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.

Program:

def count\_ways(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 t in range(1, target + 1):

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

if t - face >= 0:

dp[dice][t] += dp[dice - 1][t - face]

return dp[num\_dice][target]

num\_sides = int(input("Number of sides: "))

num\_dice = int(input("Number of dice: "))

target = int(input("Target sum: "))

print("Ways to get target sum:", count\_ways(num\_sides, num\_dice, target))

output:

Number of sides: 2

Number of dice: 6

Target sum: 34

Ways to get target sum: 0

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.

Program:

def assembly\_line(a, t, e, x, n):

T1 = [0] \* n

T2 = [0] \* n

T1[0] = e[0] + a[0][0]

T2[0] = e[1] + a[1][0]

for i in range(1, n):

T1[i] = min(T1[i-1] + a[0][i], T2[i-1] + t[1][i] + a[0][i])

T2[i] = min(T2[i-1] + a[1][i], T1[i-1] + t[0][i] + a[1][i])

return min(T1[-1] + x[0], T2[-1] + x[1])

n = int(input("Number of stations: "))

a = [list(map(int, input("Line 1 times: ").split())),

list(map(int, input("Line 2 times: ").split()))]

t = [[0] + list(map(int, input("Line 1 to 2 transfers: ").split())),

[0] + list(map(int, input("Line 2 to 1 transfers: ").split()))]

e = list(map(int, input("Entry times: ").split()))

x = list(map(int, input("Exit times: ").split()))

print("Minimum time required:", assembly\_line(a, t, e, x, n))

output:

Minimum time required: 35

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.

Program:

def assembly\_line\_3(a, t, e, x, n):

T1 = [0] \* n

T2 = [0] \* n

T3 = [0] \* n

T1[0] = e[0] + a[0][0]

T2[0] = e[1] + a[1][0]

T3[0] = e[2] + a[2][0]

for i in range(1, n):

T1[i] = min(T1[i-1] + a[0][i], T2[i-1] + t[1][i] + a[0][i], T3[i-1] + t[2][i] + a[0][i])

T2[i] = min(T2[i-1] + a[1][i], T1[i-1] + t[0][i] + a[1][i], T3[i-1] + t[2][i] + a[1][i])

T3[i] = min(T3[i-1] + a[2][i], T1[i-1] + t[0][i] + a[2][i], T2[i-1] + t[1][i] + a[2][i])

return min(T1[-1] + x[0], T2[-1] + x[1], T3[-1] + x[2])

n = int(input("Number of stations: "))

a = [list(map(int, input("Line 1 times: ").split())),

list(map(int, input("Line 2 times: ").split())),

list(map(int, input("Line 3 times: ").split()))]

t = [[0] + list(map(int, input("Line 1 to others: ").split())),

[0] + list(map(int, input("Line 2 to others: ").split())),

[0] + list(map(int, input("Line 3 to others: ").split()))]

e = list(map(int, input("Entry times: ").split()))

x = list(map(int, input("Exit times: ").split()))

print("Minimum time required:", assembly\_line\_3(a, t, e, x, n))

output:

Number of stations: 3

Line 1 times: 4 5 3

Line 2 times: 2 10 1

Line 3 times: 3 6 2

Line 1 to others: 2 3

Line 2 to others: 1 2

Line 3 to others: 4 1

Entry times: 10 12 11

Exit times: 18 7 9

Minimum time required: 27

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

Program:

#include <stdio.h>

#define MIN(a, b) ((a) < (b) ? (a) : (b))

int minPath(int matrix[100][100], int m, int n) {

int dp[100][100];

dp[0][0] = matrix[0][0];

for (int i = 1; i < n; i++)

dp[0][i] = dp[0][i-1] + matrix[0][i];

for (int i = 1; i < m; i++)

dp[i][0] = dp[i-1][0] + matrix[i][0];

for (int i = 1; i < m; i++) {

for (int j = 1; j < n; j++) {

dp[i][j] = matrix[i][j] + MIN(dp[i-1][j], dp[i][j-1]);

}

}

return dp[m-1][n-1];

}

int main() {

int m, n, matrix[100][100];

printf("Enter rows and columns: ");

scanf("%d %d", &m, &n);

printf("Enter matrix:\n");

for (int i = 0; i < m; i++) {

for (int j = 0; j < n; j++) {

scanf("%d", &matrix[i][j]);

}

}

printf("Minimum path distance: %d\n", minPath(matrix, m, n));

return 0;

}

Output:

Enter rows and columns: 3 3

Enter matrix:

1 3 5

2 1 2

4 3 1

Enter matrix:

1 3 5

2 1 2

4 3 1

Output:

Minimum path distance: 7

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.

Program:

from itertools import permutations

def tsp(graph, start):

cities = list(range(len(graph)))

cities.remove(start)

min\_path = float('inf')

for perm in permutations(cities):

current\_path = 0

k = start

for j in perm:

current\_path += graph[k][j]

k = j

current\_path += graph[k][start]

min\_path = min(min\_path, current\_path)

return min\_path

# Distance matrix for cities A, B, C, D, E

graph = [

[0, 10, 15, 20, 25], # Distances from A

[10, 0, 35, 25, 30], # Distances from B

[15, 35, 0, 30, 20], # Distances from C

[20, 25, 30, 0, 15], # Distances from D

[25, 30, 20, 15, 0] # Distances from E

]

start\_city = 0 # Starting from A

print("Minimum path distance:", tsp(graph, start\_city))

output:

Minimum path distance: 85

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

Program:

def longestPalindrome(s):

res = ""

for i in range(len(s)):

# Odd length palindromes

temp = expandFromCenter(s, i, i)

if len(temp) > len(res):

res = temp

# Even length palindromes

temp = expandFromCenter(s, i, i+1)

if len(temp) > len(res):

res = temp

return res

def expandFromCenter(s, left, right):

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

left -= 1

right += 1

return s[left+1:right]

s = input("Enter a string: ")

print("Longest palindromic substring:", longestPalindrome(s))

output:

Enter a string: babad

Longest palindromic substring: bab

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

Program:

def lengthOfLongestSubstring(s):

char\_map = {}

left = 0

max\_length = 0

for right in range(len(s)):

if s[right] in char\_map and char\_map[s[right]] >= left:

left = char\_map[s[right]] + 1

char\_map[s[right]] = right

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

return max\_length

s = input("Enter a string: ")

print("Length of longest substring without repeating characters:", lengthOfLongestSubstring(s))

output:

Enter a string: abcabcbb

Length of 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.

Program:

def wordBreak(s, wordDict):

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

dp[0] = True

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

for word in wordDict:

if i >= len(word) and s[i-len(word):i] == word:

dp[i] = dp[i] or dp[i-len(word)]

return dp[-1]

s = input("Enter a string: ")

wordDict = input("Enter dictionary words (space-separated): ").split()

print("Can the string be segmented?", wordBreak(s, wordDict))

output:

Enter a string: leetcode

Enter dictionary words (space-separated): leet code

Can the string 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}

program:

def wordBreak(s, wordDict):

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

dp[0] = True

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

for word in wordDict:

if i >= len(word) and s[i-len(word):i] == word:

dp[i] = dp[i] or dp[i-len(word)]

return dp[-1]

s = input("Enter a string: ")

wordDict = {"i", "like", "sam", "sung", "samsung", "mobile",

"ice", "cream", "icecream", "man", "go", "mango"}

print("Can the string be segmented?", wordBreak(s, wordDict))

output:

Enter a string: ilikeicecream

Can the string be segmented? True

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.

Program:

def fullJustify(words, maxWidth):

res, line, num\_of\_letters = [], [], 0

for word in words:

if num\_of\_letters + len(word) + len(line) > maxWidth:

for i in range(maxWidth - num\_of\_letters):

line[i % (len(line) - 1 or 1)] += ' '

res.append(''.join(line))

line, num\_of\_letters = [], 0

line.append(word)

num\_of\_letters += len(word)

res.append(' '.join(line).ljust(maxWidth))

return res

words = input("Enter words (space-separated): ").split()

maxWidth = int(input("Enter max width: "))

result = fullJustify(words, maxWidth)

print("Justified Text:")

for line in result:

print(f'"{line}"')

output:

Enter words (space-separated): This is an example of text justification

Enter max width: 16

Justified Text:

"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.

Program:

class WordFilter:

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

self.lookup = {}

for index, word in enumerate(words):

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

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

key = word[:i] + '#' + word[-j:]

self.lookup[key] = index

def f(self, pref, suff):

return self.lookup.get(pref + '#' + suff, -1)

words = input("Enter words (space-separated): ").split()

wf = WordFilter(words)

pref = input("Enter prefix: ")

suff = input("Enter suffix: ")

print("Index:", wf.f(pref, suff))

output:

Enter words (space-separated): apple banana application apex

Enter prefix: ap

Enter suffix: e

Index: 3