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

Output

Test Case 1:

Number of ways to reach sum 7:6

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

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

dp[0][0] = 1

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

for sum\_val in range(1, target + 1):

dp[dice][sum\_val] = sum(dp[dice - 1][sum\_val - face] for face in range(1, num\_sides + 1) if sum\_val - face >= 0)

return dp[num\_dice][target]

num\_sides = 6

num\_dice = 2

target = 7

print("Number of ways to reach sum", target, ":", dice\_throw(num\_sides, num\_dice, target))

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.

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

time1 = [0] \* n

time2 = [0] \* n

time1[0] = e1 + a1[0]

time2[0] = e2 + a2[0]

for i in range(1, n):

time1[i] = min(time1[i - 1] + a1[i], time2[i - 1] + t2[i - 1] + a1[i])

time2[i] = min(time2[i - 1] + a2[i], time1[i - 1] + t1[i - 1] + a2[i])

final\_time1 = time1[-1] + x1

final\_time2 = time2[-1] + x2

return min(final\_time1, final\_time2)

n = 4

a1 = [4, 5, 3, 2]

a2 = [2, 10, 1, 4]

t1 = [7, 4, 5]

t2 = [9, 2, 8]

e1 = 10

e2 = 12

x1 = 18

x2 = 7

print("Minimum time required to process the product:", min\_assembly\_time(n, a1, a2, t1, t2, e1, e2, x1, x2))

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).

def min\_production\_time(num\_stations, line\_times, transfer\_times, dependencies):

dp = [[float('inf')] \* 3 for \_ in range(num\_stations)]

for line in range(3):

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

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

for line in range(3):

dp[station][line] = min(

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

dp[station - 1][(line + 1) % 3] + transfer\_times[(line + 1) % 3][line] + line\_times[line][station],

dp[station - 1][(line + 2) % 3] + transfer\_times[(line + 2) % 3][line] + line\_times[line][station]

)

min\_total\_time = min(dp[num\_stations - 1][0], dp[num\_stations - 1][1], dp[num\_stations - 1][2])

return min\_total\_time

num\_stations = 3

line\_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)]

print("Minimum production time:", min\_production\_time(num\_stations, line\_times, transfer\_times, dependencies))

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

import sys

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

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

return graph[pos][0]

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

return dp[mask][pos]

ans = sys.maxsize

for city in range(len(graph)):

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

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

ans = min(ans, new\_ans)

dp[mask][pos] = ans

return ans

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

n = len(graph)

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

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

return min\_distance

graph = [

[0, 10, 15, 20],

[10, 0, 35, 25],

[15, 35, 0, 30],

[20, 25, 30, 0]

]

output = find\_min\_path\_distance(graph)

print("Output:", output)

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.

N = 5

INF = float('inf')

dist = [

[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]

]

dp = [[-1] \* (1 << N) for \_ in range(N)]

def tsp(pos, visited):

if visited == (1 << N) - 1:

return dist[pos][0]

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

return dp[pos][visited]

min\_cost = INF

for city in range(N):

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

new\_cost = dist[pos][city] + tsp(city, visited | (1 << city))

min\_cost = min(min\_cost, new\_cost)

dp[pos][visited] = min\_cost

return min\_cost

def find\_path():

path = []

visited = 1

pos = 0

for \_ in range(N - 1):

path.append(chr(pos + ord('A')))

next\_city = -1

min\_cost = INF

for city in range(N):

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

cost = dist[pos][city] + dp[city][visited | (1 << city)]

if cost < min\_cost:

min\_cost = cost

next\_city = city

pos = next\_city

visited |= (1 << pos)

path.append(chr(pos + ord('A')))

path.append('A')

return path

min\_distance = tsp(0, 1)

path = find\_path()

print("Shortest route:", " -> ".join(path))

print("Total distance:", min\_distance)

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

def longest\_palindromic\_substring(s):

if not s or len(s) == 1:

return s

start, max\_length = 0, 1

def expand\_from\_center(left, right):

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

left -= 1

right += 1

return right - left - 1

for i in range(len(s)):

len1 = expand\_from\_center(i, i)

len2 = expand\_from\_center(i, i + 1)

max\_len = max(len1, len2)

if max\_len > max\_length:

max\_length = max\_len

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

return s[start:start + max\_length]

s = "babad"

print("Longest palindromic substring:", longest\_palindromic\_substring(s))

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.

def length\_of\_longest\_substring(s):

char\_index = {}

max\_length = 0

start = 0

for end in range(len(s)):

if s[end] in char\_index and char\_index[s[end]] >= start:

start = char\_index[s[end]] + 1

char\_index[s[end]] = end

max\_length = max(max\_length, end - start + 1)

return max\_length

s = "abcabcbb"

print("Length of the longest substring without repeating characters:", length\_of\_longest\_substring(s))

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

def word\_break(s, wordDict):

word\_set = set(wordDict)

dp = [False] \* (len(s) + 1)

dp[0] = True

for i in range(1, len(s) + 1):

for j in range(i):

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

dp[i] = True

break

return dp[len(s)]

s = "leetcode"

wordDict = ["leet", "code"]

print("Can the string be segmented?", word\_break(s, wordDict))

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".

def word\_break(s, wordDict):

word\_set = set(wordDict)

dp = [False] \* (len(s) + 1)

dp[0] = True

for i in range(1, len(s) + 1):

for j in range(i):

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

dp[i] = True

break

return "Yes" if dp[len(s)] else "No"

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

print("Input: ilike -> Output:", word\_break("ilike", wordDict))

print("Input: ilikesamsung -> Output:", word\_break("ilikesamsung", wordDict))

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. " ]

def full\_justify(words, maxWidth):

result = []

current\_line = []

current\_length = 0

for word in words:

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

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)

last\_line = ' '.join(current\_line).ljust(maxWidth)

result.append(last\_line)

return result

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

maxWidth = 16

output = full\_justify(words, maxWidth)

for line in output:

print(f'"{line}"')

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] Explanation WordFilter wordFilter = new WordFilter(["apple"]); wordFilter.f("a", "e"); // return 0, because the word at index 0 has prefix = "a" and suffix = "e".

class WordFilter:

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

self.words = words

self.prefix\_suffix\_map = {}

for index, word in enumerate(words):

for i in range(len(word) + 1):

for j in range(len(word) + 1):

prefix = word[:i]

suffix = word[j:]

self.prefix\_suffix\_map[(prefix, suffix)] = index

def f(self, pref, suff):

return self.prefix\_suffix\_map.get((pref, suff), -1)

wordFilter = WordFilter(["apple"])

print(wordFilter.f("a", "e"))