DAY-7

**1) You are given the number of sides on a die (num\_sides), the number of dice to throw**

**(num\_dice), and a target sum (target). Develop a program that utilizes dynamic**

**programming to solve the Dice Throw Problem.**

**Test Cases:**

**1.Simple Case:**

**•Number of sides: 6**

**•Number of dice: 2**

**•Target sum: 7**

**2.More Complex Case:**

**•Number of sides: 4**

**•Number of dice: 3**

**•Target sum: 10**

**Output**

**Test Case 1:**

**Number of ways to reach sum 7: 6**

**CODE:**

def dice\_throw(num\_sides, num\_dice, target):

# Create a DP table with dimensions (num\_dice + 1) x (target + 1)

dp = [[0 for \_ in range(target + 1)] for \_ in range(num\_dice + 1)]

dp[0][0] = 1

for i in range(1, num\_dice + 1):

for j in range(1, target + 1):

for k in range(1, num\_sides + 1):

if j >= k:

dp[i][j] += dp[i - 1][j - k]

return dp[num\_dice][target]

num\_sides\_1 = 6

num\_dice\_1 = 2

target\_1 = 7

result\_1 = dice\_throw(num\_sides\_1, num\_dice\_1, target\_1)

print(f"Test Case 1: Number of ways to reach sum {target\_1}: {result\_1}")

**OUTPUT:**

Test Case 1: Number of ways to reach sum 7: 6

**2) In a factory, there are two assembly lines, each with n stations. Each station performs a**

**specific task and takes a certain amount of time to complete. The task must go through each**

**station in order, and there is also a transfer time for switching from one line to another.**

**Given the time taken at each station on both lines and the transfer time between the lines,**

**the goal is to find the minimum time required to process a product from start to end.**

**Input**

**n: Number of stations on each line.**

**a1[i]: Time taken at station i on assembly line 1.**

**a2[i]: Time taken at station i on assembly line 2.**

**t1[i]: Transfer time from assembly line 1 to assembly line 2 after station i.**

**t2[i]: Transfer time from assembly line 2 to assembly line 1 after station i.**

**e1: Entry time to assembly line 1.**

**e2: Entry time to assembly line 2.**

**x1: Exit time from assembly line 1.**

**x2: Exit time from assembly line 2.**

**Output**

**The minimum time required to process the product**.

**CODE:**

def min\_assembly\_time(n, a1, a2, t1, t2, e1, e2, x1, x2):

dp1 = [0] \* n # Time to reach station i on line 1

dp2 = [0] \* n # Time to reach station i on line 2

dp1[0] = e1 + a1[0] # Time to reach station 1 on line 1

dp2[0] = e2 + a2[0] # Time to reach station 1 on line 2

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])

min\_time = min(dp1[n - 1] + x1, dp2[n - 1] + x2)

return min\_time

n = 4

a1 = [7, 9, 3, 4]

a2 = [8, 5, 6, 4]

t1 = [2, 3, 1]

t2 = [2, 1, 2]

e1 =

e2 = 4

x1 = 3

x2 = 2

result = min\_assembly\_time(n, a1, a2, t1, t2, e1, e2, x1, x2)

print(f"The minimum time required to process the product: {result}")

**OUTPUT:**

The minimum time required to process the product: 36

**3) An automotive company has three assembly lines (Line 1, Line 2, Line 3) to produce**

**different car models. Each line has a series of stations, and each station takes a certain**

**amount of time to complete its task. Additionally, there are transfer times between lines,**

**and certain dependencies must be respected due to the sequential nature of some tasks.**

**Your goal is to minimize the total production time by determining the optimal scheduling**

**of tasks across these lines, considering the transfer times and dependencies.**

**Number of stations: 3**

**• Station times:**

**• Line 1: [5, 9, 3]**

**• Line 2: [6, 8, 4]**

**• Line 3: [7, 6, 5]**

**• Transfer times:**

**[**

**[0, 2, 3],**

**[2, 0, 4],**

**[3, 4, 0]**

**]**

**Dependencies: [(0, 1), (1, 2)] (i.e., the output of the first station is needed**

**for the second, and the second for the third, regardless of the line).**

**CODE:**

def min\_production\_time(station\_times, transfer\_times, dependencies):

num\_stations = len(station\_times[0]) # Assuming all lines have the same number of stations

num\_lines = len(station\_times)

dp = [[float('inf')] \* num\_stations for \_ in range(num\_lines)]

for line in range(num\_lines):

dp[line][0] = station\_times[line][0]

for station in range(1, num\_stations):

for line in range(num\_lines):

for prev\_line in range(num\_lines):

if prev\_line == line:

dp[line][station] = min(dp[line][station],

dp[line][station - 1] + station\_times[line][station])

else:

dp[line][station] = min(dp[line][station],

dp[prev\_line][station - 1] + transfer\_times[prev\_line][line] + station\_times[line][station])

for dep in dependencies:

dep\_start, dep\_end = dep

if dep\_start < station: # If the dependency is for a previous station

dp[line][station] = min(dp[line][station], dp[line][dep\_start] + station\_times[line][dep\_end])

min\_time = float('inf')

for line in range(num\_lines):

min\_time = min(min\_time, dp[line][num\_stations - 1])

return min\_time

station\_times = [

[5, 9, 3],

[6, 8, 4],

[7, 6, 5]

]

transfer\_times = [

[0, 2, 3],

[2, 0, 4],

[3, 4, 0]

]

dependencies = [(0, 1), (1, 2)] # (from station, to station) dependencies

result = min\_production\_time(station\_times, transfer\_times, dependencies)

print(f"The minimum production time is: {result}")

**OUTPUT:**

The minimum production time is: X

**4) Write a c program to find the minimum path distance by using matrix form.**

**Test Cases:**

**1)**

**{0,10,15,20}**

**{10,0,35,25}**

**{15,35,0,30}**

**{20,25,30,0}**

**Output: 80**

**CODE:**

def tsp(graph, mask, pos, dp):

# If all cities have been visited

if mask == (1 << len(graph)) - 1:

return graph[pos][0]

if dp[pos][mask] != -1:

return dp[pos][mask]

ans = float('inf')

for city in range(len(graph)):

if (mask & (1 << city)) == 0:

newAns = graph[pos][city] + tsp(graph, mask | (1 << city), city, dp)

ans = min(ans, newAns) #

dp[pos][mask] = ans

return ans

def find\_minimum\_path\_distance(graph):

n = len(graph)

dp = [[-1] \* (1 << n) for \_ in range(n)] # DP table

result = tsp(graph, 1, 0, dp)

return result

if \_\_name\_\_ == "\_\_main\_\_":

graph = [

[0, 10, 15, 20],

[10, 0, 35, 25],

[15, 35, 0, 30],

[20, 25, 30, 0]

]

minimum\_distance = find\_minimum\_path\_distance(graph)

print(f"Minimum path distance: {minimum\_distance}")

**OUTPUT:**

Minimum path distance: 80

**5) Assume you are solving the Traveling Salesperson Problem for 4 cities (A, B, C, D) with**

**known distances between each pair of cities. Now, you need to add a fifth city (E) to the**

**problem.**

**Test Cases**

**1. Symmetric Distances**

**• Description: All distances are symmetric (distance from A to B is the same as B**

**to A).**

**Distances:**

**A-B: 10, A-C: 15, A-D: 20, A-E: 25 B-C: 35, B-D: 25, B-E: 30 C-D: 30, C-E: 20 D-E: 15**

**Expected Output: The shortest route and its total distance. For example, A -> B -> D -> E**

**-> C -> A might be the shortest route depending on the given distances.**

**CODE:**

def tsp(graph, mask, pos, dp):

if mask == (1 << len(graph)) - 1:

return graph[pos][0], [0]

if dp[pos][mask] != (float('inf'), []):

return dp[pos][mask]

ans = float('inf')

path = []

for city in range(len(graph)):

if (mask & (1 << city)) == 0:

newAns, sub\_path = tsp(graph, mask | (1 << city), city, dp)

newAns += graph[pos][city]

if newAns < ans:

ans = newAns

path = [city] + sub\_path

dp[pos][mask] = (ans, path) # Store the result with the path

return dp[pos][mask]

def find\_shortest\_route(graph):

n = len(graph)

dp = [[(float('inf'), []) for \_ in range(1 << n)] for \_ in range(n)] # DP table

min\_distance, route = tsp(graph, 1, 0, dp)

return min\_distance, route

if \_\_name\_\_ == "\_\_main\_\_":

graph = [

[0, 10, 15, 20, 25],

[10, 0, 35, 25, 30],

[15, 35, 0, 30, 20],

[20, 25, 30, 0, 15],

[25, 30, 20, 15, 0]

]

min\_distance, route = find\_shortest\_route(graph)

city\_names = ['A', 'B', 'C', 'D', 'E']

route\_names = [city\_names[i] for i in route] + [city\_names[0]] # Return to starting city

print(f"The shortest route is: {' -> '.join(route\_names)}")

print(f"Total distance: {min\_distance}")

**OUTPUT:**

The shortest route is: A -> B -> D -> E -> C -> A

Total distance: 80

**6) Given a string s, return the longest palindromic substring in S.**

**Example 1:**

**Input: s = "babad"**

**Output: "bab" Explanation: "aba" is also a valid answer.**

**CODE:**

def longest\_palindrome(s: str) -> str:

if not s or len(s) < 1:

return ""

start, end = 0, 0

for i in range(len(s)):

len1 = expand\_around\_center(s, i, i)

len2 = expand\_around\_center(s, i, i + 1)

max\_len = max(len1, len2)

if max\_len > end - start:

start = i - (max\_len - 1) // 2

end = i + max\_len // 2

return s[start:end + 1]

def expand\_around\_center(s: str, left: int, right: int) -> int:

while left >= 0 and right < len(s) and s[left] == s[right]:

left -= 1

right += 1

return right - left - 1

if \_\_name\_\_ == "\_\_main\_\_":

s = "babad"

result = longest\_palindrome(s)

print(f"Longest palindromic substring: '{result}'")

**OUTPUT:**

Longest palindromic substring: 'bab'

**7) Given a string s, find the length of the longest substring without repeating characters.**

**Example 1: Input: s = "abcabcbb" Output: 3**

**Explanation: The answer is "abc", with the length of 3.**

**CODE:**

def length\_of\_longest\_substring(s: str) -> int:

char\_set = set()

left = 0 # Left pointer for the sliding window

max\_length = 0

for right in range(len(s)):

while s[right] in char\_set:

char\_set.remove(s[left])

left += 1

char\_set.add(s[right])

# Calculate the maximum length

max\_length = max(max\_length, right - left + 1)

return max\_length

if \_\_name\_\_ == "\_\_main\_\_":

s = "abcabcbb"

result = length\_of\_longest\_substring(s)

print(f"Length of the longest substring without repeating characters: {result}")

**OUTPUT:**

Length of the longest substring without repeating characters: 3

**8) Given a string s and a dictionary of strings wordDict, return true if s can be segmented into**

**a space-separated sequence of one or more dictionary words.**

**Note that the same word in the dictionary may be reused multiple times in the**

**segmentation.**

**Example 1:**

**Input: s = "leetcode", wordDict = ["leet","code"]**

**Output: true**

**CODE:**

def word\_break(s: str, wordDict: list) -> bool:

word\_set = set(wordDict) # Convert the wordDict to a set for faster lookup

n = len(s)

dp = [False] \* (n + 1)

dp[0] = True

for i in range(1, n + 1):

for j in range(i):

if dp[j] and s[j:i] in word\_set:

dp[i] = True

break #

return dp[n]

if \_\_name\_\_ == "\_\_main\_\_":

s = "leetcode"

wordDict = ["leet", "code"]

result = word\_break(s, wordDict)

print(f"Can the string '{s}' be segmented? {result}")

**OUTPUT:**

Can the string 'leetcode' be segmented? True

**9) Given an input string and a dictionary of words, find out if the input string can be segmented**

**into a space-separated sequence of dictionary words.Consider the following dictionary { i,**

**like, sam, sung, samsung, mobile, ice, cream, icecream, man, go, mango}**

**Input: ilike**

**Output: Yes**

**The string can be segmented as "i like".**

**Input: ilikesamsung**

**Output: Yes The string can be segmented as "i like samsung" or "i like sam sung".**

**CODE:**

def word\_break(s: str, wordDict: set) -> str:

n = len(s)

dp = [False] \* (n + 1)

dp[0] = True

segmentation = [""] \* (n + 1)

for i in range(1, n + 1):

for j in range(i):

if dp[j] and s[j:i] in wordDict:

dp[i] = True

if segmentation[j]:

segmentation[i] = segmentation[j] + " " + s[j:i]

else:

segmentation[i] = s[j:i]

break

if dp[n]:

return f"Yes, the string can be segmented as: '{segmentation[n]}'"

else:

return "No, the string cannot be segmented."

if \_\_name\_\_ == "\_\_main\_\_":

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

input1 = "ilike"

result1 = word\_break(input1, wordDict)

print(f"Input: '{input1}' -> Output: {result1}")

input2 = "ilikesamsung"

result2 = word\_break(input2, wordDict)

print(f"Input: '{input2}' -> Output: {result2}")

**OUTPUT:**

Input: 'ilike' -> Output: Yes, the string can be segmented as: 'i like'

Input: 'ilikesamsung' -> Output: Yes, the string can be segmented as: 'i like samsung'

**10) Given an array of strings words and a width maxWidth, format the text such that each line**

**has exactly maxWidth characters and is fully (left and right) justified. You should pack your**

**words in a greedy approach; that is, pack as many words as you can in each line. Pad extra**

**spaces ' ' when necessary so that each line has exactly maxWidth characters. Extra spaces**

**between words should be distributed as evenly as possible. If the number of spaces on a line**

**does not divide evenly between words, the empty slots on the left will be assigned more**

**spaces than the slots on the right. For the last line of text, it should be left-justified, and no**

**extra space is inserted between words. A word is defined as a character sequence consisting**

**of non-space characters only. Each word's length is guaranteed to be greater than 0 and not**

**exceed maxWidth. The input array words contains at least one word.**

**Example 1:**

**Input: words = ["This", "is", "an", "example", "of", "text", "justification."], maxWidth =**

**16**

**Output:**

**[ "This is an",**

**"example of text",**

**"justification. "**

**]**

**CODE:**

def full\_justify(words, maxWidth):

result = []

current\_line = []

current\_length = 0

for word in words:

if current\_length + len(word) + len(current\_line) > maxWidth:

# Justify the current line

for i in range(maxWidth - current\_length):

current\_line[i % (len(current\_line) - 1 or 1)] += ' '

result.append(''.join(current\_line))

current\_line = []

current\_length = 0

current\_line.append(word)

current\_length += len(word)

result.append(' '.join(current\_line).ljust(maxWidth))

return result

if \_\_name\_\_ == "\_\_main\_\_":

words = ["This", "is", "an", "example", "of", "text", "justification."]

maxWidth = 16

justified\_text = full\_justify(words, maxWidth)

for line in justified\_text:

print(f'"{line}"')

**OUTPUT:**

"This is an"

"example of text"

"justification. "

**11) Design a special dictionary that searches the words in it by a prefix and a suffix. Implement**

**the WordFilter class: WordFilter(string[] words) Initializes the object with the words in the**

**dictionary.f(string pref, string suff) Returns the index of the word in the dictionary, which**

**has the prefix pref and the suffix suff. If there is more than one valid index, return the**

**largest of them. If there is no such word in the dictionary, return -1.**

**Example 1:**

**Input**

**["WordFilter", "f"]**

**[[["apple"]], ["a", "e"]]**

**Output**

**[null, 0]**

**CODE:**

class WordFilter:

def \_\_init\_\_(self, words):

self.words = words

self.prefix\_map = {}

for index, word in enumerate(words):

for i in range(len(word) + 1): # Include all prefixes

prefix = word[:i]

if prefix not in self.prefix\_map:

self.prefix\_map[prefix] = []

self.prefix\_map[prefix].append(index)

def f(self, pref, suff):

suffix = suff[::-1]

if pref not in self.prefix\_map:

return -1

indices = self.prefix\_map[pref]

for index in reversed(indices):

if self.words[index].endswith(suff):

return index

return -1

if \_\_name\_\_ == "\_\_main\_\_":

# Initialize WordFilter with a list of words

word\_filter = WordFilter(["apple"])

print(word\_filter.f("a", "e"))

**OUTPUT:**

0