DAY-7 PROGRAMS

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

CODE:

num\_sides = 6

num\_dice = 2

target = 7

dp = [[0] \* (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):

dp[i][j] = 0

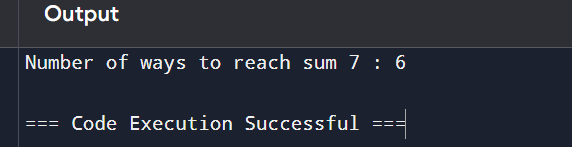
for k in range(1, num\_sides + 1):

if j >= k:

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

print("Number of ways to reach sum", target, ":", dp[num\_dice][target])

OUTPUT:



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:

n = 4

a1 = [4, 5, 3, 2]

a2 = [2, 10, 1, 4]

t1 = [0, 7, 4, 5]

t2 = [0, 9, 2, 8]

e1, e2 = 10, 12

x1, x2 = 18, 7

T1 = [0] \* n

T2 = [0] \* n

T1[0] = e1 + a1[0]

T2[0] = e2 + a2[0]

for i in range(1, n):

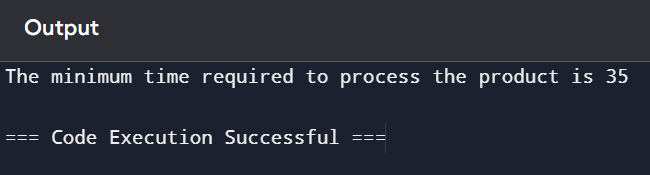
T1[i] = min(T1[i - 1] + a1[i], T2[i - 1] + t2[i] + a1[i])

T2[i] = min(T2[i - 1] + a2[i], T1[i - 1] + t1[i] + a2[i])

final\_time = min(T1[n - 1] + x1, T2[n - 1] + x2)

print("The minimum time required to process the product is", final\_time)

OUTPUT:



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:

n = 3

a1 = [5, 9, 3]

a2 = [6, 8, 4]

a3 = [7, 6, 5]

t = [

[0, 2, 3],

[2, 0, 4],

[3, 4, 0]

]

dp1 = [0] \* n

dp2 = [0] \* n

dp3 = [0] \* n

dp1[0] = a1[0]

dp2[0] = a2[0]

dp3[0] = a3[0]

for i in range(1, n):

dp1[i] = min(dp1[i-1] + a1[i], dp2[i-1] + t[1][0] + a1[i], dp3[i-1] + t[2][0] + a1[i])

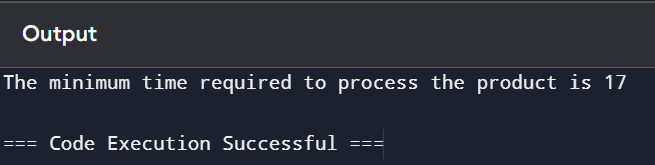
dp2[i] = min(dp2[i-1] + a2[i], dp1[i-1] + t[0][1] + a2[i], dp3[i-1] + t[2][1] + a2[i])

dp3[i] = min(dp3[i-1] + a3[i], dp1[i-1] + t[0][2] + a3[i], dp2[i-1] + t[1][2] + a3[i])

min\_time = min(dp1[n-1], dp2[n-1], dp3[n-1])

print("The minimum time required to process the product is", min\_time)

OUTPUT:



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:

V = 4

graph = [

[0, 10, 15, 20],

[10, 0, 35, 25],

[15, 35, 0, 30],

[20, 25, 30, 0]

]

dp = [[float('inf')] \* V for \_ in range(1 << V)]

dp[1][0] = 0

for mask in range(1 << V):

for u in range(V):

if mask & (1 << u):

for v in range(V):

if not mask & (1 << v):

dp[mask | (1 << v)][v] = min(dp[mask | (1 << v)][v], dp[mask][u] + graph[u][v])

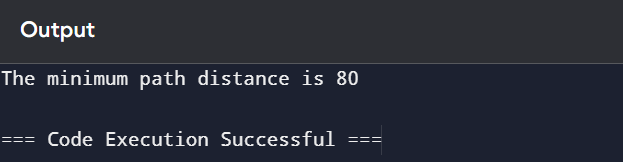
min\_path\_distance = float('inf')

for u in range(1, V):

min\_path\_distance = min(min\_path\_distance, dp[(1 << V) - 1][u] + graph[u][0])

print("The minimum path distance is", min\_path\_distance)

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.

CODE:

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

}

for key in list(distances.keys()):

a, b = key

distances[(b, a)] = distances[(a, b)]

cities = ['A', 'B', 'C', 'D', 'E']

def generate\_permutations(array):

if len(array) == 1:

return [array]

perms = []

for i in range(len(array)):

rest = array[:i] + array[i+1:]

for perm in generate\_permutations(rest):

perms.append([array[i]] + perm)

return perms

other\_cities = cities[1:]

all\_routes = generate\_permutations(other\_cities)

min\_path = []

min\_distance = float('inf')

for route in all\_routes:

current\_path = ['A'] + route + ['A']

total\_distance = 0

for i in range(len(current\_path) - 1):

total\_distance += distances[(current\_path[i], current\_path[i + 1])]

if total\_distance < min\_distance:

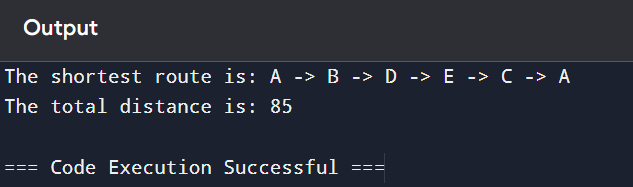
min\_distance = total\_distance

min\_path = current\_path

print("The shortest route is:", ' -> '.join(min\_path))

print("The total distance is:", min\_distance)

OUTPUT:



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.

Example 2:

Input: s = "cbbd"

Output: "bb"

CODE:

s = "babad"

start = 0

end = 0

for i in range(len(s)):

left = i

right = i

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

left -= 1

right += 1

length1 = right - left - 1

left = i

right = i + 1

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

left -= 1

right += 1

length2 = right - left - 1

max\_len = length1 if length1 > length2 else length2

if max\_len > end - start:

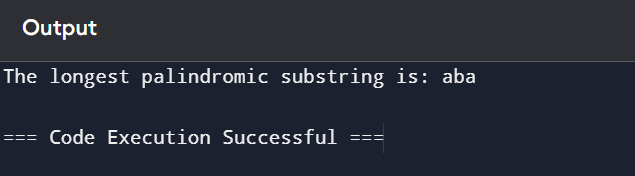
start = i - (max\_len - 1) // 2

end = i + max\_len // 2

longest\_palindrome = s[start:end + 1]

print("The longest palindromic substring is:", longest\_palindrome)

OUTPUT:



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

Example 1:

Input: s = "abcabcbb"

Output: 3

CODE:

s = "abcabcbb"

n = len(s)

max\_len = 0

start = 0

used\_chars = {}

for i in range(n):

if s[i] in used\_chars and start <= used\_chars[s[i]]:

start = used\_chars[s[i]] + 1

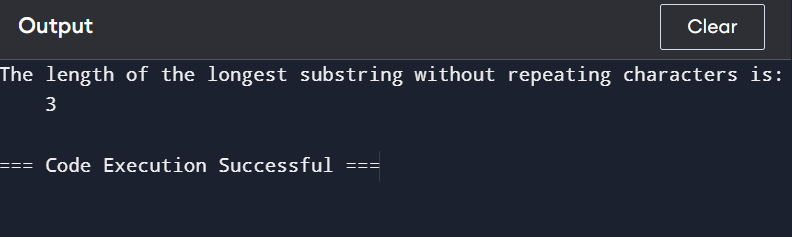
else:

max\_len = max(max\_len, i - start + 1)

used\_chars[s[i]] = i

print("The length of the longest substring without repeating characters is:", max\_len)

OUTPUT:



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:

s = "leetcode"

wordDict = ["leet", "code"]

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):

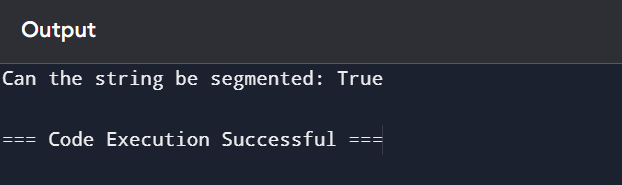
if dp[j] and s[j:i] in word\_set:

dp[i] = True

break

print("Can the string be segmented:", dp[len(s)])

OUTPUT:



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".

CODE:

input\_strings = ["ilike", "ilikesamsung"]

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

for s in input\_strings:

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\_dict:

dp[i] = True

break

print("Can the string '{}' be segmented?".format(s), "Yes" if dp[len(s)] else "No")

if dp[len(s)]:

segmentation = []

i = len(s)

while i > 0:

for j in range(i):

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

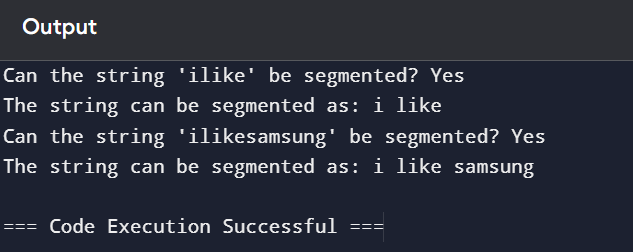
segmentation.append(s[j:i])

i = j

break

print("The string can be segmented as:", ' '.join(segmentation[::-1]))

OUTPUT:



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:

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

maxWidth = 16

lines = []

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)] += ' '

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

current\_line = []

current\_length = 0

current\_line.append(word)

current\_length += len(word)

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

for line in lines:

print(f'"{line}"')

OUTPUT:

