# DAY 7

1.

```
def dice_throw(num_sides, num_dice, target):
    # Create a DP table where dp[i][j] represents the number of ways to get sum j with i
dice
    dp = [[0] * (target + 1) for _ in range(num_dice + 1)]

    # Base case: There's one way to get sum 0 with 0 dice (by rolling nothing)
    dp[0][0] = 1

    # Fill the DP table
    for dice in range(1, num_dice + 1):  # For each die
        for sum_value in range(1, target + 1):  # For each sum from 1 to target
            # Check all values rolled by the current die (1 to num_sides)
            for roll in range(1, num_sides + 1):
                if sum_value - roll >= 0:
                    dp[dice][sum_value] += dp[dice - 1][sum_value - roll]

    # The result is the number of ways to get the target sum with all dice
    return dp[num_dice][target]

# Test Case 1
num_sides_1 = 6
num_dice_1 = 2
target_1 = 7
print(f"Number of ways to reach sum {target_1}: {dice_throw(num_sides_1,
num_dice_1, target_1)}")

# Test Case 2
num_sides_2 = 4
num_dice_2 = 3
target_2 = 10
print(f"Number of ways to reach sum {target_2}: {dice_throw(num_sides_2,
num_dice_2, target_2)}")
```

2.

```
def min_time_to_process(n, a1, a2, t1, t2, e1, e2, x1, x2):
    # Initialize dp arrays for assembly lines 1 and 2
    dp1 = [0] * n
    dp2 = [0] * n

    # Base case: First station entry
    dp1[0] = e1 + a1[0]  # Start at assembly line 1
    dp2[0] = e2 + a2[0]  # Start at assembly line 2

    # Fill the dp arrays for each subsequent station
    for i in range(1, n):
        dp1[i] = min(dp1[i-1] + a1[i], dp2[i-1] + t2[i-1] + a1[i])
        dp2[i] = min(dp2[i-1] + a2[i], dp1[i-1] + t1[i-1] + a2[i])

    # Calculate the final minimum time by considering exit times
    return min(dp1[n-1] + x1, dp2[n-1] + x2)

# Example Test Case 1
n = 4
a1 = [4, 5, 3, 2]  # Time at each station on assembly line 1
a2 = [2, 10, 1, 4]  # Time at each station on assembly line 2
t1 = [7, 4, 5]      # Transfer times from assembly line 1 to 2 after each station
t2 = [9, 2, 8]      # Transfer times from assembly line 2 to 1 after each station
e1 = 10             # Entry time for assembly line 1
e2 = 12             # Entry time for assembly line 2
x1 = 18             # Exit time from assembly line 1
x2 = 7             # Exit time from assembly line 2

print("Minimum time to process the product:", min_time_to_process(n, a1, a2, t1, t2, e1,
e2, x1, x2))
```

3.

```
def min_production_time(line1, line2, line3, transfer, dependencies):
    n = len(line1)  # Number of stations (3 in this case)
```

```
    # Initialize DP table
    # dp[line][station] stores the minimum time to reach the given station on a given line
    dp = [[float('inf')] * n for _ in range(3)]

    # Initialize the base case (processing the first station on each line)
    dp[0][0] = line1[0]
    dp[1][0] = line2[0]
    dp[2][0] = line3[0]

    # Fill the DP table
    for i in range(1, n):  # Process stations 1 and 2
        for line in range(3):
            # For each line, check the minimum time from all lines at previous station
            for prev_line in range(3):
                # Stay on the same line (no transfer)
                dp[line][i] = min(dp[line][i], dp[prev_line][i-1] + transfer[prev_line][line] + [line1,
line2, line3][line][i])

    # The final minimum time is the minimum time to reach the last station on any of the
lines
    return min(dp[0][n-1], dp[1][n-1], dp[2][n-1])


# Test Case
line1 = [5, 9, 3]
line2 = [6, 8, 4]
line3 = [7, 6, 5]

transfer = [
    [0, 2, 3],
    [2, 0, 4],
    [3, 4, 0]
]

dependencies = [(0, 1), (1, 2)]  # Station 0 -> 1 -> 2

# Call function to get the minimum production time
result = min_production_time(line1, line2, line3, transfer, dependencies)
print(f"Minimum production time: {result}")
```

4.

```
from itertools import permutations

# Function to implement the Floyd-Warshall algorithm
def floyd_warshall(graph):
    n = len(graph)

    # Initialize the dist matrix with the input graph
    dist = [[float('inf')] * n for _ in range(n)]

    # Set the initial distances based on the graph input
    for i in range(n):
        for j in range(n):
            if graph[i][j] != 0:
                dist[i][j] = graph[i][j]
            elif i == j:
                dist[i][j] = 0

    # Apply the Floyd-Warshall algorithm to find the shortest paths
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist

# Function to calculate the minimum path for visiting all nodes
def minimum_path(graph):
    n = len(graph)

    # First, get the shortest paths between all pairs using Floyd-Warshall
    dist = floyd_warshall(graph)

    # Generate all permutations of the nodes (except the first node, which we will
consider fixed)
    nodes = list(range(n))
    min_path = float('inf')
```

```
    # Generate all permutations of the nodes excluding the first one (i.e., visiting all
nodes)
    for perm in permutations(nodes[1:]):
        # Create a full path starting from node 0
        path = [0] + list(perm)

        # Calculate the total distance of this path
        path_cost = 0
        for i in range(len(path) - 1):
            path_cost += dist[path[i]][path[i + 1]]

        # Update the minimum path cost if this path is shorter
        if path_cost < min_path:
            min_path = path_cost

    return min_path

# Test Case 1
graph1 = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]
print("Minimum path distance for Test Case 1:", minimum_path(graph1))

# Test Case 2
graph2 = [
    [0, 10, 10, 10],
    [10, 0, 10, 10],
    [10, 10, 0, 10],
    [10, 10, 10, 0]
]
print("Minimum path distance for Test Case 2:", minimum_path(graph2))

# Test Case 3
graph3 = [
    [0, 1, 2, 3],
    [1, 0, 4, 5],
    [2, 4, 0, 6],
```

```
    [3, 5, 6, 0]
]
print("Minimum path distance for Test Case 3:", minimum_path(graph3))
```

5.

```
from itertools import permutations

# Function to calculate the total distance of a path
def calculate_total_distance(path, dist_matrix):
    total_distance = 0
    n = len(path)
    for i in range(n - 1):
        total_distance += dist_matrix[path[i]][path[i + 1]]
    # Add the distance to return to the starting city
    total_distance += dist_matrix[path[-1]][path[0]]
    return total_distance

# Function to solve the Traveling Salesperson Problem
def tsp(dist_matrix):
    # Number of cities (nodes)
    n = len(dist_matrix)

    # Generate all permutations of cities, except the first one (start from city 0)
    cities = list(range(n))
    min_distance = float('inf')
    min_route = None

    # Generate all permutations of the cities (excluding the starting city)
    for perm in permutations(cities[1:]):  # Fix city 0 as the starting point
        # Full path including city 0 as the start and end
        path = [0] + list(perm)

        # Calculate the total distance of the current path
        current_distance = calculate_total_distance(path, dist_matrix)
```

```
        # Check if this is the shortest path found so far
        if current_distance < min_distance:
            min_distance = current_distance
            min_route = path

    # Return the shortest route and its total distance
    return min_route, min_distance

# Distance matrix for the cities (A, B, C, D, E)
# Distances are symmetric: dist[i][j] == dist[j][i]
dist_matrix = [
    [0, 10, 15, 20, 25],  # A
    [10, 0, 35, 25, 30],  # B
    [15, 35, 0, 30, 20],  # C
    [20, 25, 30, 0, 15],  # D
    [25, 30, 20, 15, 0]   # E
]

# Solve the TSP and print the result
route, distance = tsp(dist_matrix)
print("Shortest route:", " -> ".join(chr(65 + i) for i in route))
print("Total distance:", distance)
```

6.

```
def longestPalindrome(s: str) -> str:
    # Helper function to expand around a given center
    def expand_around_center(left: int, right: int) -> str:
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        # Return the longest palindrome by the current center expansion
        return s[left+1:right]

    # If the string length is less than or equal to 1, it's already a palindrome
    if len(s) <= 1:
```

```python
        return s

    longest_palindrome = ""

    # Iterate over each character in the string
    for i in range(len(s)):
        # Check for odd-length palindromes
        palindrome1 = expand_around_center(i, i)
        # Check for even-length palindromes
        palindrome2 = expand_around_center(i, i + 1)

        # Update the longest palindrome if we find a longer one
        if len(palindrome1) > len(longest_palindrome):
            longest_palindrome = palindrome1
        if len(palindrome2) > len(longest_palindrome):
            longest_palindrome = palindrome2

    return longest_palindrome

# Example test cases
print(longestPalindrome("babad"))  # Output: "bab" or "aba"
print(longestPalindrome("cbbd"))   # Output: "bb"
```

7.

```python
def lengthOfLongestSubstring(s: str) -> int:
    # Dictionary to store the characters in the current window
    char_set = set()
    start = 0
    max_len = 0

    for end in range(len(s)):
        # If the character at end pointer is in the set, remove characters from start to end
        while s[end] in char_set:
            char_set.remove(s[start])
            start += 1
```

```
        # Add the current character to the set
        char_set.add(s[end])

        # Update the maximum length
        max_len = max(max_len, end - start + 1)

    return max_len

# Example test cases
print(lengthOfLongestSubstring("abcabcbb"))  # Output: 3
print(lengthOfLongestSubstring("bbbbb"))     # Output: 1
print(lengthOfLongestSubstring("pwwkew"))    # Output: 3
```

8.

```
def wordBreak(s: str, wordDict: list) -> bool:
    # Convert the wordDict to a set for faster lookups
    word_set = set(wordDict)

    # Initialize a DP array with False values, and dp[0] = True (empty string can always
be segmented)
    dp = [False] * (len(s) + 1)
    dp[0] = True

    # Iterate through each position in the string
    for i in range(1, len(s) + 1):
        # Check every possible partition of the substring s[0:i]
        for j in range(i):
            # If s[j:i] is in the word set and dp[j] is True (the substring s[0:j] can be
segmented)
            if dp[j] and s[j:i] in word_set:
                dp[i] = True
                break

    # The result will be stored in dp[len(s)]
```

```python
    return dp[len(s)]

# Example test cases
print(wordBreak("leetcode", ["leet", "code"]))  # Output: True
print(wordBreak("applepenapple", ["apple", "pen"]))  # Output: True
print(wordBreak("catsandog", ["cats", "dog", "sand", "and", "cat"]))  # Output: False
```

9.

```python
def wordBreak(s: str, wordDict: set) -> str:
    # Initialize dp array, with False values. dp[0] = True because an empty string is
always segmented.
    dp = [False] * (len(s) + 1)
    dp[0] = True  # Empty string can always be segmented.

    # Iterate over each character in the string
    for i in range(1, len(s) + 1):
        # Check all possible substrings s[j:i] (substring from index j to i)
        for j in range(i):
            if dp[j] and s[j:i] in wordDict:
                dp[i] = True
                break  # No need to check further once we find a valid segmentation

    # Final result is stored in dp[len(s)], which tells if the entire string can be segmented
    return "Yes" if dp[len(s)] else "No"

# Example usage:
wordDict = {"i", "like", "sam", "sung", "samsung", "mobile", "ice", "cream", "icecream",
"man", "go", "mango"}

# Test cases
print(wordBreak("ilike", wordDict))  # Output: Yes (can be segmented as "i like")
print(wordBreak("ilikesamsung", wordDict))  # Output: Yes (can be segmented as "i like
samsung" or "i like sam sung")
```

10.

```
def fullJustify(words, maxWidth):
    result = []  # To store the final result of lines
    current_line = []  # To hold words for the current line
    current_length = 0  # To track the current length of the line

    for word in words:
        # If adding the word and a space would exceed maxWidth, process the current line
        if current_length + len(word) + len(current_line) > maxWidth:
            # Calculate the spaces to distribute
            total_spaces = maxWidth - current_length
            if len(current_line) == 1:
                # If there's only one word in the line, add all spaces at the end
                result.append(current_line[0] + ' ' * total_spaces)
            else:
                # Calculate spaces between words
                spaces_between_words = total_spaces // (len(current_line) - 1)
                extra_spaces = total_spaces % (len(current_line) - 1)

                line = current_line[0]
                for i in range(1, len(current_line)):
                    # Add the calculated number of spaces
                    if i <= extra_spaces:
                        line += ' ' * (spaces_between_words + 1) + current_line[i]
                    else:
                        line += ' ' * spaces_between_words + current_line[i]
                result.append(line)

            # Reset for the next line
            current_line = [word]
            current_length = len(word)
        else:
            # Otherwise, add the word to the current line
            current_line.append(word)
            current_length += len(word)

    # Handle the last line, which is left-justified
    last_line = ' '.join(current_line)
    result.append(last_line + ' ' * (maxWidth - len(last_line)))  # Add remaining spaces
```

```python
    return result


# Example Test Cases
words1 = ["This", "is", "an", "example", "of", "text", "justification."]
maxWidth1 = 16
print(fullJustify(words1, maxWidth1))  # Output: ["This is an", "example of text", "justification. "]

words2 = ["What", "must", "be", "acknowledgment", "shall", "be"]
maxWidth2 = 16
print(fullJustify(words2, maxWidth2))  # Output: ["What must be", "acknowledgment ", "shall be "]
```

11.

```python
class WordFilter:
    def __init__(self, words):
        self.prefix_map = {}
        self.suffix_map = {}

        for i, word in enumerate(words):
            # Store all possible prefixes for the current word
            for j in range(len(word) + 1):
                prefix = word[:j]  # Prefix from 0 to j
                if prefix not in self.prefix_map:
                    self.prefix_map[prefix] = []
                self.prefix_map[prefix].append(i)

            # Store all possible suffixes for the current word
            for j in range(len(word) + 1):
                suffix = word[-j:]  # Suffix from -j to the end
                if suffix not in self.suffix_map:
                    self.suffix_map[suffix] = []
                self.suffix_map[suffix].append(i)
```

```python
    def f(self, pref, suff):
        # Get the indices of words with the given prefix and suffix
        prefix_indices = self.prefix_map.get(pref, [])
        suffix_indices = self.suffix_map.get(suff, [])

        # Find the largest index that is common in both lists
        i, j = len(prefix_indices) - 1, len(suffix_indices) - 1
        result = -1
        while i >= 0 and j >= 0:
            if prefix_indices[i] == suffix_indices[j]:
                result = prefix_indices[i]
                break
            elif prefix_indices[i] > suffix_indices[j]:
                i -= 1
            else:
                j -= 1

        return result

# Example Test Case
words = ["apple"]
wordFilter = WordFilter(words)
print(wordFilter.f("a", "e"))  # Output: 0
```