Synthesis Assignment 3
Erdun E
December 2, 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.

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

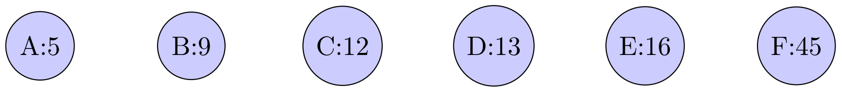

Figure 1: Initial Priority Queue with Characters and Frequencies

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.

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

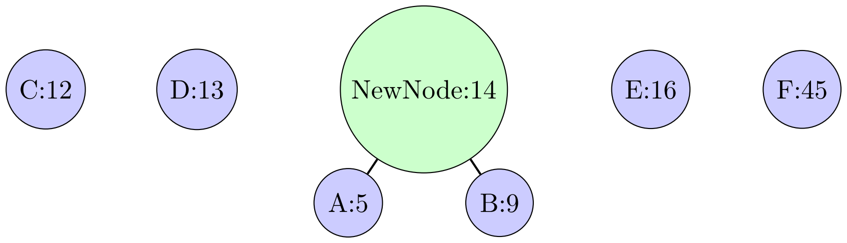

Figure 2: Merging A and B to create New Node with frequency 14

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

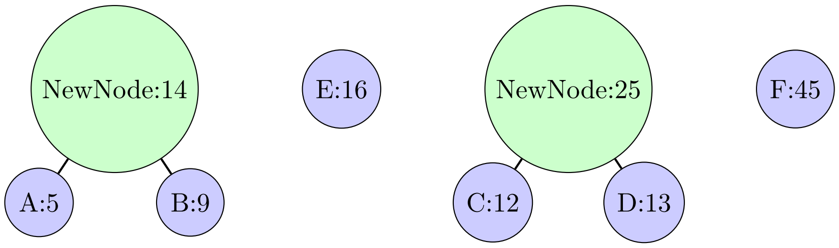

Figure 3: Merging C and D to create New Node with frequency 25

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

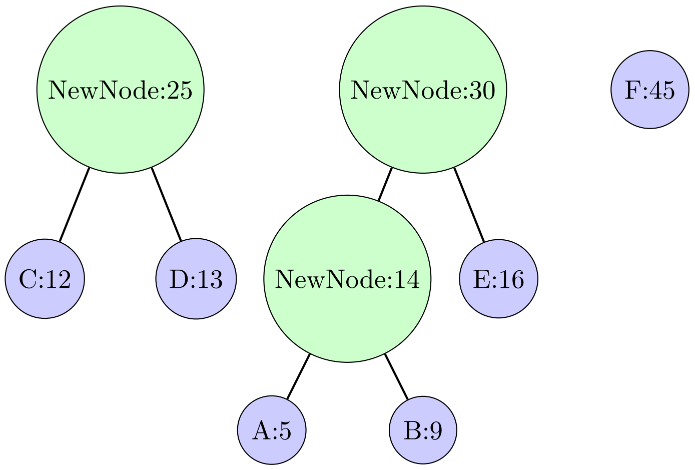

Figure 4: Merging E and Node (14) to create New Node with frequency 30

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

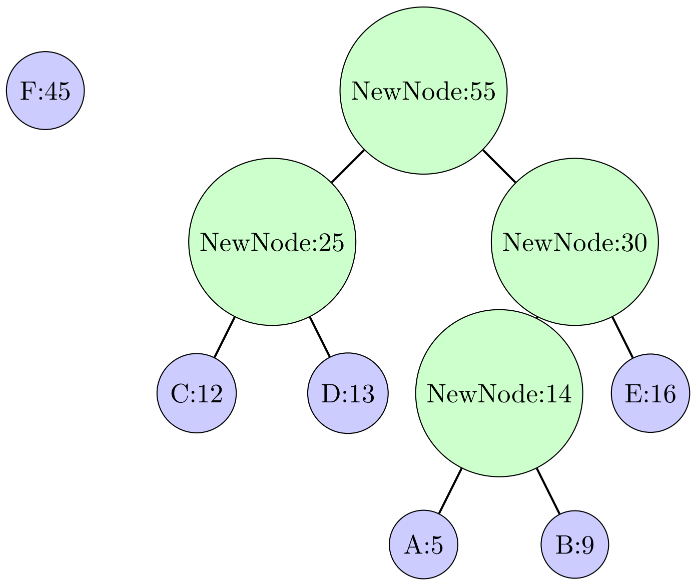

Figure 5: Merging Nodes 25 and 30 to create New Node with frequency 55

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.

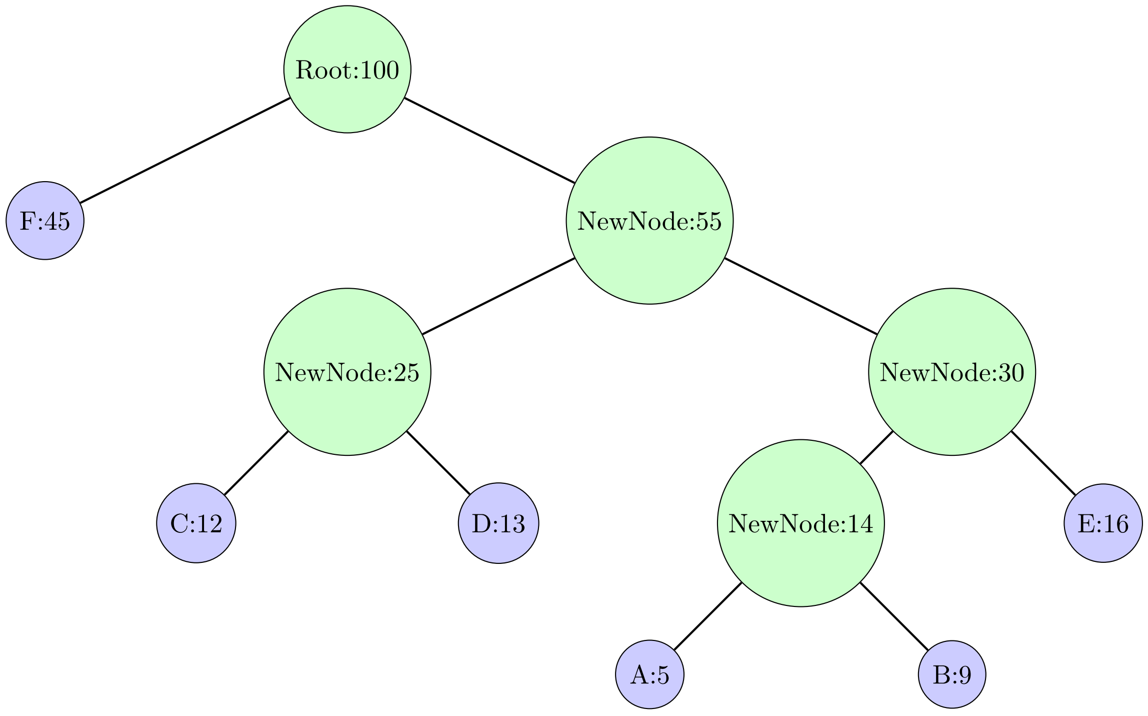

Figure 6: Final Huffman Tree with Root Node (100)

Step 7: Traverse the tree and assign a number

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

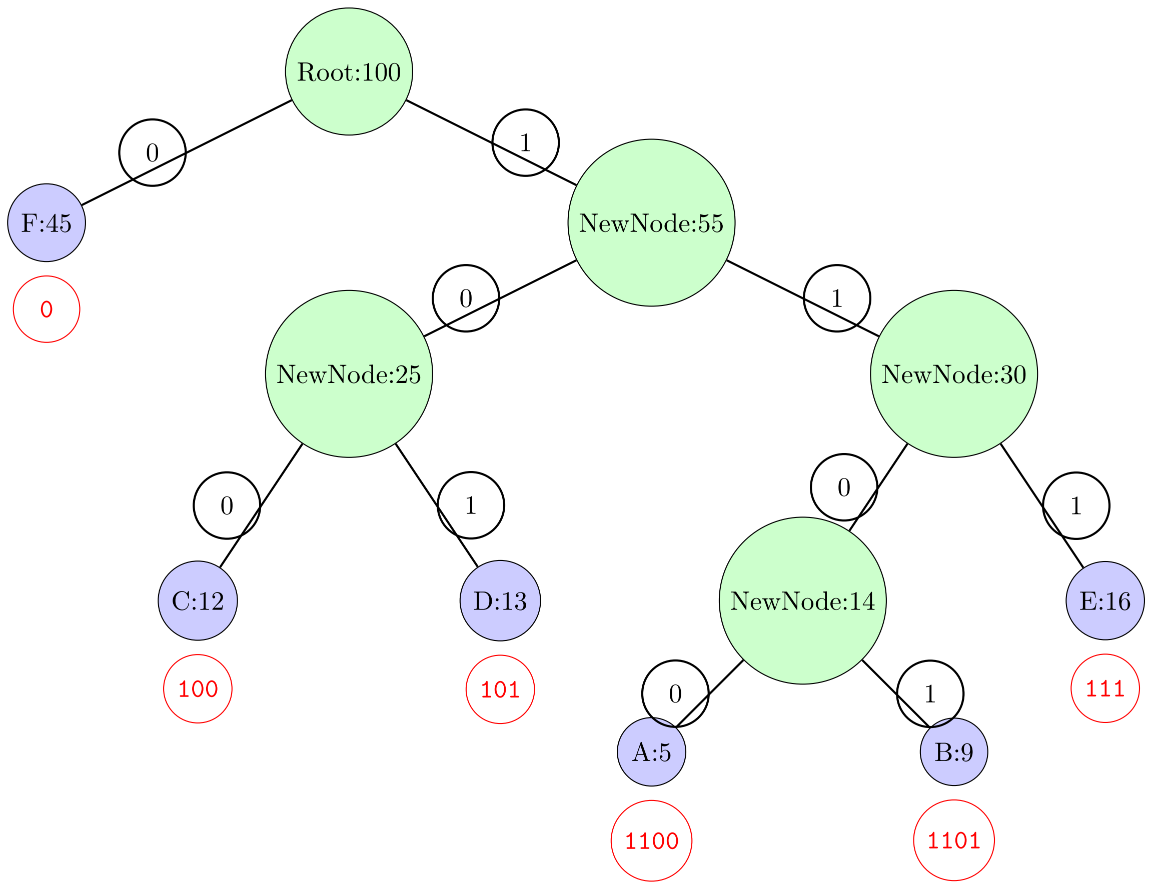

Figure 7: Final Huffman Tree with Binary Code Assignments

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:

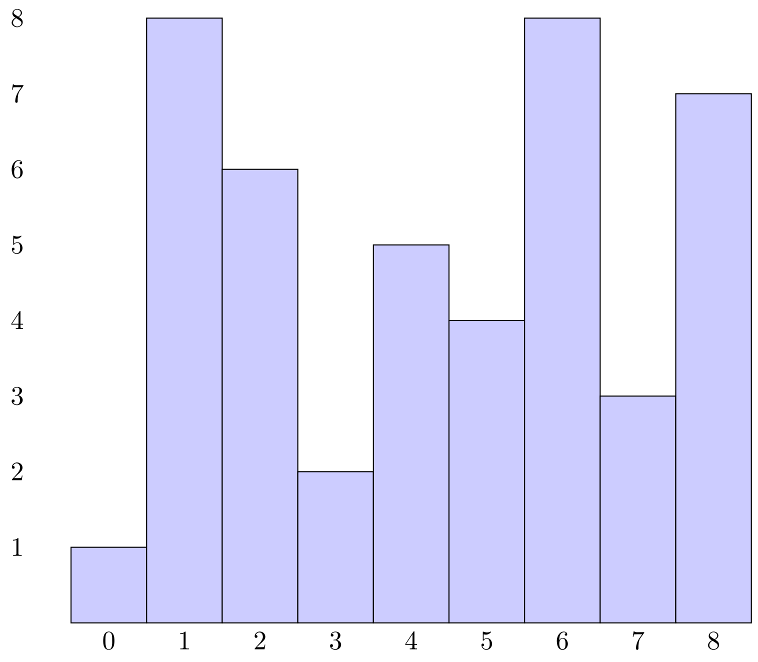

Figure 8: Initial Bar Heights for the Container Problem

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 × 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

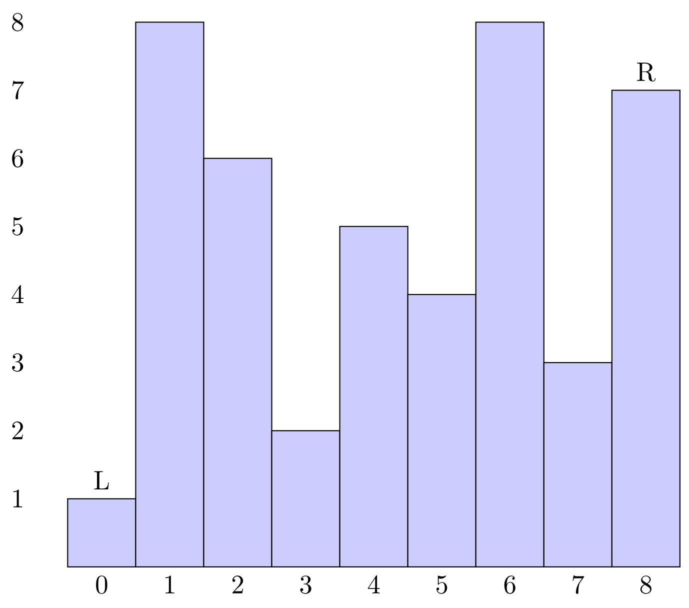

Figure 9: Step 1: Initial Setup with Left and Right Pointers

Step 2

- Calculate width: right - left = 8 - 0 = 8
- 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.

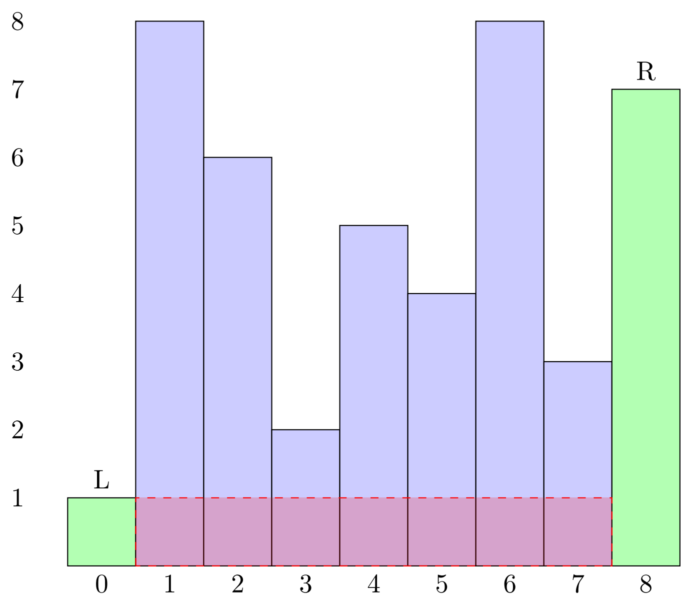

Figure 10: Step 2: Calculated Area = 8

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.

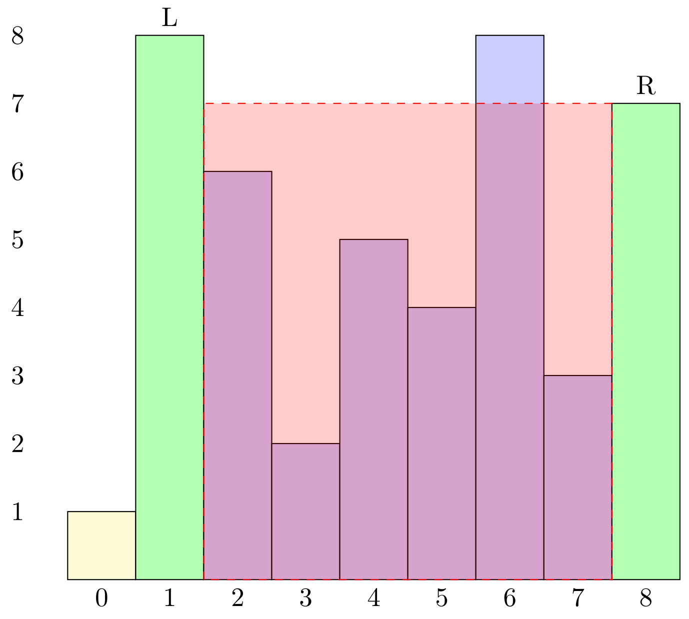

Figure 11: Step 3: Calculated Area = 49

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.

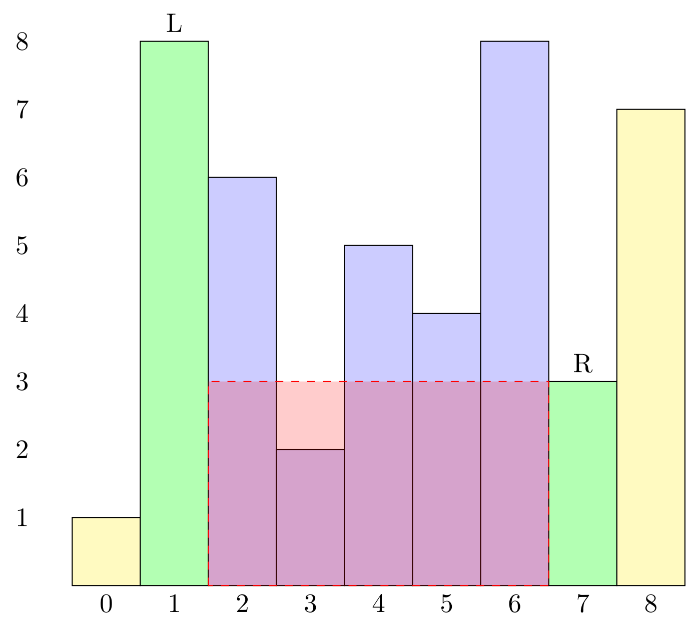

Figure 12: Step 4: Calculated Area = 18

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.

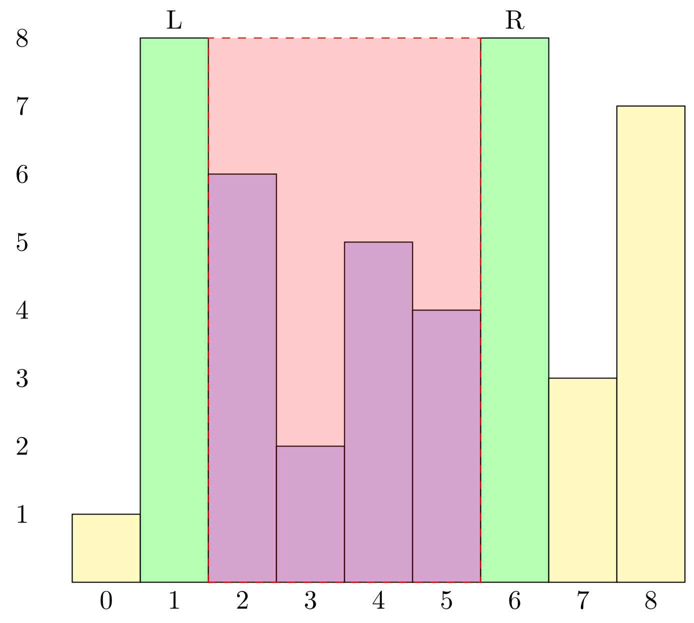

Figure 13: Step 5: Calculated Area = 40

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.

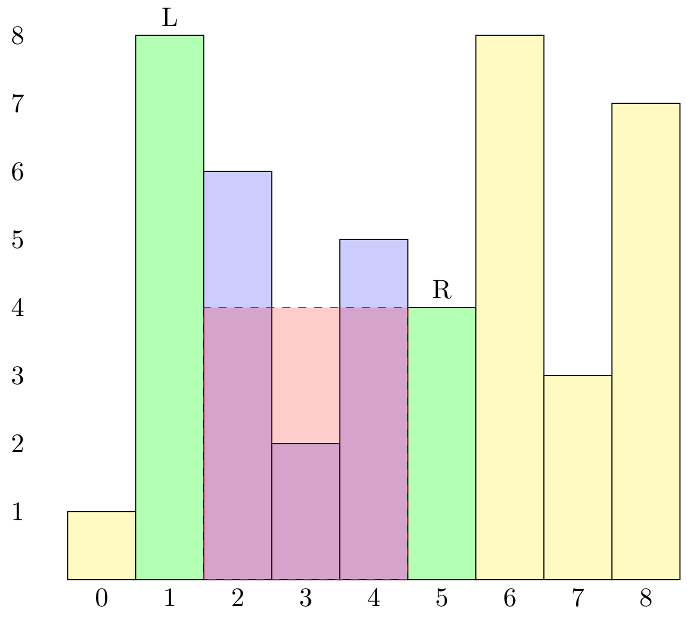

Figure 14: Step 6: Calculated Area = 16

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.

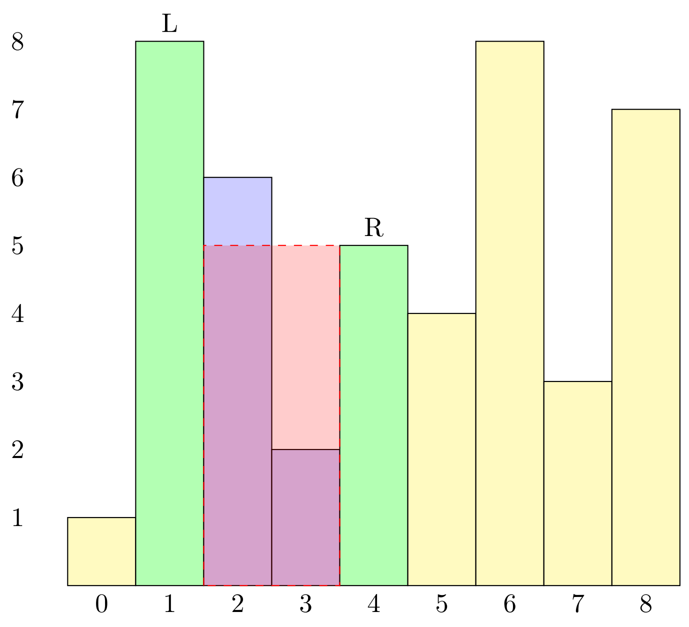

Figure 15: Step 7: Calculated Area = 15

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.

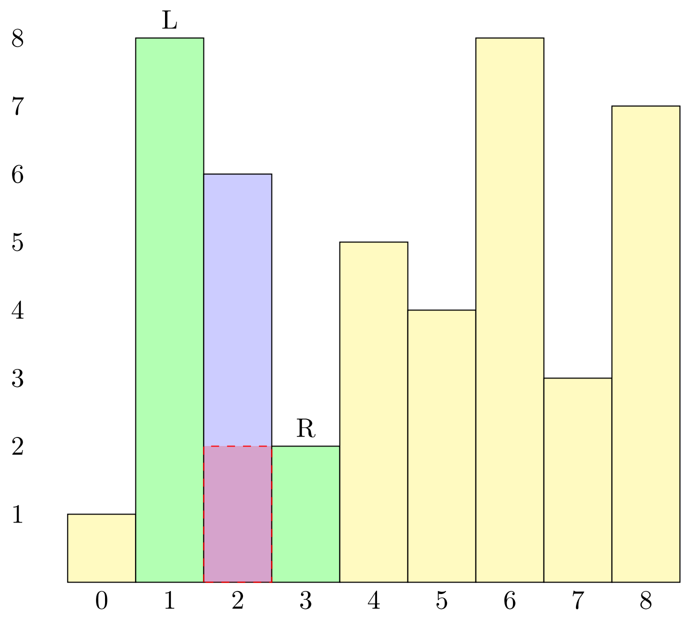

Figure 16: Step 8: Calculated Area = 4

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.

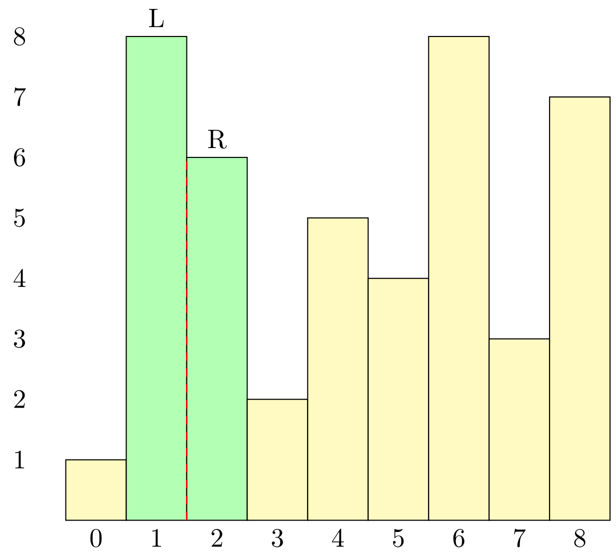

Figure 17: Step 9: Calculated Area = 6

Step 10: Final Result

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

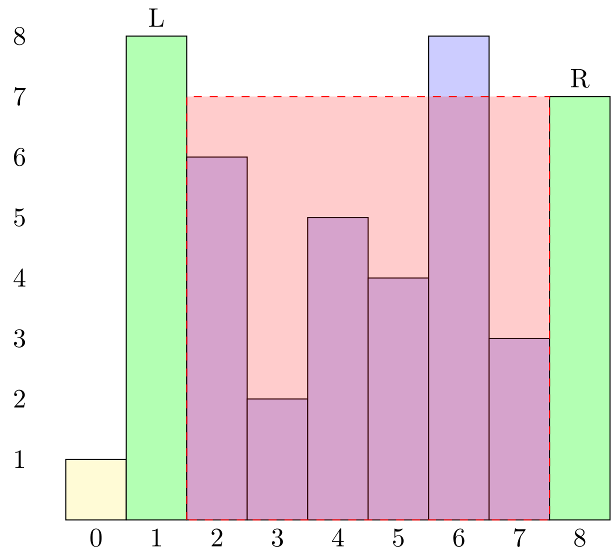

Figure 18: Step 10: Final Result = 49

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

Using the bottom-up dynamic programming approach, calculate the minimum number of operations required to convert "editing" into "distance" by step-by-step execution of the algorithm.

7.23 Solution

Step 1: Initialization

- Initialize the table with the base cases. The first row and first column represent the costs of converting an empty string into prefixes of the other string.

		d	**i**	**s**	**t**	**a**	**n**	**c**	**e**
	0	1	2	3	4	5	6	7	8
e	1								
d	2								
i	3								
t	4								
i	5								
n	6								
g	7								

Table 2: Step 1: Initialization of Base Cases

Step 2: Comparing "e" with "d"

- Compute the cost for converting e to d. The values in the first row and first column remain unchanged.
- dp[1][1] = 1 + min(dp[0][1], dp[1][0], dp[0][0])
- dp[1][1] = 1 + min(1, 1, 0) = 1

		d	**i**	**s**	**t**	**a**	**n**	**c**	**e**
	0	1	2	3	4	5	6	7	8
e	1	1							
d	2								
i	3								
t	4								
i	5								
n	6								
g	7								

Table 3: Step 2: Comparing "e" with "d"

Step 3: Comparing "e" with "i"

- Compute the cost for converting "e" to "i".
- dp[1][2] = 1 + min(dp[0][2], dp[1][1], dp[0][1])
- dp[1][2] = 1 + min(2, 1, 1) = 2

		d	**i**	**s**	**t**	**a**	**n**	**c**	**e**
	0	1	2	3	4	5	6	7	8
e	1	1	2						
d	2								
i	3								
t	4								
i	5								
n	6								
g	7								

Table 4: Step 3: Comparing "e" with "i"

Step 4:

- Compute the cost for converting "e" to "s".
- dp[1][3] = 1 + min(dp[0][3], dp[1][2], dp[0][2])
- dp[1][3] = 1 + min(3, 2, 2) = 3

		d	**i**	**s**	**t**	**a**	n	**c**	**e**
	0	1	2	3	4	5	6	7	8
e	1	1	2	3					
d	2								
i	3								
t	4								
i	5								
n	6								
g	7								

Table 5: Step 4: Comparing "e" with "s"

Step 5: Comparing "e" with "t"

- Compute the cost for converting "e" to "t".
- dp[1][4] = 1 + min(dp[0][4], dp[1][3], dp[0][3])
- dp[1][4] = 1 + min(4, 3, 3) = 4

		d	**i**	**s**	**t**	**a**	**n**	**c**	**e**
	0	1	2	3	4	5	6	7	8
e	1	1	2	3	4				
d	2								
i	3								
t	4								
i	5								
n	6								
g	7								

Table 6: Step 5: Comparing "e" with "t"

Step 6: Comparing "e" with "a"

- Compute the cost for converting "e" to "a".
- dp[1][5] = 1 + min(dp[0][5], dp[1][4], dp[0][4])
- dp[1][5] = 1 + min(5, 4, 4) = 5

		d	**i**	**s**	**t**	**a**	**n**	**c**	**e**
	0	1	2	3	4	5	6	7	8
e	1	1	2	3	4	5			
d	2								
i	3								
t	4								
i	5								
n	6								
g	7								

Table 7: Step 6: Comparing "e" with "a"

Step 7: Comparing "e" with "n"

- Compute the cost for converting "e" to "n".
- dp[1][6] = 1 + min(dp[0][6], dp[1][5], dp[0][5])
- dp[1][6] = 1 + min(6, 5, 5) = 6

		d	**i**	**s**	**t**	**a**	**n**	**c**	**e**
	0	1	2	3	4	5	6	7	8
e	1	1	2	3	4	5	6		
d	2								
i	3								
t	4								
i	5								
n	6								
g	7								

Table 8: Step 7: Comparing "e" with "n"

Step 8: Comparing "e" with "c"

- Compute the cost for converting "e" to "c".
- dp[1][7] = 1 + min(dp[0][7], dp[1][6], dp[0][6])
- dp[1][7] = 1 + min(7, 6, 6) = 7

		d	**i**	**s**	**t**	**a**	n	**c**	**e**
	0	1	2	3	4	5	6	7	8
e	1	1	2	3	4	5	6	7	
d	2								
i	3								
t	4								
i	5								
n	6								
g	7								

Table 9: Step 8: Comparing "e" with "c"

Step 9: Comparing "e" with "e"

- Compute the cost for converting "e" to "e".
- Since they are the same, no operation is needed.
- dp[1][8] = dp[0][7] = 7

		d	**i**	**s**	**t**	**a**	**n**	**c**	**e**
	0	1	2	3	4	5	6	7	8
e	1	1	2	3	4	5	6	7	7
d	2								
i	3								
t	4								
i	5								
n	6								
g	7								

Table 10: Step 9: Comparing "e" with "e"

Step 10: Fill out the second row for "d"

- Compare "d" with each character in "distance"
- dp[2][1]:
  - Comparing "d" with "d".
  - Characters are the same, so no operation is needed
  - dp[2][1] = dp[1][0] = 1
- dp[2][2]:
  - Comparing "d" with "i"
  - dp[2][2] = 1 + min(dp[1][2], dp[2][1], dp[1][1])
  - dp[2][2] = 1 + min(2, 1, 1) = 2

- dp[2][3]:
  - Comparing "d" with "s".
  - dp[2][3] = 1 + min(dp[1][3], dp[2][2], dp[1][2])
  - dp[2][3] = 1 + min(3, 2, 2) = 3
- dp[2][4]:
  - Comparing "d" with "t".
  - dp[2][4] = 1 + min(dp[1][4], dp[2][3], dp[1][3])
  - dp[2][4] = 1 + min(4, 3, 3) = 4
- dp[2][5]:
  - Comparing "d" with "a".
  - dp[2][5] = 1 + min(dp[1][5], dp[2][4], dp[1][4])
  - dp[2][5] = 1 + min(5, 4, 4) = 5
- dp[2][6]:
  - Comparing "d" with "n".
  - dp[2][6] = 1 + min(dp[1][6], dp[2][5], dp[1][5])
  - dp[2][6] = 1 + min(6, 5, 5) = 6
- dp[2][7]:
  - Comparing "d" with "c".
  - dp[2][7] = 1 + min(dp[1][7], dp[2][6], dp[1][6])
  - dp[2][7] = 1 + min(7, 6, 6) = 7
- dp[2][8]:
  - Comparing "d" with "e".
  - dp[2][8] = 1 + min(dp[1][8], dp[2][7], dp[1][7])
  - dp[2][8] = 1 + min(8, 7, 7) = 8

		d	**i**	**s**	**t**	**a**	**n**	**c**	**e**
	0	1	2	3	4	5	6	7	8
e	1	1	2	3	4	5	6	7	7
d	2	1	2	3	4	5	6	7	8
i	3								
t	4								
i	5								
n	6								
g	7								

Table 11: Step 10: Fill out the second row for "d"

Step 11: Fill out the third row for "i"

- Compare "i" with each character in "distance"
- dp[3][1]:
  - Comparing "i" with "d".
  - dp[3][1] = 1 + min(dp[2][1], dp[3][0], dp[2][0])
  - dp[3][1] = 1 + min(1, 3, 2) = 2
- dp[3][2]:
  - Comparing "i" with "i".
  - Characters are the same, so no operation is needed.
  - dp[3][2] = dp[2][1] = 1
- dp[3][3]:
  - Comparing "i" with "s".
  - dp[3][3] = 1 + min(dp[2][3], dp[3][2], dp[2][2])
  - dp[3][3] = 1 + min(3, 1, 2) = 2
- dp[3][4]:
  - Comparing "i" with "t".
  - dp[3][4] = 1 + min(dp[2][4], dp[3][3], dp[2][3])
  - dp[3][4] = 1 + min(4, 2, 3) = 3
- dp[3][5]:
  - Comparing "i" with "a".
  - dp[3][5] = 1 + min(dp[2][5], dp[3][4], dp[2][4])
  - dp[3][5] = 1 + min(5, 3, 4) = 4
- dp[3][6]:
  - Comparing "i" with "n".
  - dp[3][6] = 1 + min(dp[2][6], dp[3][5], dp[2][5])
  - dp[3][6] = 1 + min(6, 4, 5) = 5
- dp[3][7]:
  - Comparing "i" with "c".
  - dp[3][7] = 1 + min(dp[2][7], dp[3][6], dp[2][6])
  - dp[3][7] = 1 + min(7, 5, 6) = 6
- dp[3][8]:
  - Comparing "i" with "e".
  - dp[3][8] = 1 + min(dp[2][8], dp[3][7], dp[2][7])
  - dp[3][8] = 1 + min(8, 6, 7) = 7

		d	**i**	**s**	**t**	**a**	**n**	**c**	**e**
	0	1	2	3	4	5	6	7	8
e	1	1	2	3	4	5	6	7	7
d	2	1	2	3	4	5	6	7	8
i	3	2	1	2	3	4	5	6	7
t	4								
i	5								
n	6								
g	7								

Table 12: Step 11: Fill out the third row for "i"

Step 12: Fill out the fourth row for "t"

- Compare "t" with each character in "distance"
- dp[4][1]:
  - Comparing "t" with "d".
  - dp[4][1] = 1 + min(dp[3][1], dp[4][0], dp[3][0])
  - dp[4][1] = 1 + min(2, 4, 3) = 3
- dp[4][2]:
  - Comparing "t" with "i".
  - dp[4][2] = 1 + min(dp[3][2], dp[4][1], dp[3][1])
  - dp[4][2] = 1 + min(1, 3, 2) = 2
- dp[4][3]:
  - Comparing "t" with "s".
  - dp[4][3] = 1 + min(dp[3][3], dp[4][2], dp[3][2])
  - dp[4][3] = 1 + min(2, 2, 1) = 2
- dp[4][4]:
  - Comparing "t" with "t".
  - Characters are the same, so no operation is needed.
  - dp[4][4] = dp[3][3] = 2
- dp[4][5]:
  - Comparing "t" with "a".
  - dp[4][5] = 1 + min(dp[3][5], dp[4][4], dp[3][4])
  - dp[4][5] = 1 + min(4, 2, 3) = 3
- dp[4][6]:
  - Comparing "t" with "n".
  - dp[4][6] = 1 + min(dp[3][6], dp[4][5], dp[3][5])
  - dp[4][6] = 1 + min(5, 3, 4) = 4
- dp[4][7]:
  - Comparing "t" with "c".

- dp[4][7] = 1 + min(dp[3][7], dp[4][6], dp[3][6])
- dp[4][7] = 1 + min(6, 4, 5) = 5

- dp[4][8]:
  - Comparing "t" with "e".
  - dp[4][8] = 1 + min(dp[3][8], dp[4][7], dp[3][7])
  - dp[4][8] = 1 + min(7, 5, 6) = 6

		d	**i**	**s**	**t**	**a**	**n**	**c**	**e**
	0	1	2	3	4	5	6	7	8
e	1	1	2	3	4	5	6	7	7
d	2	1	2	3	4	5	6	7	8
i	3	2	1	2	3	4	5	6	7
t	4	3	2	2	2	3	4	5	6
i	5								
n	6								
g	7								

Table 13: Step 12: Fill out the fourth row for "t"

Step 13: Fill out the fifth row for "i"

- Compare "i" with each character in "distance"
- dp[5][1]:
  - Comparing "i" with "d".
  - dp[5][1] = 1 + min(dp[4][1], dp[5][0], dp[4][0])
  - dp[5][1] = 1 + min(3, 5, 4) = 4
- dp[5][2]:
  - Comparing "i" with "i".
  - Characters are the same, so no operation is needed.
  - dp[5][2] = dp[4][1] = 3
- dp[5][3]:
  - Comparing "i" with "s".
  - dp[5][3] = 1 + min(dp[4][3], dp[5][2], dp[4][2])
  - dp[5][3] = 1 + min(2, 3, 2) = 3
- dp[5][4]:
  - Comparing "i" with "t".
  - dp[5][4] = 1 + min(dp[4][4], dp[5][3], dp[4][3])
  - dp[5][4] = 1 + min(2, 3, 2) = 3
- dp[5][5]:
  - Comparing "i" with "a".
  - dp[5][5] = 1 + min(dp[4][5], dp[5][4], dp[4][4])
  - dp[5][5] = 1 + min(3, 3, 2) = 3

- dp[5][6]:
  - Comparing "i" with "n".
  - dp[5][6] = 1 + min(dp[4][6], dp[5][5], dp[4][5])
  - dp[5][6] = 1 + min(4, 3, 3) = 4
- dp[5][7]:
  - Comparing "i" with "c".
  - dp[5][7] = 1 + min(dp[4][7], dp[5][6], dp[4][6])
  - dp[5][7] = 1 + min(5, 4, 4) = 5
- dp[5][8]:
  - Comparing "i" with "e".
  - dp[5][8] = 1 + min(dp[4][8], dp[5][7], dp[4][7])
  - dp[5][8] = 1 + min(6, 5, 5) = 6

		d	**i**	**s**	**t**	**a**	**n**	**c**	**e**
	0	1	2	3	4	5	6	7	8
e	1	1	2	3	4	5	6	7	7
d	2	1	2	3	4	5	6	7	8
i	3	2	1	2	3	4	5	6	7
t	4	3	2	2	2	3	4	5	6
i	5	4	3	3	3	3	4	5	6
n	6								
g	7								

Table 14: Step 13: Fill out the fifth row for "i"

Step 14: Fill out the sixth row for "n"

- Compare "n" with each character in "distance"
- dp[6][1]:
  - Comparing "n" with "d".
  - dp[6][1] = 1 + min(dp[5][1], dp[6][0], dp[5][0])
  - dp[6][1] = 1 + min(4, 6, 5) = 5
- dp[6][2]:
  - Comparing "n" with "i".
  - dp[6][2] = 1 + min(dp[5][2], dp[6][1], dp[5][1])
  - dp[6][2] = 1 + min(3, 5, 4) = 4
- dp[6][3]:
  - Comparing "n" with "s".
  - dp[6][3] = 1 + min(dp[5][3], dp[6][2], dp[5][2])
  - dp[6][3] = 1 + min(3, 4, 3) = 4
- dp[6][4]:

- Comparing "n" with "t".
- dp[6][4] = 1 + min(dp[5][4], dp[6][3], dp[5][3])
- dp[6][4] = 1 + min(3, 4, 3) = 4

- dp[6][5]:
  - Comparing "n" with "a".
  - dp[6][5] = 1 + min(dp[5][5], dp[6][4], dp[5][4])
  - dp[6][5] = 1 + min(3, 4, 3) = 4
- dp[6][6]:
  - Comparing "n" with "n".
  - Characters are the same, so no operation is needed.
  - dp[6][6] = dp[5][5] = 3
- dp[6][7]:
  - Comparing "n" with "c".
  - dp[6][7] = 1 + min(dp[5][7], dp[6][6], dp[5][6])
  - dp[6][7] = 1 + min(5, 3, 4) = 4
- dp[6][8]:
  - Comparing "n" with "e".
  - dp[6][8] = 1 + min(dp[5][8], dp[6][7], dp[5][7])
  - dp[6][8] = 1 + min(6, 4, 5) = 5

		d	**i**	**s**	**t**	**a**	**n**	**c**	**e**
	0	1	2	3	4	5	6	7	8
e	1	1	2	3	4	5	6	7	7
d	2	1	2	3	4	5	6	7	8
i	3	2	1	2	3	4	5	6	7
t	4	3	2	2	2	3	4	5	6
i	5	4	3	3	3	3	4	5	6
n	6	5	4	4	4	4	3	4	5
g	7								

Table 15: Step 14: Fill out the sixth row for "n"

Step 15: Fill out the seventh row for "g"

- Compare "g" with each character in "distance"
- dp[7][1]:
  - Comparing "g" with "d".
  - dp[7][1] = 1 + min(dp[6][1], dp[7][0], dp[6][0])
  - dp[7][1] = 1 + min(5, 7, 6) = 6
- dp[7][2]:
  - Comparing "g" with "i".
  - dp[7][2] = 1 + min(dp[6][2], dp[7][1], dp[6][1])

  - dp[7][2] = 1 + min(4, 6, 5) = 5

- dp[7][3]:
  - Comparing "g" with "s".
  - dp[7][3] = 1 + min(dp[6][3], dp[7][2], dp[6][2])
  - dp[7][3] = 1 + min(4, 5, 4) = 5
- dp[7][4]:
  - Comparing "g" with "t".
  - dp[7][4] = 1 + min(dp[6][4], dp[7][3], dp[6][3])
  - dp[7][4] = 1 + min(4, 5, 4) = 5
- dp[7][5]:
  - Comparing "g" with "a".
  - dp[7][5] = 1 + min(dp[6][5], dp[7][4], dp[6][4])
  - dp[7][5] = 1 + min(4, 5, 4) = 5
- dp[7][6]:
  - Comparing "g" with "n".
  - dp[7][6] = 1 + min(dp[6][6], dp[7][5], dp[6][5])
  - dp[7][6] = 1 + min(3, 5, 4) = 4
- dp[7][7]:
  - Comparing "g" with "c".
  - dp[7][7] = 1 + min(dp[6][7], dp[7][6], dp[6][6])
  - dp[7][7] = 1 + min(4, 4, 3) = 4
- dp[7][8]:
  - Comparing "g" with "e".
  - dp[7][8] = 1 + min(dp[6][8], dp[7][7], dp[6][7])
  - dp[7][8] = 1 + min(5, 4, 4) = 5

		d	**i**	**s**	**t**	**a**	**n**	**c**	**e**
	0	1	2	3	4	5	6	7	8
e	1	1	2	3	4	5	6	7	7
d	2	1	2	3	4	5	6	7	8
i	3	2	1	2	3	4	5	6	7
t	4	3	2	2	2	3	4	5	6
i	5	4	3	3	3	3	4	5	6
n	6	5	4	4	4	4	3	4	5
g	7	6	5	5	5	5	4	4	5

Table 16: Step 15: Fill out the seventh row for "g"

Step 16: Final Result

- After completed the table, the final value in the bottom-right cell which dp[7][8] = 5 represents the minimum edit distance between "editing" and "distance".
- This means that a minimum of 5 operations (insertions, deletions, or substitutions) are required to transform the word "editing" into "distance".

7.3 Knapsack (both 0-1 and unrestrained)

7.31 Explanation

The knapsack problem is a classic optimization problem that involves selecting items with given weights and values to maximize the total value that can be carried in a knapsack of limited capacity. There are two primary variations of the knapsack problem:

0-1 Knapsack:

- Each item can either be included or excluded in its entirety. Fractional selections are not allowed.

Unbounded Knapsack:

- Items can be selected multiple times, allowing for unlimited quantities of each item.

For the 0-1 knapsack, the solution depends on whether to include or exclude an item, whereas for the unbounded knapsack, need to consider the possibility of repeatedly selecting an item.

Imagine you are at a grocery store with a fixed budget of $50. Each grocery item has a price (weight) and a nutritional value (value). Your goal is to maximize the total nutritional value while staying within the budget:

- In the 0-1 scenario, you can only buy one unit of each item, one unit of milk, one unit of bread
- In the unbounded scenario, you can buy multiple units of the same item like 2 units of milk, or 3 units of bread, as long as you do not exceed the budget.

Pseudocode for 0-1 Knapsack

Algorithm 6 0-1 Knapsack Problem

```
Input  : Weights array $w$, Values array $v$, Capacity $C$, Number of items $n$
Output: Maximum value that can be carried in the knapsack
Initialize a DP table $dp[n+1][C+1]$ with all zeros
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $C$ do
        if $w[i-1] \leq j$ then
            $dp[i][j] \leftarrow \max(dp[i-1][j], v[i-1] + dp[i-1][j - w[i-1]])$
        else
            $dp[i][j] \leftarrow dp[i-1][j]$
return $dp[n][C]$
```

Step-by-Step Explanation for 0-1 Knapsack

- Initialize a DP table with dimensions (n+1) X (C+1), where n is the number of items and C is the capacity of the knapsack. Each cell dp[i][j] represents the maximum value achievable with the first i items and capacity j.
- Iterate over each item. For each item i and capacity j, decide whether to include the item:
  - If the item's weight w[i-1] exceeds the current capacity j, exclude it.
  - Otherwise, calculate the maximum value by including or excluding the item, and store the result in dp[i][j].
- The final answer is stored in dp[n][C], representing the maximum value achievable with all items and the given capacity.

Pseudocode for Unbounded Knapsack

Algorithm 7 Unbounded Knapsack Problem

```
Input  : Weights array $w$, Values array $v$, Capacity $C$, Number of items $n$
Output: Maximum value that can be carried in the knapsack
Initialize a DP array $dp[C + 1]$ with all zeros
for $i \leftarrow 0$ to $n - 1$ do
    for $j \leftarrow w[i]$ to $C$ do
        $dp[j] \leftarrow \max(dp[j], v[i] + dp[j - w[i]])$
return $dp[C]$
```

Step-by-Step Explanation for Unbounded Knapsack

- Initialize a DP array of size C+1, where each index j represents the maximum value achievable with capacity j.
- For each item i, iterate over all capacities j starting from w[i]. Update dp[j] to include the maximum value achievable by repeatedly selecting item I.
- The final answer is stored in dp[C], representing the maximum value achievable with unlimited quantities of each item and the given capacity.

7.32 Exercise

Consider the following scenario: You are an adventurer collecting treasures in a dungeon. Each treasure has a specific weight and value, and you have a backpack with a maximum capacity of 10 units. The goal is to maximize the total value of the treasures you can carry.

Items Available:

Item	**Weight**	**Value**
1 (Gold Coin)	1	1
2 (Silver Coin)	3	4
3 (Emerald)	4	5
4 (Diamond)	5	7
5 (Artifact)	6	10

Table 17: List of Items with Weight and Value

Knapsack Capacity: C = 10

Solve the problem using both the 0-1 knapsack and unbounded knapsack approaches:

- For the 0-1 Knapsack, each item can be picked at most once.
- For the Unbounded Knapsack, items can be picked multiple times as long as the total weight does not exceed the capacity.

Goal: Determine the maximum total value that can be carried in the knapsack under each scenario (0-1 and unbounded).

7.33 0-1 Knapsack Solution

Step 1: Initialize DP Table

- Initialize the DP table dp[n+1][C+1], where n is the number of items, and C is the capacity it's 10 in this case.
- The base cases are:
  - dp[0][j] = 0 for all j (no items means no value).
  - dp[i][0] = 0 for all i (zero capacity means no value).

Item/Capacity	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1 (Gold Coin)	0										
2 (Silver Coin)	0										
3 (Emerald)	0										
4 (Diamond)	0										
5 (Artifact)	0										

Table 18: Step 1: Initialize DP Table for 0-1 Knapsack

Step 2: Process Item 1 (Gold Coin, Weight = 1, Value = 1)

- For j = 1 to 10, calculate:
  - If j ≥ weight of item, dp[1][j] = max(dp[0][j], dp[0][j-1] + 1).
  - Otherwise, dp[1][j] = dp[0][j].

Item/Capacity	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1 (Gold Coin)	0	1	1	1	1	1	1	1	1	1	1
2 (Silver Coin)	0										
3 (Emerald)	0										
4 (Diamond)	0										
5 (Artifact)	0										

Table 19: Step 2: Update DP Table for Item 1 (Gold Coin)

Step 3: Process Item 2 (Silver Coin, Weight = 3, Value = 4)

- For j = 1 to 10, calculate:
  - If j ≥ weight of item, dp[2][j] = max(dp[1][j], dp[1][j-3] + 4).
  - Otherwise, dp[2][j] = dp[1][j].

Item/Capacity	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1 (Gold Coin)	0	1	1	1	1	1	1	1	1	1	1
2 (Silver Coin)	0	1	1	4	5	5	5	5	5	5	5
3 (Emerald)	0										
4 (Diamond)	0										
5 (Artifact)	0										

Table 20: Step 3: Update DP Table for Item 2 (Silver Coin)

Step 4: Process Item 3 (Emerald, Weight = 4, Value = 5)

- For j = 1 to 10, calculate:
  - If j ≥ weight of item, dp[3][j] = max(dp[2][j], dp[2][j-4] + 5).
  - Otherwise, dp[3][j] = dp[2][j].

Item/Capacity	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1 (Gold Coin)	0	1	1	1	1	1	1	1	1	1	1
2 (Silver Coin)	0	1	1	4	5	5	5	5	5	5	5
3 (Emerald)	0	1	1	4	5	6	6	9	10	10	10
4 (Diamond)	0										
5 (Artifact)	0										

Table 21: Step 4: Update DP Table for Item 3 (Emerald)

Step 5: Process Item 4 (Diamond, Weight = 5, Value = 7)

- For j = 1 to 10, calculate:
  - If j ≥ weight of item, dp[4][j] = max(dp[3][j], dp[3][j-5] + 7).
  - Otherwise, dp[4][j] = dp[3][j].

Item/Capacity	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1 (Gold Coin)	0	1	1	1	1	1	1	1	1	1	1
2 (Silver Coin)	0	1	1	4	5	5	5	5	5	5	5
3 (Emerald)	0	1	1	4	5	6	6	9	10	10	10
4 (Diamond)	0	1	1	4	5	7	8	9	10	11	12
5 (Artifact)	0										

Table 22: Step 5: Update DP Table for Item 4 (Diamond)

Step 6: Process Item 5 (Artifact, Weight = 6, Value = 10)

- For j = 1 to 10, calculate:
  - If j ≥ weight of item, dp[5][j] = max(dp[4][j], dp[4][j-6] + 10).
  - Otherwise, dp[5][j] = dp[4][j].

Item/Capacity	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1 (Gold Coin)	0	1	1	1	1	1	1	1	1	1	1
2 (Silver Coin)	0	1	1	4	5	5	5	5	5	5	5
3 (Emerald)	0	1	1	4	5	6	6	9	10	10	10
4 (Diamond)	0	1	1	4	5	7	8	9	10	11	12
5 (Artifact)	0	1	1	4	5	7	10	11	11	14	15

Table 23: Step 6: Final DP Table for Item 5 (Artifact)

Step 7: Final Result

- Maximum Value for 0-1 Knapsack: dp[5][10] = 15

7.34 Unbounded Knapsack Solution

Step 1: Initialize DP Table

- Initialize the DP array dp[C+1], where C is the capacity (10 in this case).
- The base case is dp[0] = 0 (zero capacity means no value).

Capacity	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Value	0	0	0	0	0	0	0	0	0	0	0

Table 24: Step 1: Initialize DP Table for Unbounded Knapsack

Step 2: Process Item 1 (Gold Coin, Weight = 1, Value = 1)

- For each capacity j = 1 to 10, calculate dp[j] = max(dp[j], dp[j-1] + 1).

Capacity	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Value	0	1	2	3	4	5	6	7	8	9	10

Table 25: Step 2: Update DP Table for Item 1 (Gold Coin)

Step 3: Process Item 2 (Silver Coin, Weight = 3, Value = 4)

- For each capacity j = 3 to 10, calculate dp[j] = max(dp[j], dp[j-3] + 4).

Capacity	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Value	0	1	2	4	5	6	8	9	10	12	13

Table 26: Step 3: Update DP Table for Item 2 (Silver Coin)

Step 4: Process Item 3 (Emerald, Weight = 4, Value = 5)

- For each capacity j = 4 to 10, calculate dp[j] = max(dp[j], dp[j-4] + 5).

Capacity	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Value	0	1	2	4	5	6	8	9	10	12	13

Table 27: Step 4: Update DP Table for Item 3 (Emerald)

Step 5: Process Item 4 (Diamond, Weight = 5, Value = 7)

- For each capacity j = 5 to 10, calculate dp[j] = max(dp[j], dp[j-5] + 7).

Capacity	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Value	0	1	2	4	5	7	8	9	10	12	14

Table 28: Step 5: Update DP Table for Item 4 (Diamond)

Step 6: Process Item 5 (Artifact, Weight = 6, Value = 10)

- For each capacity j = 6 to 10, calculate dp[j] = max(dp[j], dp[j-6] + 10).

Capacity	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Value	0	1	2	4	5	7	10	11	12	14	15

Table 29: Step 6: Update DP Table for Item 5 (Artifact)

Final Result

- The maximum value that can be obtained with a capacity of 10 is $dp[10] = 15$.
- This value is achieved by selecting:
  - One Artifact (Weight = 6, Value = 10),
  - One Gold Coin (Weight = 1, Value = 1),
  - One Diamond (Weight = 3, Value = 4).
- Total Weight: 6 + 1 + 3 = 10.
- Total Value: 10 + 1 + 4 = 15.

7.4 Dynamic Programming Algorithm Design

7.41 Problem: Optimal Meal Plan for Calorie Budget

You are planning your meals for a day with a specific calorie budget. Each meal option has a specific calorie count and satisfaction value. The goal is to maximize the total satisfaction while staying within your calorie budget.

Input:

- A list of meal options, where each option has a calorie count and a satisfaction value.
- A calorie budget (integer) that represents the maximum calories you can consume.

Output:

- The maximum satisfaction value achievable within the calorie budget, allowing multiple servings of the same meal.

Meal Options:

- Option 1: calories = 200, satisfaction = 50
- Option 2: calories = 300, satisfaction = 60
- Option 3: calories = 400, satisfaction = 70
- Option 4: calories = 500, satisfaction = 85

And a calorie budget of 800.

7.42 Pseudocode and Explanation

This is an Unbounded Knapsack Problem where meals can be selected multiple times. Each meal option has a weight (calorie count) and value (satisfaction). First, to solve this problem, build a DP table and each entry represents the maximum satisfaction value achievable for a given calorie budget.

Algorithm 8 Optimal Meal Plan for Calorie Budget (Unbounded Knapsack)

```
Input  : List of meals, where each meal has calories and satisfaction, and a calorie budget
Output: Maximum satisfaction achievable within the calorie budget
Initialize a DP array dp of size calorie budget + 1, filled with 0
for calories = 1 to calorie budget do
    foreach meal in meals do
        if meal.calories ≤ calories then
            dp[calories] = max(dp[calories], dp[calories - meal.calories] + meal.satisfaction)
return dp[calorie budget]
```

Step-by-step Explanation of Pseudocode:

- A DP array dp is initialized, where dp[j] represents the maximum satisfaction achievable with a calorie budget of j.
- For each meal option, update the DP array by considering whether to include the current meal.
- For each calorie budget, use the formula dp[calories] = max(dp[calories], dp[calories - meal.calories] + meal.satisfaction) to ensure always choose the option with the highest satisfaction value.
- After processing all meals, dp[calorie budget] contains the maximum satisfaction value.

7.43 Solution

Calorie Budget	**0**	**100**	**200**	**300**	**400**	**500**	**600**	**700**	**800**
Satisfaction	0	0	0	0	0	0	0	0	0

Table 30: Step 1: Initialize DP Table for Unbounded Knapsack

Step 1: Initialization of DP Table

Step 2: Process Meal 1 (Calories = 200, Satisfaction = 50)

- For calorie budgets from 200 to 800, update the DP table:
- dp[j] = max(dp[j], dp[j - 200] + 50).

Calorie Budget	**0**	**100**	**200**	**300**	**400**	**500**	**600**	**700**	**800**
Satisfaction	0	0	50	50	100	100	150	150	200

Table 31: Step 2: Update DP Table for Meal 1

Step 3: Process Meal 2 (Calories = 300, Satisfaction = 60)

- For calorie budgets from 300 to 800, update the DP table:
- dp[j] = max(dp[j], dp[j - 300] + 60).

Calorie Budget	**0**	**100**	**200**	**300**	**400**	**500**	**600**	**700**	**800**
Satisfaction	0	0	50	60	100	110	150	160	200

Table 32: Step 3: Update DP Table for Meal 2

Step 4: Process Meal 3 (Calories = 400, Satisfaction = 70)

- For calorie budgets $j \geq 400$, calculate:
- dp[j] = max(dp[j], dp[j - 400] + 70).

Calorie Budget	**0**	**100**	**200**	**300**	**400**	**500**	**600**	**700**	**800**
Satisfaction	0	0	50	60	100	110	150	160	200

Table 33: Step 4: Update DP Table for Meal 3

Step 5: Process Meal 4 (Calories = 500, Satisfaction = 85)

- For calorie budgets $j \geq 500$, calculate:
- dp[j] = max(dp[j], dp[j - 500] + 85).

Calorie Budget	0	100	200	300	400	500	600	700	800
Satisfaction	0	0	50	60	100	110	150	160	200

Table 34: Step 5: Update DP Table for Meal 4

Step 6: Final Result

- The maximum satisfaction value achievable with a calorie budget of 800 is 200.
- This is achieved by selecting:
  - 4 servings of Meal 1 (Calories = 200, Satisfaction = 50 each),
  - or other combinations that respect the calorie constraint and maximize satisfaction.
- Total Calories: $4 \times 200 = 800$.
- Total Satisfaction: $4 \times 50 = 200$.

7.44 Correctness

Optimal Substructure

- The satisfaction for a calorie budget j can be determined by considering whether to include a specific meal. If we include it, the remaining problem is just to find the optimal solution for j - meal.calories. This recursive structure ensures that solving subproblems leads directly to solving the main problem.

Overlapping Subproblem

- Since the same calorie budgets are evaluated multiple times for different meals, overlapping subproblems naturally arise. The DP array ensures we only compute each subproblem once, storing the result for reuse.

Formula Validity

- The formula dp[j] = max(dp[j], dp[j - meal.calories] + meal.satisfaction) include two possibilities:
- Excluding the current meal (dp[j] remains unchanged).
- Including the current meal (add the satisfaction value to the optimal solution for the remaining calories).
- This ensures that every possible combination of meals is evaluated, and the maximum value is correctly recorded.

7.45 Time Complexity

Outer loop

- The outer loop iterates through all possible calorie budgets from 1 to C, where C is the total calorie budget. This gives a complexity of $O(C)$.

Inner loop

- For each calorie budget, the inner iterate over all meal options. If there are n meals, this loop runs n times for each budget value. Thus, the total complexity contributed by the inner loop is O(n).

In total the Time Complexity is $O(C) \times O(n) = O(n \times C)$

8 Linear Programming

8.1 Technique Definition And Problem Specification

8.11 Basic Definition

Linear Programming (LP) is a mathematical technique used to optimize a specific objective, such as maximizing profit or minimizing cost, subject to a set of linear constraints. The objective function and constraints must be linear equations or inequalities. This technique is widely applied in various fields like supply chain optimization, financial planning, and resource allocation.

I think of linear programming as a way to solve real-world optimization problems where resources are limited. It's like planning a balanced meal, we want the meal to taste the best (objective) while sticking to a budget and meeting nutritional requirements (constraints). Linear programming helps define these relationships mathematically to find the best possible solution.

Pseudocode for Solving Linear Programming Problems

Algorithm 9 Linear Programming Solver

```
Input  : Objective function Z, Constraints $C_1, C_2, \ldots, C_n$
Output: Optimal solution for decision variables and value of Z
Initialize the problem with the given objective function and constraints
Convert all inequalities into standard form if required
Set up the initial simplex tableau or input the problem into a linear programming solver
while the current solution is not optimal do
    Identify the entering variable
    Identify the leaving variable
    Perform a pivot operation to update the solution
return values of decision variables and optimal objective value
```

Step-by-Step Explanation of the Pseudocode

- **Step 1: Initialization**
  - Define the objective function that maximizes or minimizes Z.
  - Convert all constraints into standard form by adding slack variables if needed.
- **Step 2: Set up Simplex Tableau**
  - Represent the objective function and constraints in a tableau format for computation.
- **Step 3: Pivoting and Optimization**
  - Identify the entering variable.
  - Identify the leaving variable.
  - Perform a pivot to update the tableau.
- **Step 4: Optimality Check**
  - Stop when no variable can further improve Z.
  - Return the values of decision variables and the optimal value of Z.

8.12 Exercise

A farmer owns 100 acres of land and wants to maximize profit by growing two crops, wheat and corn. Each crop requires a certain amount of land, water, and fertilizer, and generates a specific profit per acre. The farmer has limited resources, 200 hours of water and 240 units of fertilizer available.

Data:

- **Crops:**
  - **Wheat:**
    - Land required: 1 acre per unit
    - Water required: 2 hours per acre
    - Fertilizer required: 3 units per acre
    - Profit: \$50 per acre
  - **Corn:**
    - Land required: 1 acre per unit
    - Water required: 4 hours per acre
    - Fertilizer required: 2 units per acre
    - Profit: \$70 per acre
- **Resources:**
  - Total land: 100 acres
  - Total water: 200 hours
  - Total fertilizer: 240 units

Determine how much of each crop to plant (acres of wheat and corn) to maximize profit while staying within resource constraints.

Linear Programming Formulation:

- **Decision Variables:**
  - x = acres of wheat to plant
  - y = acres of corn to plant
- **Objective Function:**

$$\text{Maximize: } Z = 50x + 70y$$

- **Constraints:**
  1. Land Constraint: Total land cannot exceed 100 acres

$$x + y \leq 100$$

  2. Water Constraint: Total water usage cannot exceed 200 hours

$$2x + 4y \leq 200$$

  3. Fertilizer Constraint: Total fertilizer usage cannot exceed 240 units

$$3x + 2y \leq 240$$

  4. Non-negativity Constraint: Acres planted cannot be negative

$$x \geq 0,\ y \geq 0$$

8.13 Solution

Step 1: Define the Problem and Graph the Constraints

First is a graph of the constraints to identify the feasible region. The constraints are:

- $x + y \leq 100$ (Land Constraint)
- $2x + 4y \leq 200$ (Water Constraint)
- $3x + 2y \leq 240$ (Fertilizer Constraint)
- $x \geq 0$, $y \geq 0$ (Non-negativity Constraints)

Graph of Constraints:

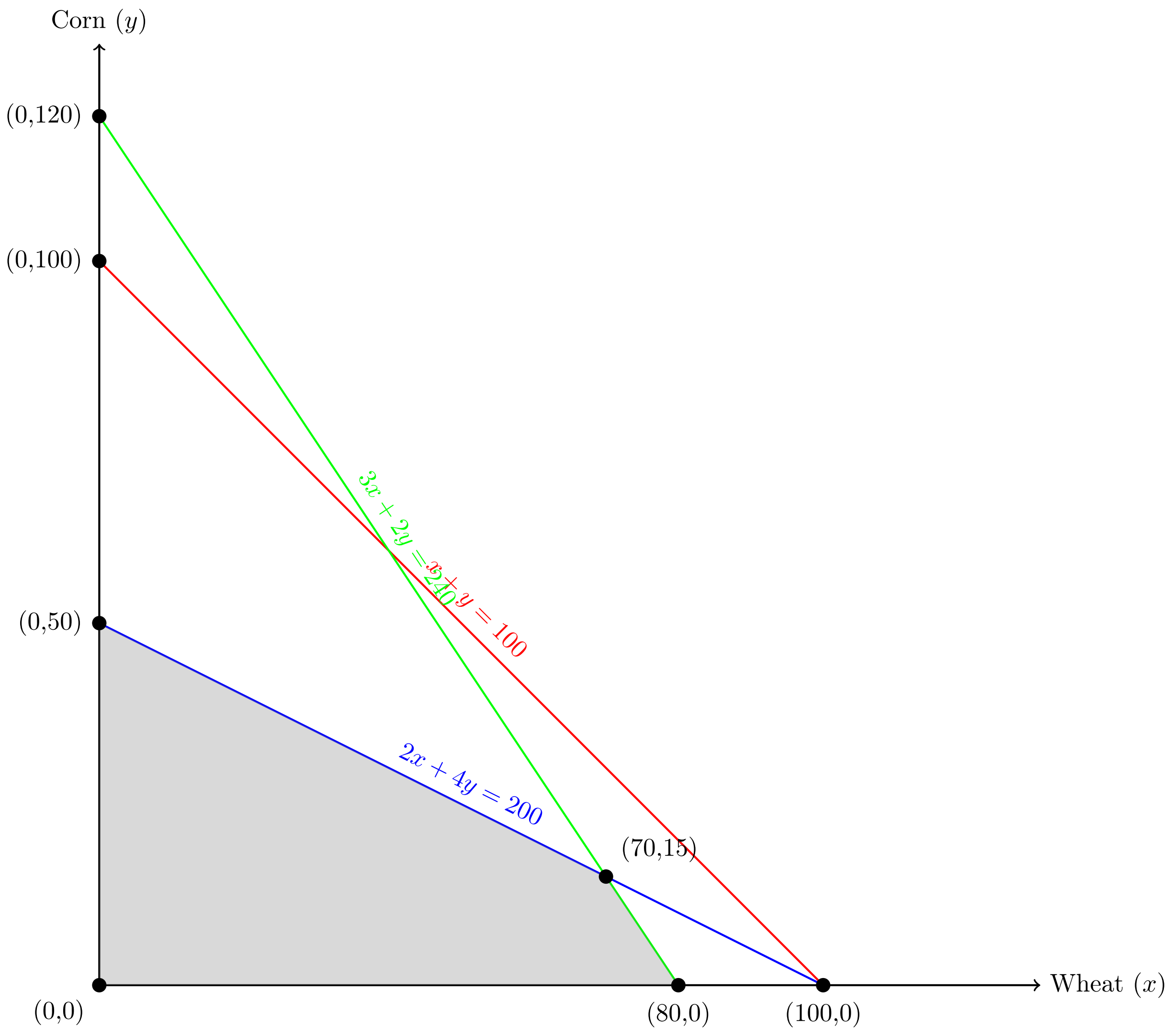

Figure 19: Feasible Region for Farming Output Problem

Step 2: Identify the Vertices of the Feasible Region

The feasible region is bounded by the intersection of the constraints. The vertices of this region are:

- $(0, 50)$: Intersection of $2x + 4y = 200$ and the y-axis.
- $(70, 15)$: Intersection of $2x + 4y = 200$ and $3x + 2y = 240$.
- $(80, 0)$: Intersection of $3x + 2y = 240$ and the x-axis.
- $(0, 0)$: Intersection of the x-axis and y-axis (non-negativity constraints).

Step 3: Compute the Objective Function at Each Vertex

Evaluate the profit function $\mathbf{Z = 50x + 70y}$ at each vertex:

- At $(0, 50)$: $Z = 50(0) + 70(50) = 3500$
- At $(70, 15)$: $Z = 50(70) + 70(15) = 3500 + 1050 = 4550$
- At $(80, 0)$: $Z = 50(80) + 70(0) = 4000$
- At $(0, 0)$: $Z = 50(0) + 70(0) = 0$

Step 4: Identify the Optimal Solution

Based on the computed values of Z, the maximum profit is achieved at the vertex $(70, 15)$, with a profit of \$4550.

Step 5: Final Result

- Optimal planting strategy is to plant 70 acres of wheat and 15 acres of corn.
- Maximum profit is \$4550.

9 NP-Completeness

9.1 Definition (no problem/solution required)

9.11 Technical Definition

NP Completeness refers to a class of problems that are both in NP (non deterministic polynomial time) and as hard as any problem in NP. It means, these problems are challenging because no known algorithm can solve them efficiently for all inputs. However, given a potential solution, it can be verified its correctness in polynomial time. If a polynomial time algorithm is discovered for one NP complete problem, all NP problems can be solved in polynomial time, which would imply $P = NP$. NP-completeness lies at the heart of computational complexity theory and directly impacts practical decision making in optimization, cryptography, and scheduling.

9.12 Example

Imagine there are tasked with scheduling university exams. Each course has a set of students enrolled, and some students take multiple courses. The goal is to assign a time slot to each exam such that no student has two exams at the same time while minimizing the total number of time slots. This problem is NP complete because:

- Checking a solution that verifying that no two exams overlap for any student can be done in polynomial time.
- Finding an optimal schedule requires exploring all possible assignments, which grows exponentially with the number of courses and students.

9.13 Pseudocode

Algorithm 10 Exam Scheduling

```
Input  : Set of courses $C$, Conflict matrix $M$ (where $M[i][j] = 1$ if course $i$ and $j$ share a student)
Output: Exam schedule with assigned time slots for each course
Sort courses in descending order of conflicts (most conflicts first)
Initialize an empty schedule map
foreach course $c$ in sorted courses do
    Assign the lowest available time slot $t$ to $c$
    Ensure no conflicts with already scheduled courses sharing $t$
return schedule map
```

Step-by-Step Explanation of Pseudocode

- **Step 1: Input Data**
  - The list of courses C and the conflict matrix M are given as inputs.
  - $M[i][j] = 1$ means course i and j cannot share the same time slot.
- **Step 2: Sort Courses**
  - Courses are sorted by the number of conflicts they have that degrees in the conflict graph.
  - This ensures that highly constrained courses are scheduled first.
- **Step 3: Assign Time Slots**
  - Iterate through each course in the sorted list.
  - Assign the first available time slot that does not conflict with previously scheduled courses.
- **Step 4: Return Schedule**
  - Return the map of courses to time slots.

9.2 SAT and 3SAT

9.21 Explanation

SAT (Satisfiability) is a basic problem in computer science that asks if there exists an assignment of truth values to variables such that a given Boolean formula evaluates to true. If such an assignment exists, the formula is said to be satisfiable. 3SAT is a specific form of SAT where the formula is expressed in conjunctive normal form, and each clause contains exactly three literals.
I understand SAT as trying to solve a "logic puzzle" by assigning values to variables so that a formula evaluates to true. 3SAT is like a restricted version of this puzzle where each condition involves three variables or their negations. SAT is crucial in many fields, such as automated theorem proving and hardware verification, but it is also NP-complete, making it challenging to solve efficiently for large inputs.

Imagine I am managing a team with multiple overlapping tasks. Each task has dependencies like the Task A must be done before Task B, or Tasks A and C cannot run at the same time. The goal is to create a feasible schedule that satisfies all dependencies while ensuring no conflicts. This scheduling problem can be formulated as a SAT problem by expressing the dependencies as Boolean constraints.

Pseudocode

Algorithm 11 SAT Solver

```
Input  : Boolean formula F with n variables in CNF
Output: A satisfying assignment if one exists, else UNSAT
Initialize all variables to UNASSIGNED
while there are unassigned variables do
    Pick an unassigned variable v
    Assign v a truth value (T or F)
    if F evaluates to true under current assignments then
        return the assignment
    Backtrack if a conflict is detected and flip the assignment of v
return UNSAT if all possibilities are exhausted
```

Step-by-Step Explanation of the Pseudocode

- **Step 1: Initialize Variables**
  - Start with all variables unassigned to ensure flexibility during evaluation.
- **Step 2: Assign Truth Values**
  - Select an unassigned variable and assign it a truth value that True or False.
  - Use a heuristic like Most Constrained Variable to improve efficiency.
- **Step 3: Evaluate the Formula**
  - Substitute the current truth assignments into the formula F.
  - If F evaluates to true, a satisfying assignment is found.
- **Step 4: Backtracking**

  - If a conflict occurs like a clause evaluates to false, backtrack by flipping the truth value of the last assigned variable.

- **Step 5: Exhaustion**
  - If all assignments are tried and no satisfying solution is found, return UNSAT.

9.22 Exercise

Given the Boolean formula in Conjunctive Normal Form (CNF):

- $F = (x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor \neg x_3) \land (x_1 \lor x_3 \lor \neg x_4)$

Prove whether this formula is satisfiable by finding a satisfying assignment of the variables x_1, x_2, x_3, and x_4, or prove that no such assignment exists.

Details:

- The formula consists of three clauses, each containing exactly three literals, making it a 3SAT problem.
- The variables x_1, x_2, x_3, and x_4 can be assigned truth values (True or False).
- The goal is to determine if there exists at least one assignment of truth values that satisfies all three clauses simultaneously.

Approach:

- Construct a truth table for all possible combinations of truth values for x_1, x_2, x_3, and x_4.
- Evaluate the formula F for each combination.
- Identify whether there exists at least one combination where F evaluates to True.

Outcome:

- If at least one assignment satisfies F, the formula is satisfiable (SAT). Otherwise, it is unsatisfiable (UNSAT).

9.23 Solution

Step 1: Construct a Truth Table

Evaluate the Boolean formula $F = (x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor \neg x_3) \land (x_1 \lor x_3 \lor \neg x_4)$ for all possible assignments of x_1, x_2, x_3, and x_4.

x_1	x_2	x_3	x_4	$x_1 \vee \neg x_2 \vee x_3$	$\neg x_1 \vee x_2 \vee \neg x_3$	$x_1 \vee x_3 \vee \neg x_4$	F
T	T	T	T	T	T	T	T
T	T	T	F	T	T	T	T
T	T	F	T	T	T	T	T
T	T	F	F	T	T	T	T
T	F	T	T	T	F	T	F
T	F	T	F	T	F	T	F
T	F	F	T	T	F	T	F
T	F	F	F	T	F	T	F
F	T	T	T	T	T	T	T
F	T	T	F	T	T	T	T
F	T	F	T	F	T	T	F
F	T	F	F	F	T	T	F
F	F	T	T	F	T	T	F
F	F	T	F	F	T	T	F
F	F	F	T	F	T	F	F
F	F	F	F	F	T	F	F

Table 35: Truth Table for Boolean Formula F

Step 2: Analyze the Results

From the truth table, there observe the following:

- The formula F evaluates to True in multiple cases, such as:
  - $x_1 =$ T, $x_2 =$ T, $x_3 =$ T, $x_4 =$ T
  - $x_1 =$ T, $x_2 =$ T, $x_3 =$ F, $x_4 =$ T
  - $x_1 =$ F, $x_2 =$ T, $x_3 =$ T, $x_4 =$ T
- These assignments satisfy all three clauses in the formula.

Step 3: Conclusion

Since there found assignments where F evaluates to True, the formula is satisfiable. A satisfying assignment is:

- $x_1 =$ T, $x_2 =$ T, $x_3 =$ T, $x_4 =$ T

This means that the Boolean equation is SAT.

9.3 Proving NP-Completeness

9.31 Explanation

Proving that a problem is NP-complete involves demonstrating two key points. First is the problem belongs to the class NP, which means that given a candidate solution, its correctness can be verified in polynomial time. Second is the problem is at least as hard as every other problem in NP, which is shown by reducing a known NP-complete problem to the target problem in polynomial time.

Imagine I am scheduling tasks with dependencies, such as task A needing to finish before task B. The goal is to find the shortest time to complete all tasks without violating dependencies. While verifying a proposed schedule is straightforward that checking dependencies is quick, finding the optimal schedule can be as hard as solving NP-complete problems like the Traveling Salesman Problem. Thus, task dependency scheduling can be shown to be NP-complete.

Assume use Pseudocode to proving NP-Completeness

Algorithm 12 Proving NP-Completeness

```
Input  : A problem P to prove NP-complete, and a known NP-complete problem Q
Output: Proof that P is NP-complete
Step 1: Prove P ∈ NP
Verify that any solution to P can be checked for correctness in polynomial time
Step 2: Reduction from Q to P
while there exists a known NP-complete problem Q do
    Show that any instance of Q can be transformed into an instance of P in polynomial time
Step 3: Conclude NP-completeness
If both steps are satisfied, P is NP-complete
```

Step-by-Step Explanation of the Pseudocode

- Verify that P belongs to NP by showing that a candidate solution for P can be verified in polynomial time. This step ensures that the problem is not harder than NP.
- Identify a known NP-complete problem Q, such as SAT, 3SAT, or Traveling Salesman Problem. Demonstrate a polynomial-time reduction from Q to P. This shows that solving P would inherently solve Q.
- Conclude that P is NP-complete since it satisfies the two conditions: $P \in NP$ and $Q \leq P$ that reduction in polynomial time.

9.32 Exercise

The Hamiltonian Cycle Problem asks whether a given undirected graph G = (V, E) contains a Hamiltonian cycle, determine if a cycle that visits each vertex exactly once and returns to the starting vertex.

9.33 Solution

Step 1: Verify Hamiltonian Cycle Belongs to NP

A Hamiltonian Cycle is a cycle in a graph that visits every vertex exactly once and returns to the starting vertex

Given a sequence of vertices, and then check:

- Each vertex is visited exactly once ($O(V)$).
- All edges in the sequence exist in the graph ($O(E)$).
- The last vertex connects back to the first vertex ($O(1)$).
- Verification time is $O(V + E)$, which is polynomial.

Therefore, the Hamiltonian Cycle belongs to NP.

Step 2: Reduction from Vertex Cover to Hamiltonian Cycle

To prove Hamiltonian Cycle is NP-complete, there needs to reduce it to a known NP-complete problem (Vertex Cover) to Hamiltonian Cycle. This involves creating a new graph G' from $G = (V, E)$ such that solving the Hamiltonian Cycle in G' implies solving Vertex Cover in G.

Step 3: Start with the Original Graph G

The Vertex Cover problem starts with a graph $G = (V, E)$:

$$V = \{v_1, v_2, v_3\}, \quad E = \{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}$$

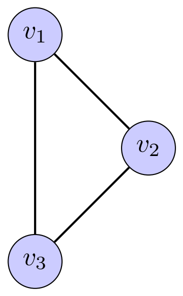

Figure 20: Original Graph G

Step 4: Replace Each Vertex with a Vertex Gadget

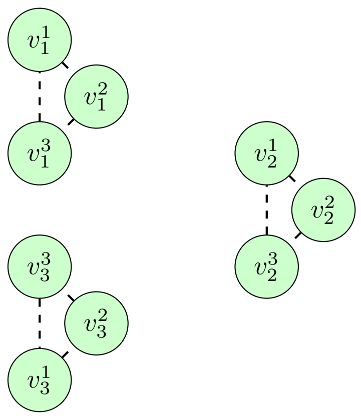

Figure 21: Vertex Gadgets for Each Vertex

For each vertex v_i, create a "vertex gadget" consisting of three nodes (v_i^1, v_i^2, v_i^3) connected in a triangle (cycle).

Step 5: Connect Gadgets According to Edges in G

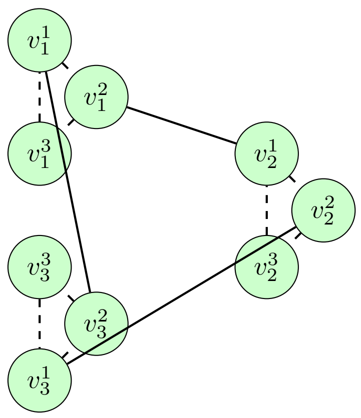

Figure 22: Edges Added Between Gadgets to Simulate Original Graph

For each edge $(u, v) \in E$, add edges between the gadgets for u and v. Connect v_i^2 from u's gadget to v_j^1 in v's gadget.

Step 6: Verify the Reduction

Hamiltonian Cycle in G':

- A valid Hamiltonian Cycle visits each gadget and satisfies edge connections.
- This corresponds to covering edges in G.

Mapping Back to Vertex Cover:

- Selecting nodes from gadgets in G' corresponds to selecting vertices in G for the Vertex Cover problem.

Step 7: Conclusion

Hamiltonian Cycle is NP-complete because:

- It is in NP (Step 1).
- Vertex Cover reduces to Hamiltonian Cycle in polynomial time (Step 2–6).