edit-distance

7.21 Explanation

7.22 Exercise

7.23 Solution

7.3 Knapsack (both 0-1 and unrestrained)

7.31 Explanation

7.32 Exercise

7.33 Solution

7.4 Dynamic Programming Algorithm Design

7.41 Explanation

7.42 Exercise

7.43 Solution

8 Linear Programming

8.1 Technique Definition And Problem Specification

8.11 Basic Definition

8.12 Exercise

8.13 Solution

9 NP-Completeness

9.1 Definition (no problem/solution required)

9.11 Basic Definition

9.12 Exercise

9.13 Solution

9.2 SAT and 3SAT

9.21 Explanation

9.22 Exercise

9.23 Solution

9.3 Proving NP-Completeness)

9.31 Explanation

9.32 Exercise

9.33 Solution

9.4 Getting Around NP-Completeness (no problem/solution required)

9.41 Explanation

9.42 Exercise

9.43 Solution