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.

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

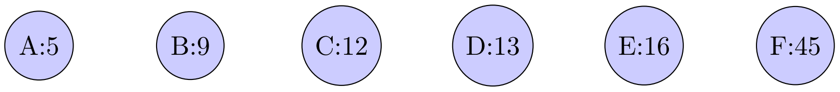

Figure 1: Initial Priority Queue with Characters and Frequencies

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.

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

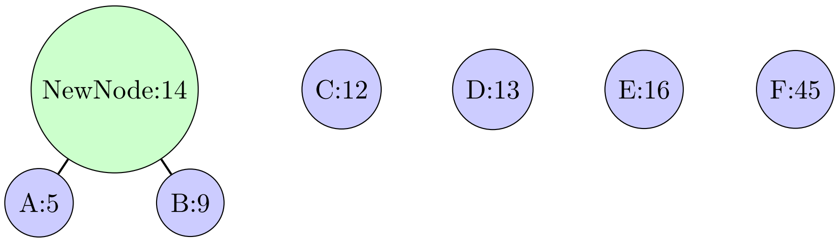

Figure 2: Merging A and B to create New Node with frequency 14

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.

Priority Queue: $(E : 16), (NewNode : 14), (NewNode : 25), (F : 45)$

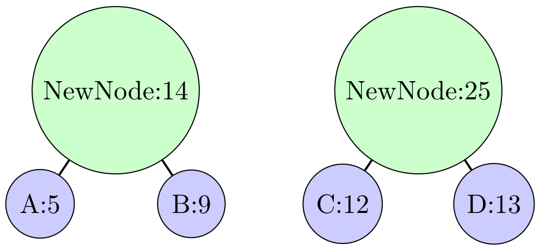

Figure 3: Merging C and D to create New Node with frequency 25

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.

Priority Queue: $(NewNode : 25), (NewNode : 30), (F : 45)$

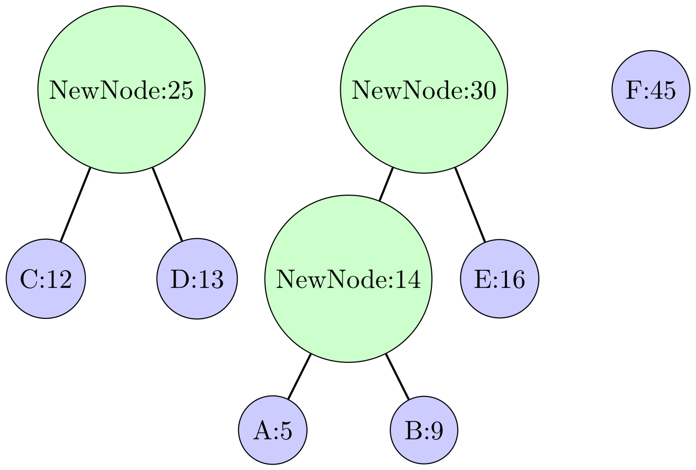

Figure 4: Merging E and Node (14) to create New Node with frequency 30

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

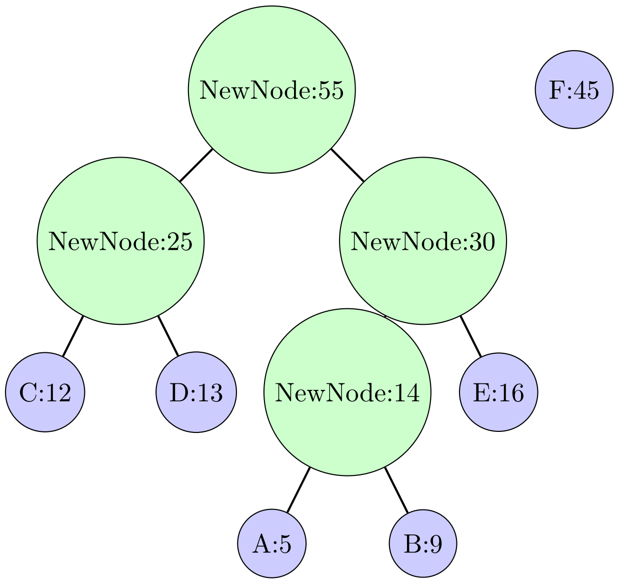

Figure 5: Merging Nodes 25 and 30 to create New Node with frequency 55

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.

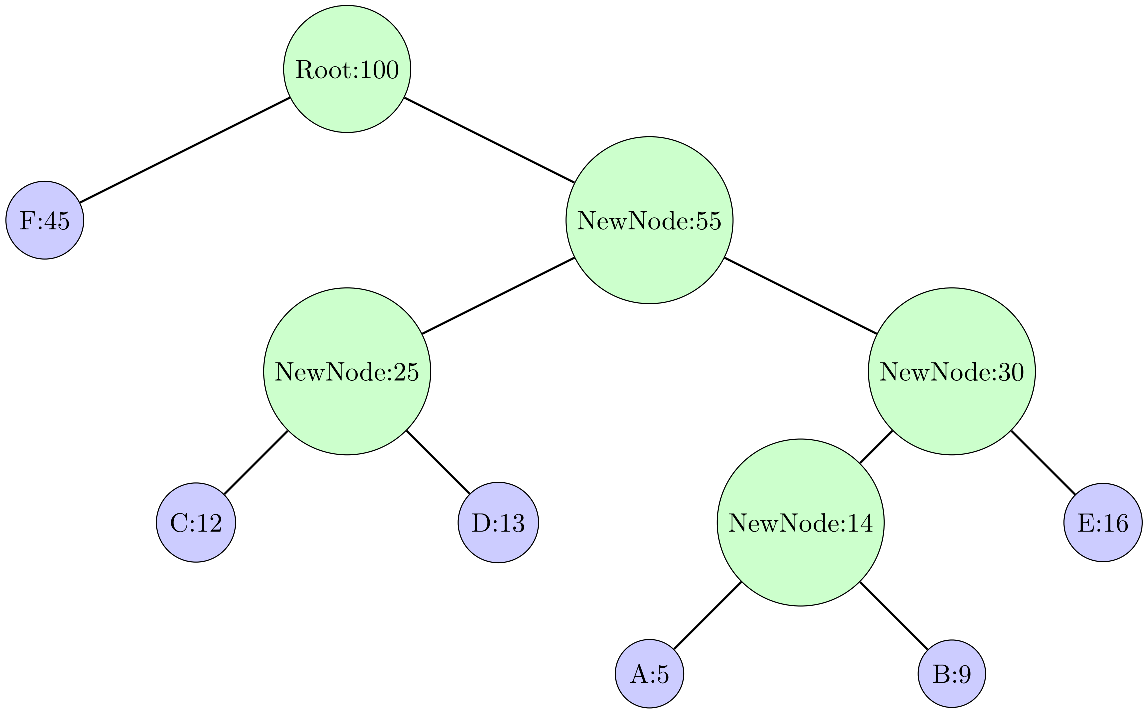

Figure 6: Final Huffman Tree with Root Node (100)

Step 7: Traverse the tree and assign a number

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

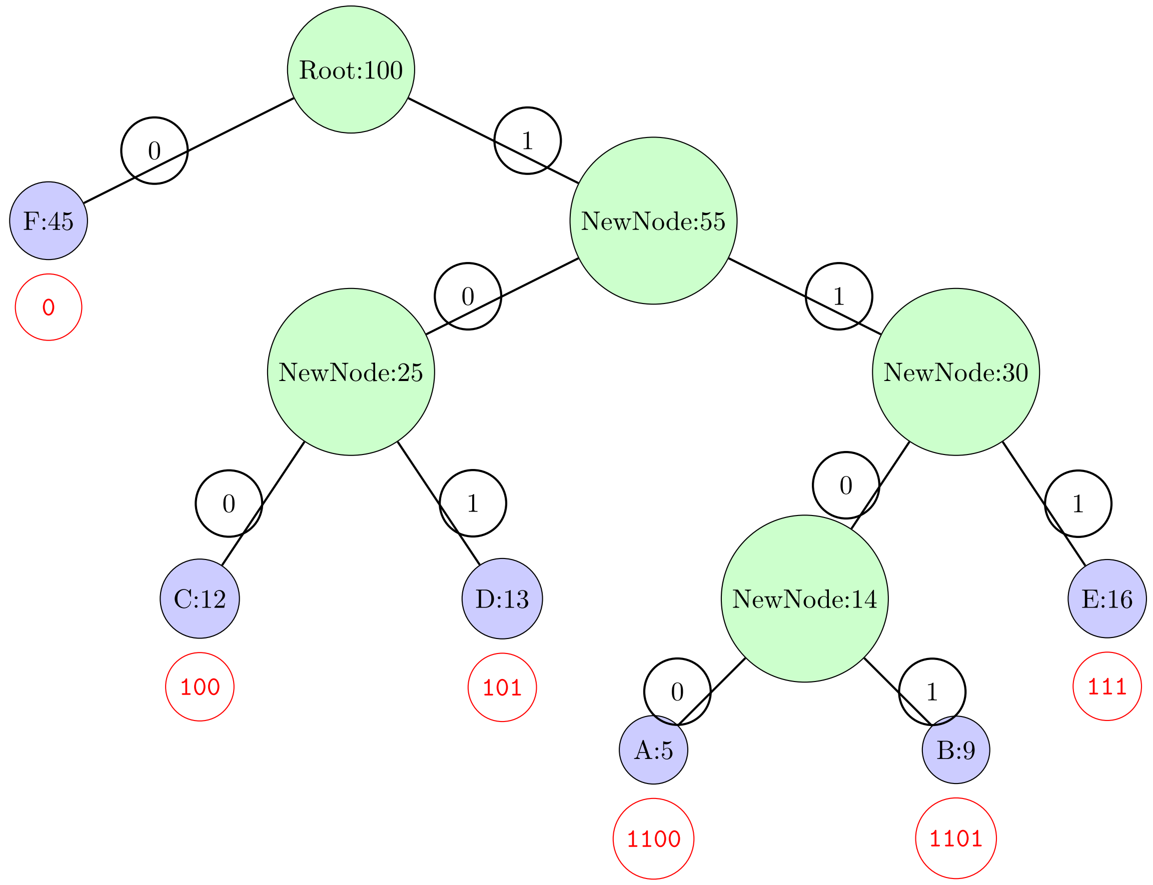

Figure 7: Final Huffman Tree with Binary Code Assignments

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:

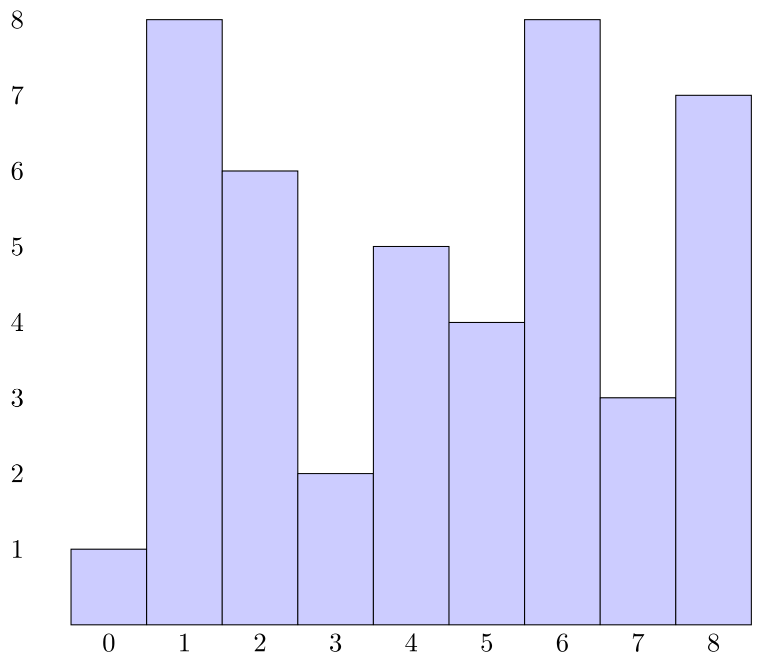

Figure 8: Initial Bar Heights for the Container Problem

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 $\times$ 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 $width \times min(height[left], height[right])$
    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

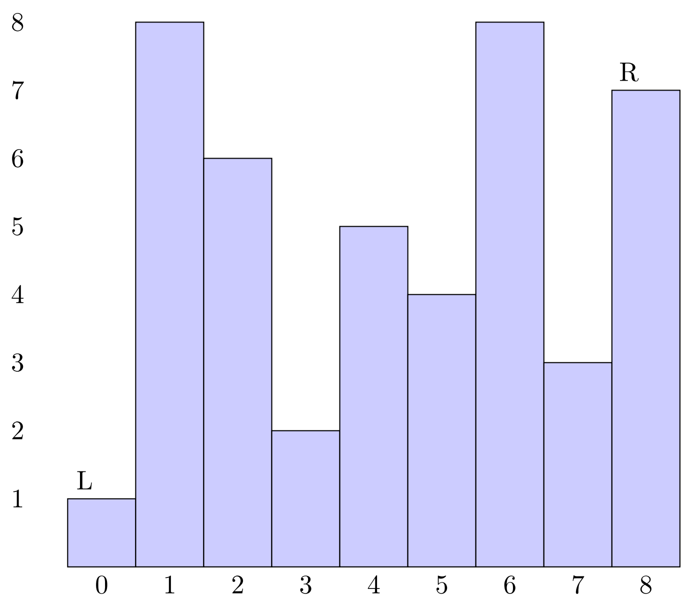

Figure 9: Step 1: Initial Setup with Left and Right Pointers

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.

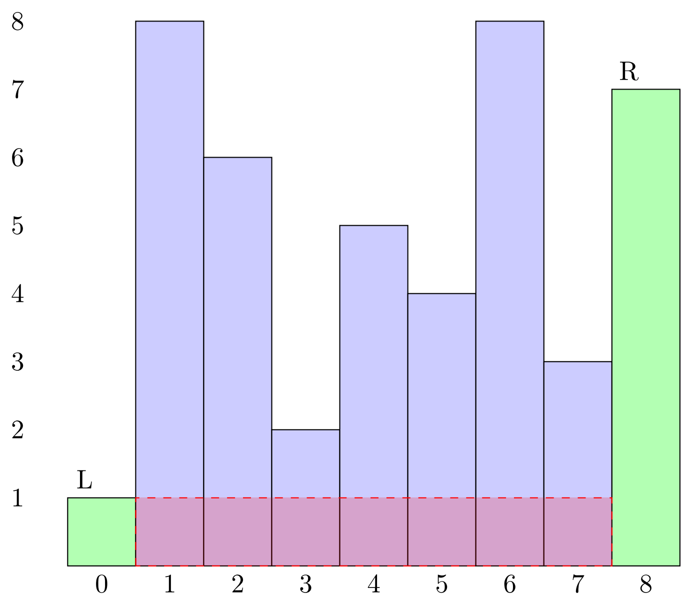

Figure 10: Step 2: Calculated Area = 8

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.

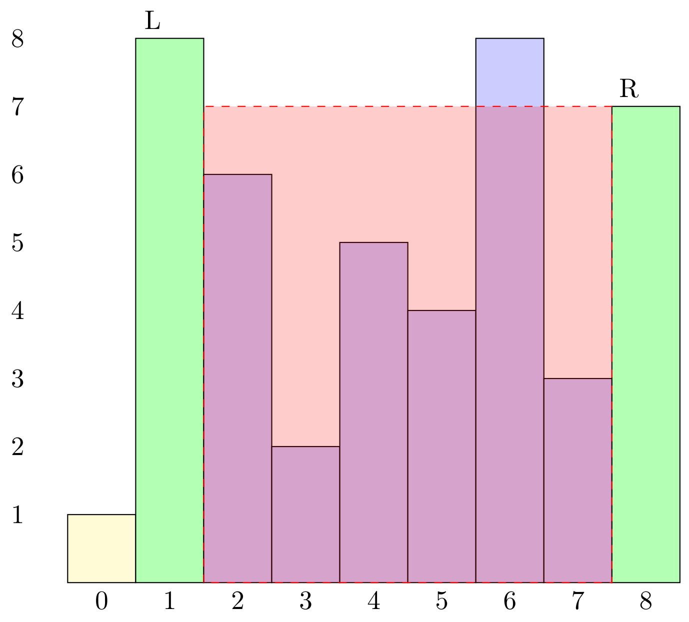

Figure 11: Step 3: Calculated Area = 49

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.

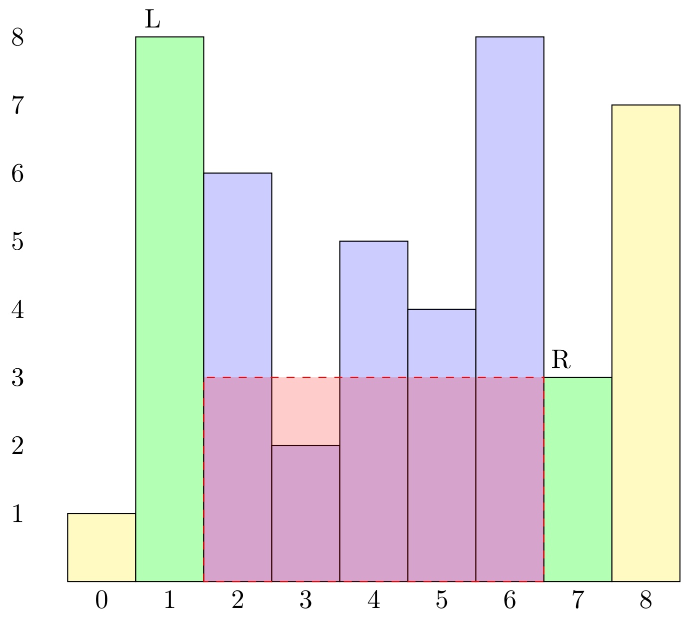

Figure 12: Step 4: Calculated Area = 18

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.

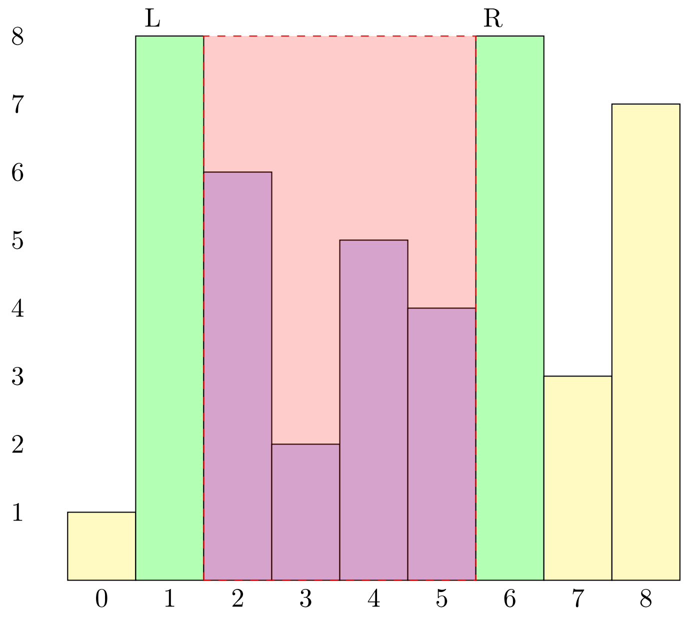

Figure 13: Step 5: Calculated Area = 40

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.

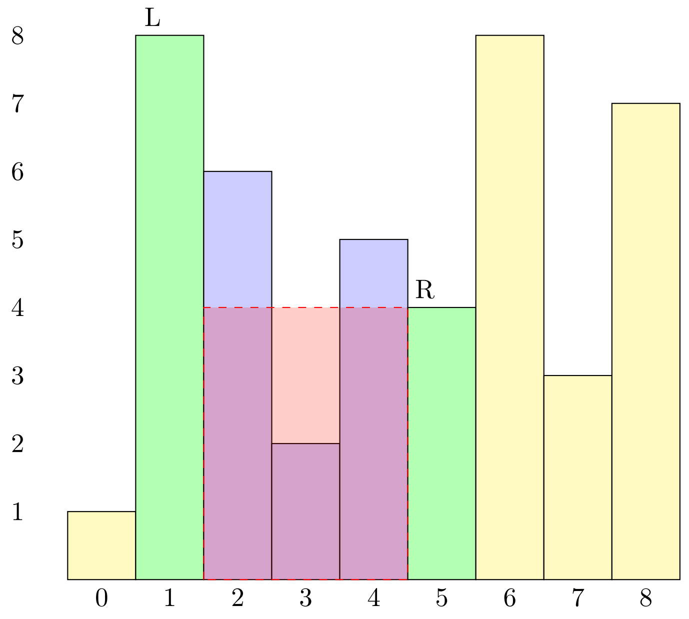

Figure 14: Step 6: Calculated Area = 16

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.

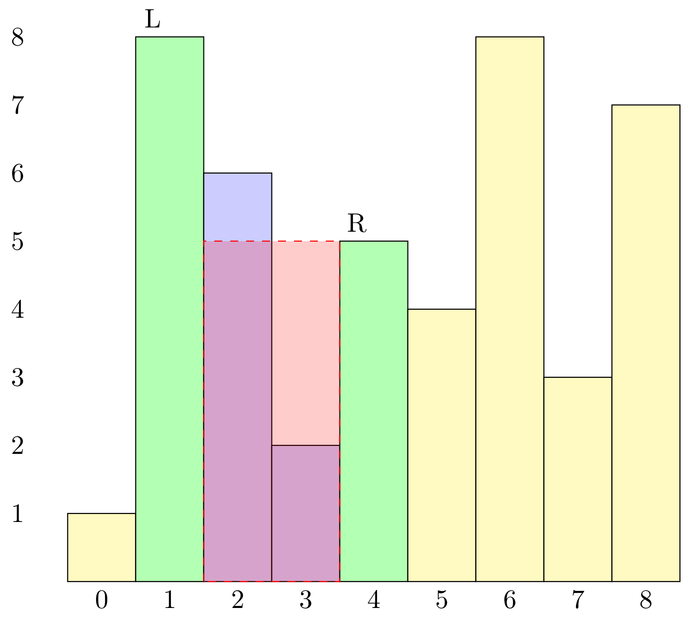

Figure 15: Step 7: Calculated Area = 15

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.

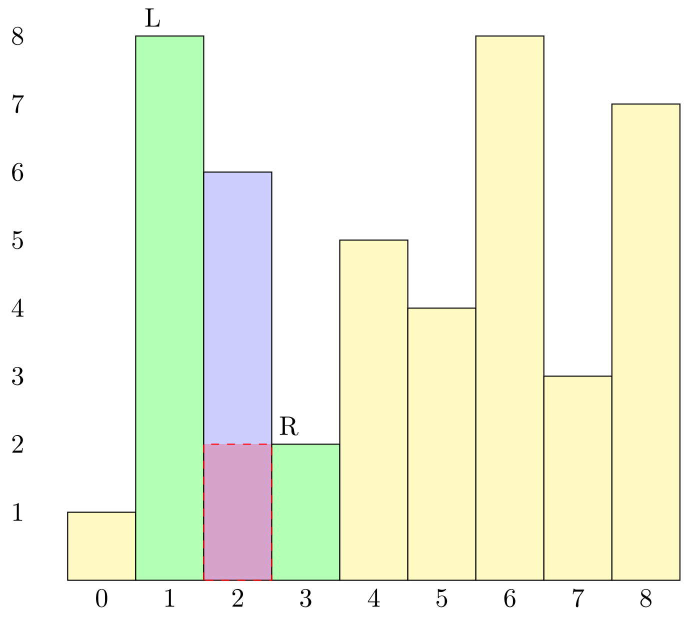

Figure 16: Step 8: Calculated Area = 4

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.

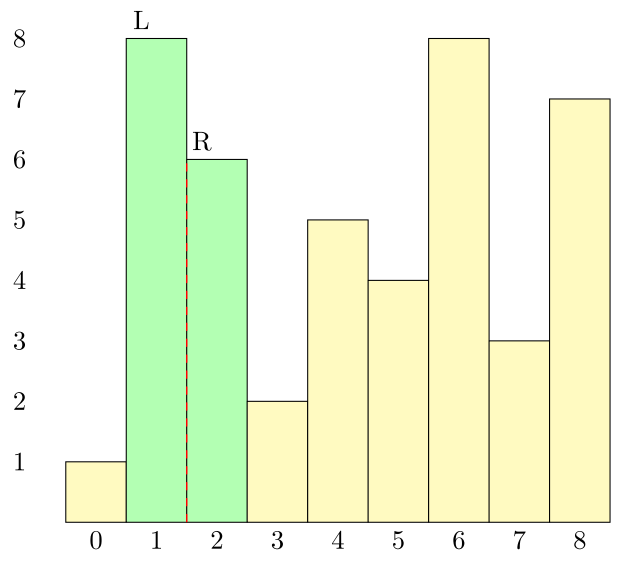

Figure 17: Step 9: Calculated Area = 6

Step 10: Final Result

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

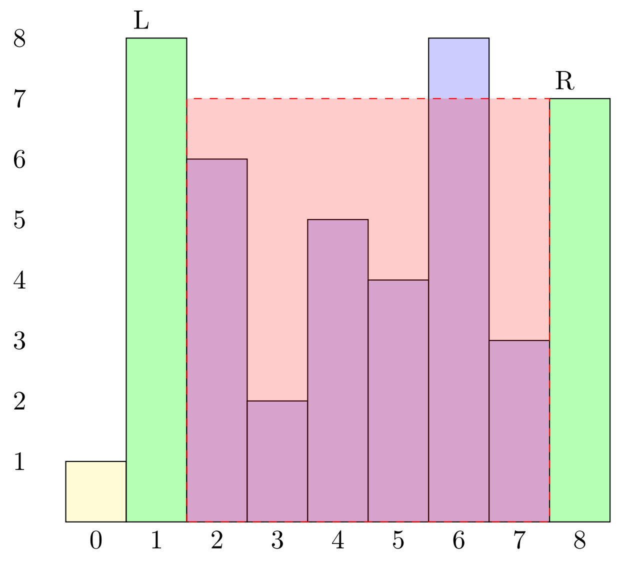

Figure 18: Step 10: Final Result = 49

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

Dynamic Programming (DP) is a method used to solve complex problems by breaking them down into smaller, overlapping subproblems. Each subproblem is solved only once, and its solution is stored. This avoids redundant calculations and speeds up the overall process. DP is particularly effective for optimization problems where decisions depend on previously computed results.
And DP has two main strategies which are the following:

Top - Down (Memoization)

- Start with the full problem, recursively solve smaller subproblems, and store their results.

Bottom - Up (Tabulation)

- Start with the smallest subproblems, iteratively build up solutions, and solve the full problem last.

7.12 Top - Down Example

Imagine I am planning a trip that needs to go through multiple cities and I want to try to minimize the total cost. Each city has multiple routes to other cities, and some routes go through specific in-between cities. Instead of repeating the cost of each possible route, start at the final destination, work backward to calculate the cheapest cost to get to that destination, and store the cost for each city.

7.13 Pseudocode

Algorithm 3 Fibonacci (Top-Down DP)

```
Input  : An integer $n$, where $n \geq 0$
Output: The $n$-th Fibonacci number
Define an array memo of size $n + 1$ and initialize all values to -1
Function Fibo($n$):
    if $n \leq 1$ then
        return $n$
    if memo[n] != -1 then
        return memo[n]
    memo[n] = Fibo($n - 1$) + Fibo($n - 2$)
    return memo[n]
Call Fibo($n$) and return the result
```

Step-by-Step Explanation of the Pseudocode

- Create an array memo to store solutions for already solved subproblems. Initialize all values to -1 to indicate they haven't been computed.
- If n is 0 or 1, return n directly, as these are the smallest subproblems.
- Before calculating Fibo(n), check if memo[n] has already been computed. If it has, return the stored value.
- If not, compute Fibo(n - 1) and Fibo(n - 2) recursively, store the result in memo[n], and return it.
- The initial call to Fibo(n) solves the full problem, using previously computed subproblems as needed.

7.14 Bottom - Up Example

Imagine I am in the process of renovating my house and have a choice of building materials to choose from. Each material has a price and a quality score. The first calculate and store each material's price-to-quality ratio to save time. Later, when I choose materials for my renovation, I can refer to this stored information to quickly find the best combo without recalculating.

7.15 Pseudocode

Algorithm 4 Fibonacci (Bottom-Up DP)

```
Input  : An integer $n$, where $n \geq 0$
Output: The $n$-th Fibonacci number
if $n \leq 1$ then
    return $n$
Initialize an array fib of size $n + 1$
Set fib[0] = 0
Set fib[1] = 1
for $i = 2$ to $n$ do
    fib[i] = fib[i - 1] + fib[i - 2]
return fib[n]
```

Step-by-Step Explanation of the Pseudocode

- If n is 0 or 1, return n directly. These are the smallest subproblems with known solutions.
- Create an array fib to store Fibonacci numbers up to the nth position. Set fib[0] to 0 and fib[1] to 1.
- For each i from 2 to n, calculate fib[i] by adding fib[i - 1] and fib[i - 2].
- The nth Fibonacci number is stored in fib[n] and returned.

7.16 Difference

Summary of Top-Down vs. Bottom-Up

Aspect	**Top-Down**	**Bottom-Up**
Approach	Recursive	Iterative
Storage	Stores results only when computed	Precomputes all subproblems
Execution Order	Starts with full problem	Starts with smallest subproblems
Example	Traveling through cities	Renovation materials

Table 1: Comparison of Top-Down and Bottom-Up Approaches in Dynamic Programming

7.2 Edit-distance

7.21 Explanation

Edit Distance, also known as Levenshtein Distance, measures how similar two strings are by calculating the minimum number of operations needed to convert one string into the other. Edit Distance is a common dynamic programming problem and particularly useful for applications in natural language processing, such as spell checkers, DNA sequence analysis, and text similarity measures.
The allowed operations are:

- Insertion: Add a character to one string.
- Deletion: Remove a character from one string.
- Substitution: Replace one character with another.

Imagine I am editing a rough draft of a document to match the final version. Every typo or mistake in the draft requires me to insert a missing word, delete an unnecessary word, or replace a misspelled one. The edit distance tells me the minimum number of edits required to transform the draft into the polished version, ensuring the two are identical.

Algorithm 5 Edit Distance Algorithm (Bottom-Up DP)

```
Input  : Two strings, $word1$ of length $m$ and $word2$ of length $n$
Output: Minimum edit distance to transform $word1$ into $word2$
Define a 2D array dp of size $(m+1) \times (n+1)$
for $i = 0$ to $m$ do
    dp[i][0] = $i$
for $j = 0$ to $n$ do
    dp[0][j] = $j$
for $i = 1$ to $m$ do
    for $j = 1$ to $n$ do
        if $word1[i-1] == word2[j-1]$ then
            dp[i][j] = dp[i-1][j-1]
        else
            dp[i][j] = $1 + \min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])$
return dp[m][n]
```

Step-by-Step Explanation of the Pseudocode

Initialize the DP Table

- The table dp[i][j] stores the minimum edit distance to convert the first i characters of word1 into the first j characters of word2.
- If one of the words is empty, the distance is the length of the other word.

Base Cases

- dp[i][0]: Transforming the first i characters of word1 into an empty word2 requires i deletions.
- dp[0][j]: Transforming an empty word1 into the first j characters of word2 requires j insertions.

Recursive Transitions

- If the characters word1[i-1] and word2[j-1] are equal, no operation is required. The value is carried over from dp[i-1][j-1].
- Otherwise, the value is 1 + min(...), where:
  - dp[i-1][j]: Represents a deletion in word1.
  - dp[i][j-1]: Represents an insertion in word1.
  - dp[i-1][j-1]: Represents a substitution.

Final Result

- The value at dp[m][n] gives the minimum number of edits required to transform word1 into word2.

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