## TOPIC 4 : DYNAMIC PROGRAMMING

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

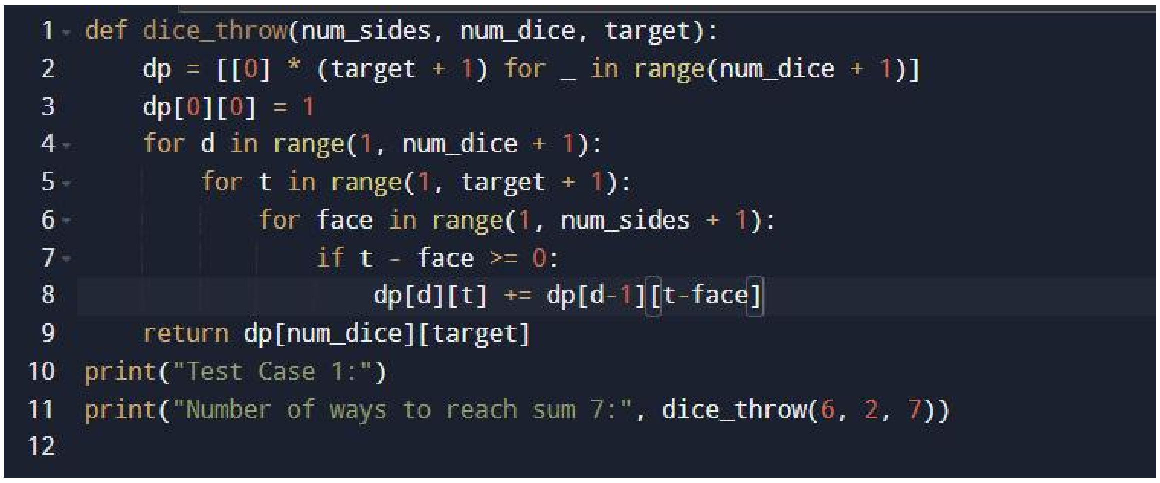
   Output

   Test Case 1:

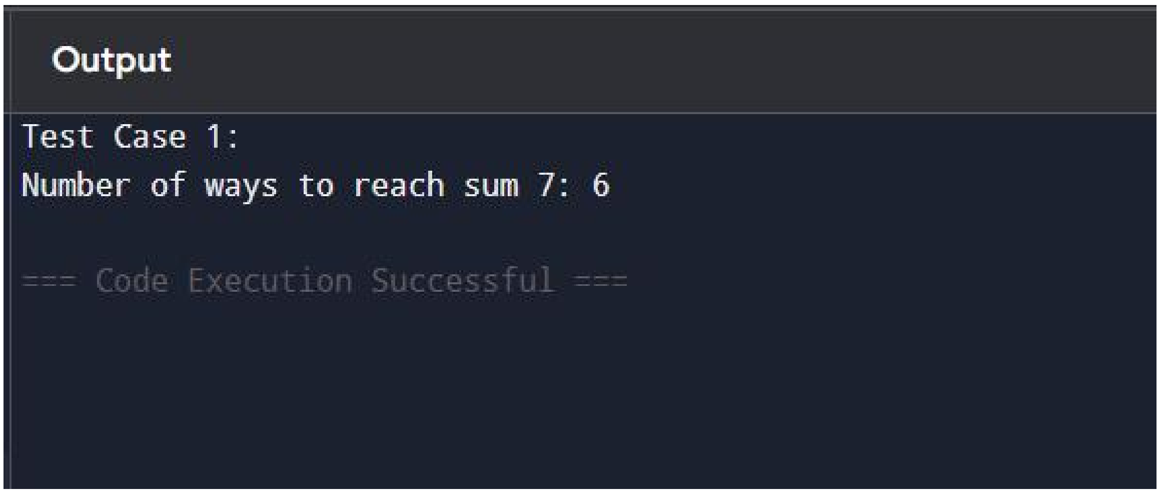
   Number of ways to reach sum 7: 6

   Test Case 2:

   Number of ways to reach sum 10: 27



```python
def dice_throw(num_sides, num_dice, target):
    dp = [[0] * (target + 1) for _ in range(num_dice + 1)]
    dp[0][0] = 1
    for d 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[d][t] += dp[d-1][t-face]
    return dp[num_dice][target]
print("Test Case 1:")
print("Number of ways to reach sum 7:", dice_throw(6, 2, 7))

```



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

=== Code Execution Successful ===
```

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.

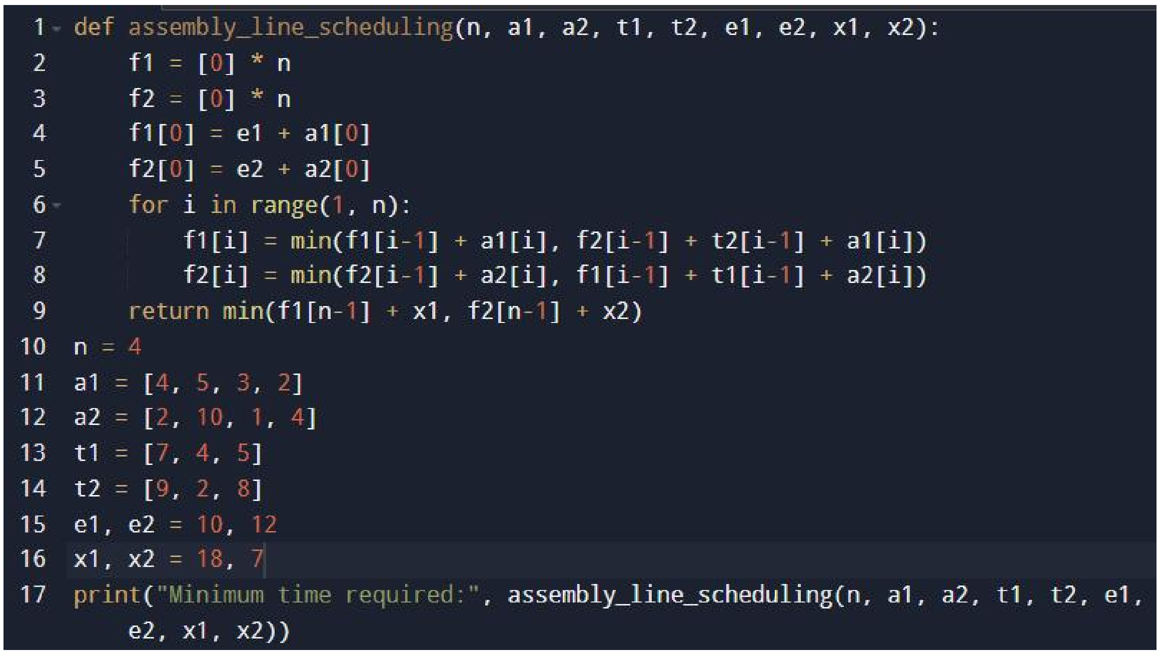
   e2: Entry time to assembly line 2.

   x1: Exit time from assembly line 1.

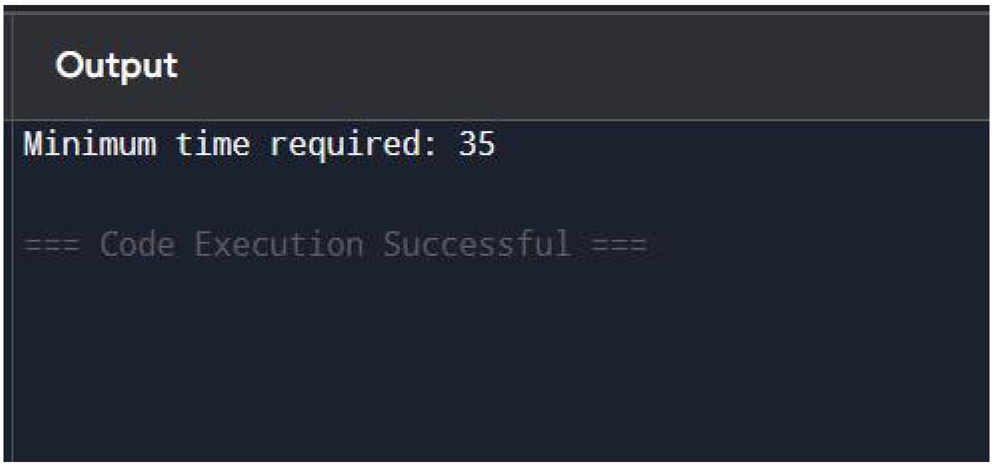
   x2: Exit time from assembly line 2.

   Output

   The minimum time required to process the product.



```python
def assembly_line_scheduling(n, a1, a2, t1, t2, e1, e2, x1, x2):
    f1 = [0] * n
    f2 = [0] * n
    f1[0] = e1 + a1[0]
    f2[0] = e2 + a2[0]
    for i in range(1, n):
        f1[i] = min(f1[i-1] + a1[i], f2[i-1] + t2[i-1] + a1[i])
        f2[i] = min(f2[i-1] + a2[i], f1[i-1] + t1[i-1] + a2[i])
    return min(f1[n-1] + x1, f2[n-1] + x2)
n = 4
a1 = [4, 5, 3, 2]
a2 = [2, 10, 1, 4]
t1 = [7, 4, 5]
t2 = [9, 2, 8]
e1, e2 = 10, 12
x1, x2 = 18, 7
print("Minimum time required:", assembly_line_scheduling(n, a1, a2, t1, t2, e1,
    e2, x1, x2))
```



**Output**

```
Minimum time required: 35


=== Code Execution Successful ===
```

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:

[

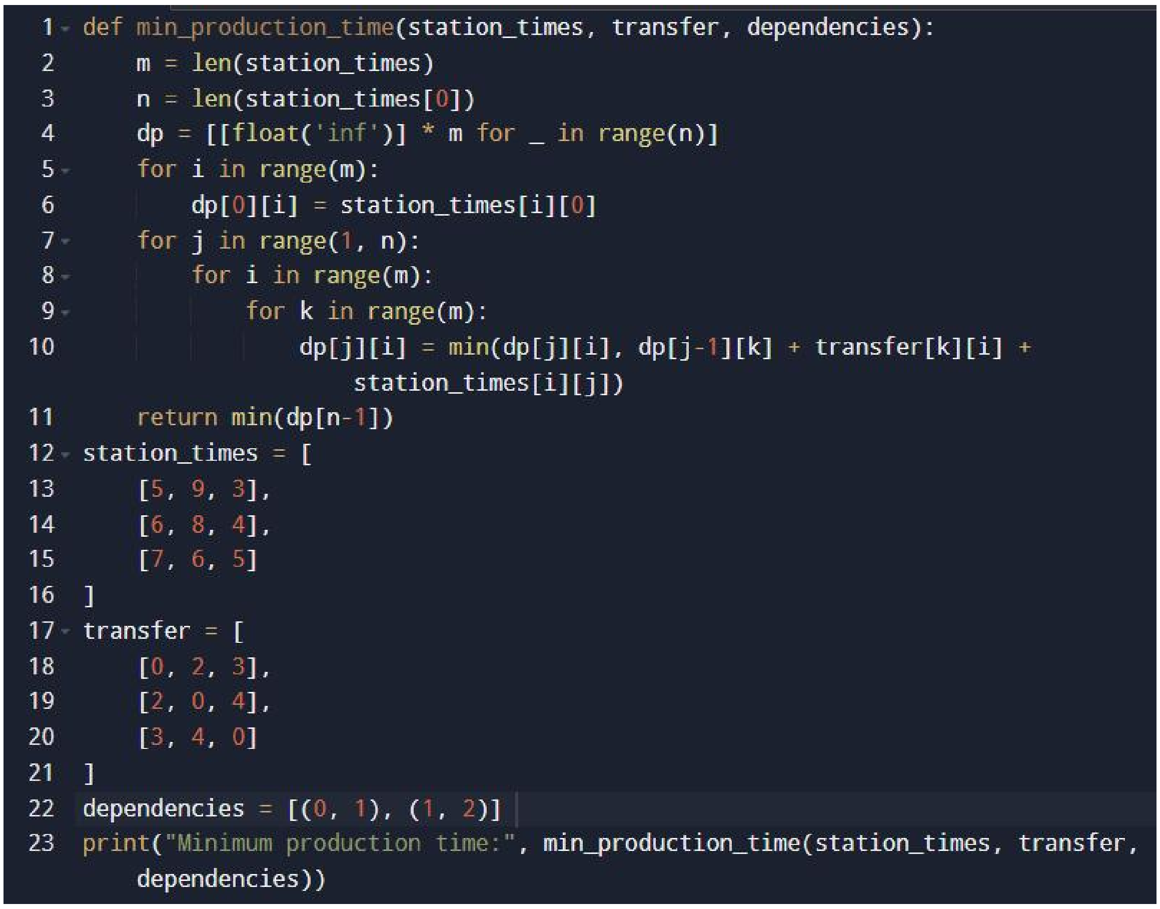
  [0, 2, 3],

  [2, 0, 4],

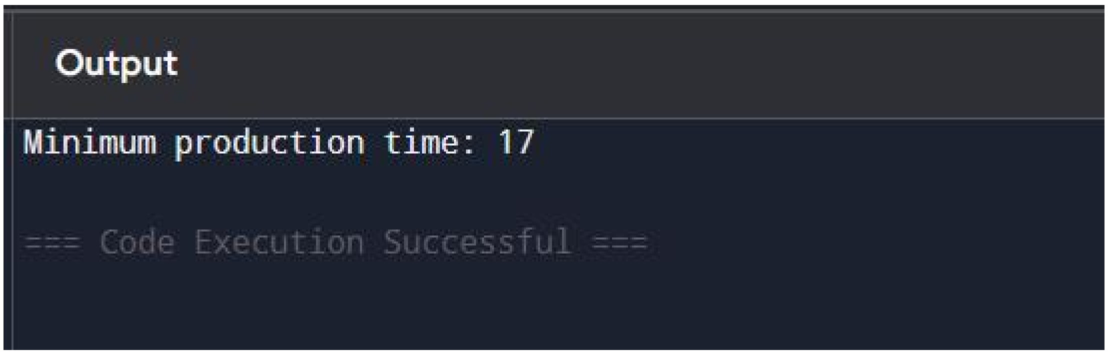
  [3, 4, 0]

]

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).



```python
def min_production_time(station_times, transfer, dependencies):
    m = len(station_times)
    n = len(station_times[0])
    dp = [[float('inf')] * m for _ in range(n)]
    for i in range(m):
        dp[0][i] = station_times[i][0]
    for j in range(1, n):
        for i in range(m):
            for k in range(m):
                dp[j][i] = min(dp[j][i], dp[j-1][k] + transfer[k][i] +
                    station_times[i][j])
    return min(dp[n-1])
station_times = [
    [5, 9, 3],
    [6, 8, 4],
    [7, 6, 5]
]
transfer = [
    [0, 2, 3],
    [2, 0, 4],
    [3, 4, 0]
]
dependencies = [(0, 1), (1, 2)]
print("Minimum production time:", min_production_time(station_times, transfer,
    dependencies))
```



Output

```
Minimum production time: 17


=== Code Execution Successful ===
```

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

2)

{0,10,10,10}

{10,0,10,10}

{10,10,0,10}

{10,10,10,0}

Output: 40

3)

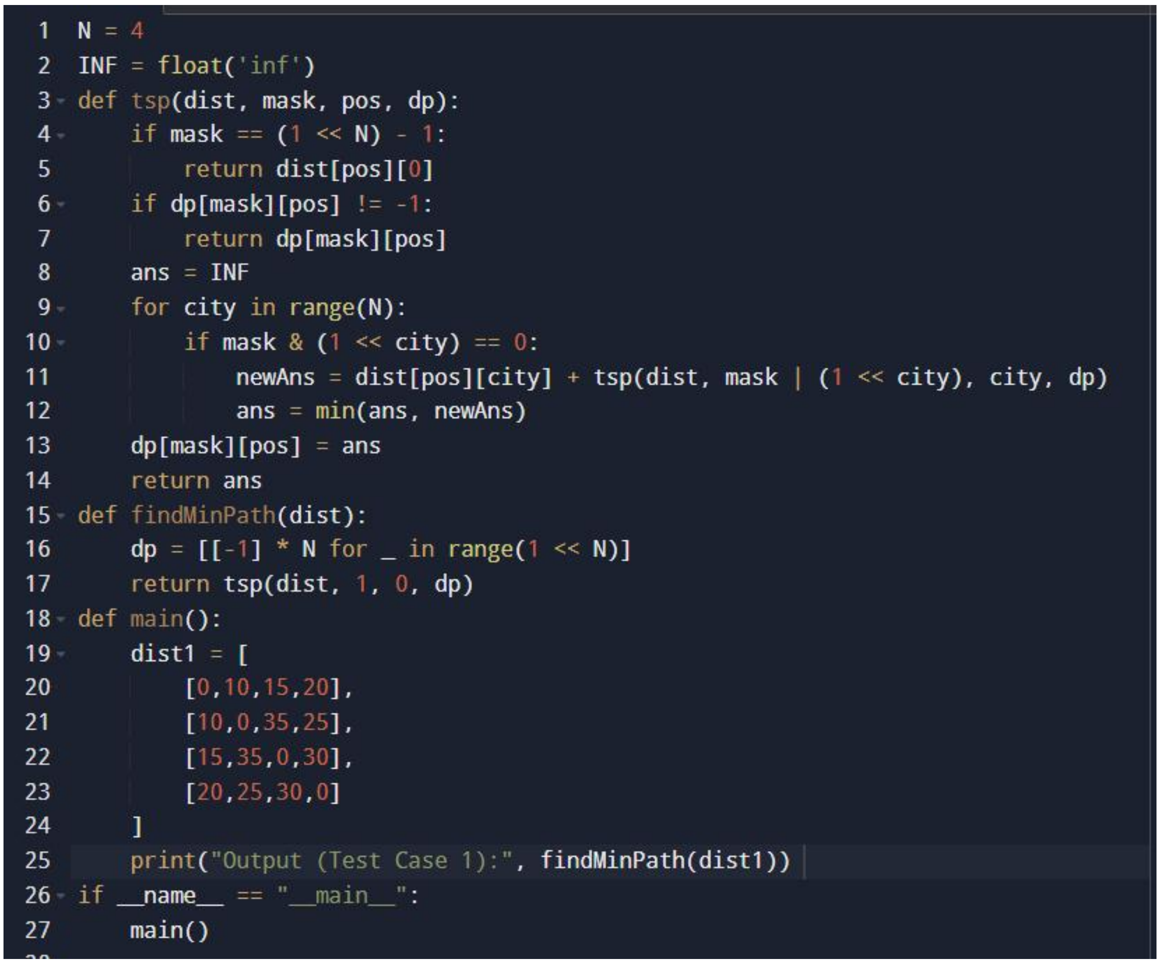
{0,1,2,3}

{1,0,4,5}

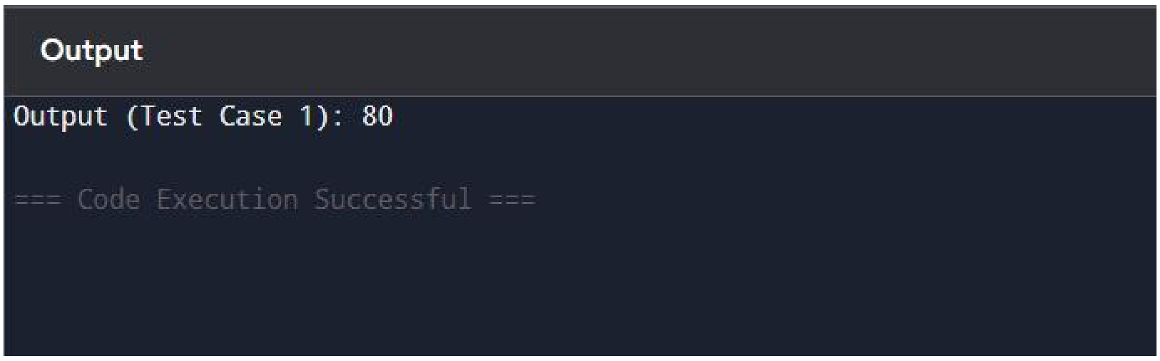
{2,4,0,6}

{3,5,6,0}

Output: 12



```python
N = 4
INF = float('inf')
def tsp(dist, mask, pos, dp):
    if mask == (1 << N) - 1:
        return dist[pos][0]
    if dp[mask][pos] != -1:
        return dp[mask][pos]
    ans = INF
    for city in range(N):
        if mask & (1 << city) == 0:
            newAns = dist[pos][city] + tsp(dist, mask | (1 << city), city, dp)
            ans = min(ans, newAns)
    dp[mask][pos] = ans
    return ans
def findMinPath(dist):
    dp = [[-1] * N for _ in range(1 << N)]
    return tsp(dist, 1, 0, dp)
def main():
    dist1 = [
        [0,10,15,20],
        [10,0,35,25],
        [15,35,0,30],
        [20,25,30,0]
    ]
    print("Output (Test Case 1):", findMinPath(dist1))
if __name__ == "__main__":
    main()
```



Output

```
Output (Test Case 1): 80


=== Code Execution Successful ===
```

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

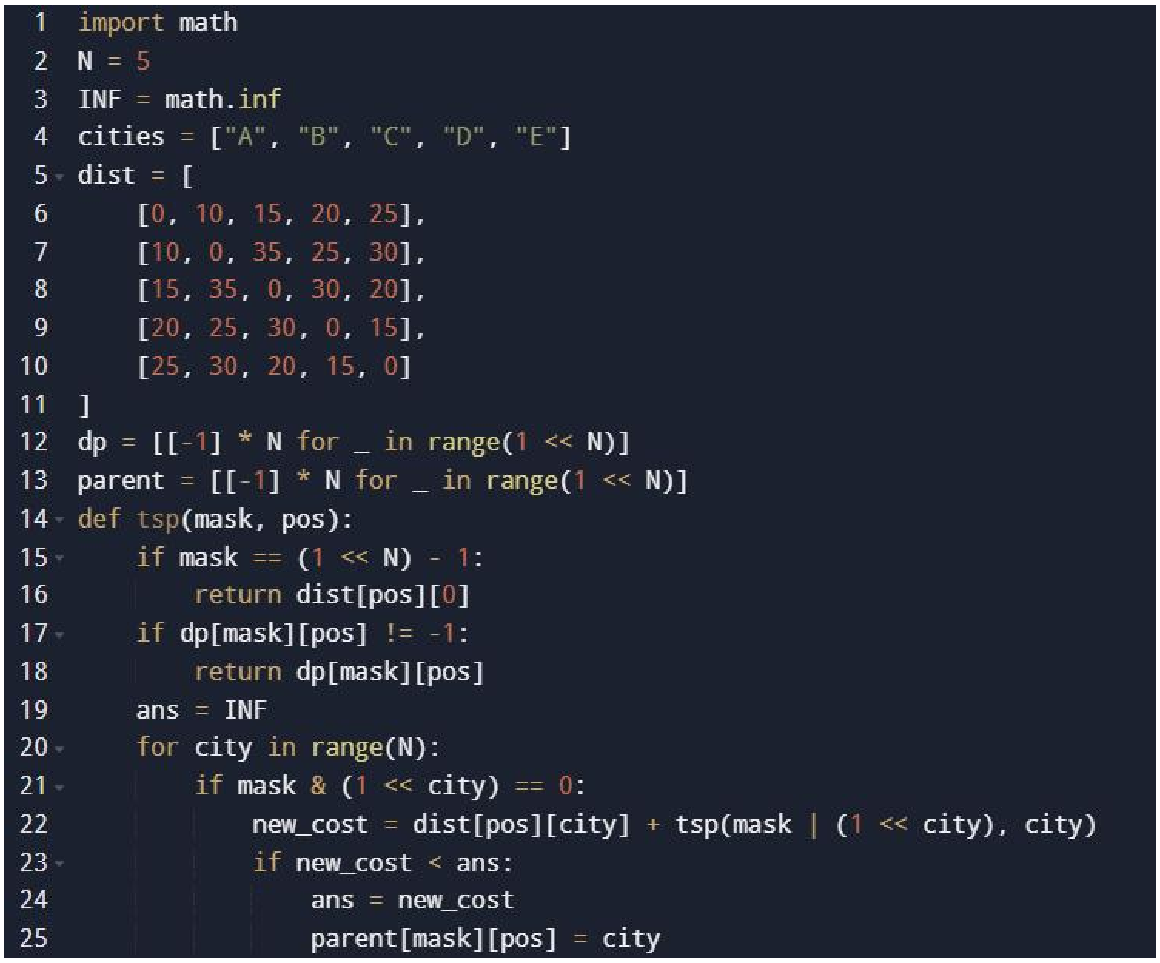
1. Symmetric Distances

- Description: All distances are symmetric (distance from A to B is the same as B to A).

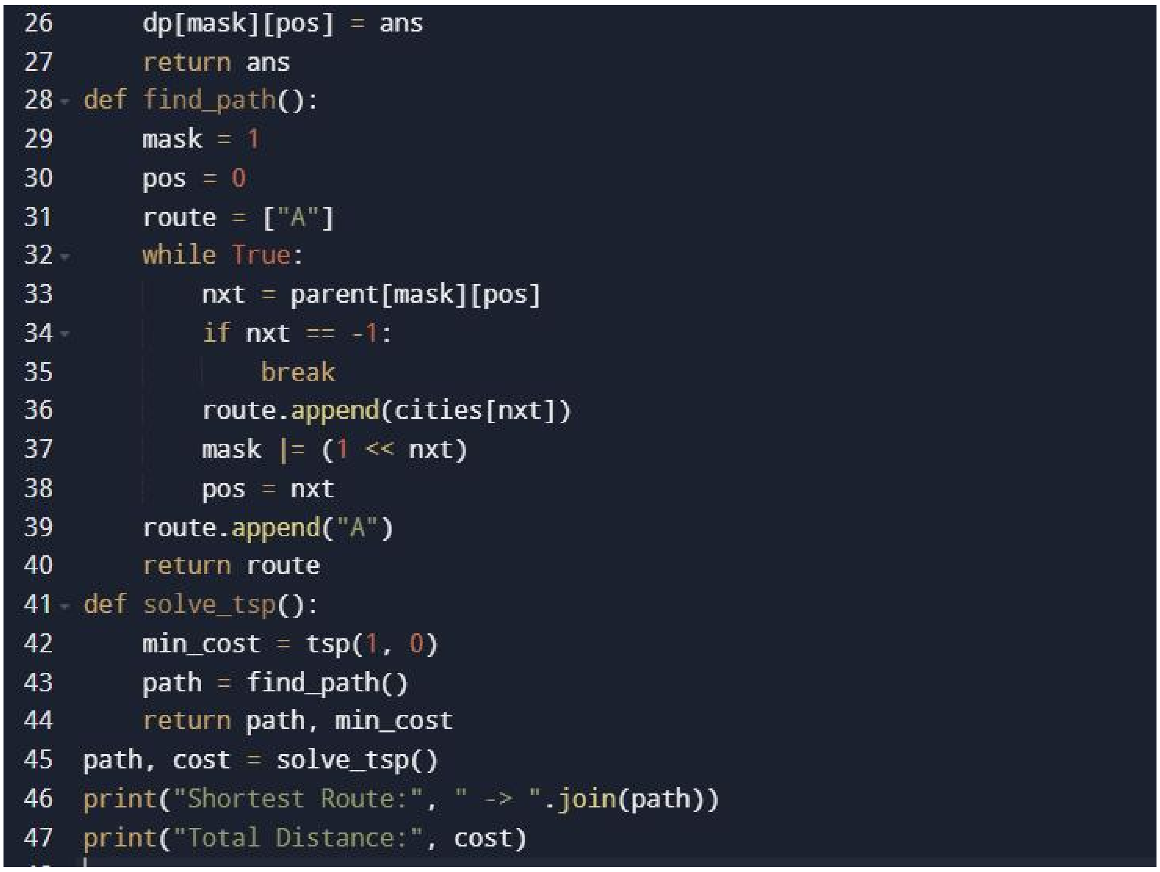
Distances:

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

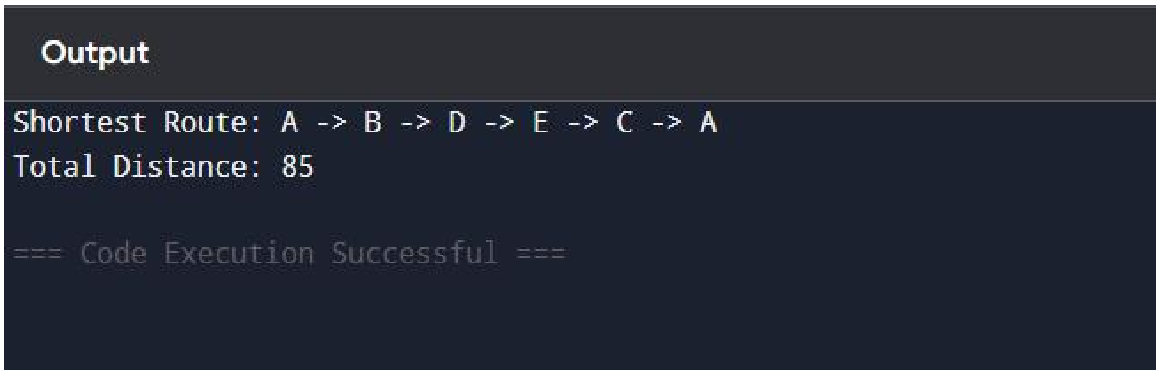
Expected Output: The shortest route and its total distance. For example, A -> B -> D -> E -> C -> A might be the shortest route depending on the given distances.



```python
import math
N = 5
INF = math.inf
cities = ["A", "B", "C", "D", "E"]
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] * N for _ in range(1 << N)]
parent = [[-1] * N for _ in range(1 << N)]
def tsp(mask, pos):
    if mask == (1 << N) - 1:
        return dist[pos][0]
    if dp[mask][pos] != -1:
        return dp[mask][pos]
    ans = INF
    for city in range(N):
        if mask & (1 << city) == 0:
            new_cost = dist[pos][city] + tsp(mask | (1 << city), city)
            if new_cost < ans:
                ans = new_cost
                parent[mask][pos] = city
```



```python
    dp[mask][pos] = ans
    return ans
def find_path():
    mask = 1
    pos = 0
    route = ["A"]
    while True:
        nxt = parent[mask][pos]
        if nxt == -1:
            break
        route.append(cities[nxt])
        mask |= (1 << nxt)
        pos = nxt
    route.append("A")
    return route
def solve_tsp():
    min_cost = tsp(1, 0)
    path = find_path()
    return path, min_cost
path, cost = solve_tsp()
print("Shortest Route:", " -> ".join(path))
print("Total Distance:", cost)
```



Output

```
Shortest Route: A -> B -> D -> E -> C -> A
Total Distance: 85

=== Code Execution Successful ===
```

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

   Example 1:

   Input: s = "babad"

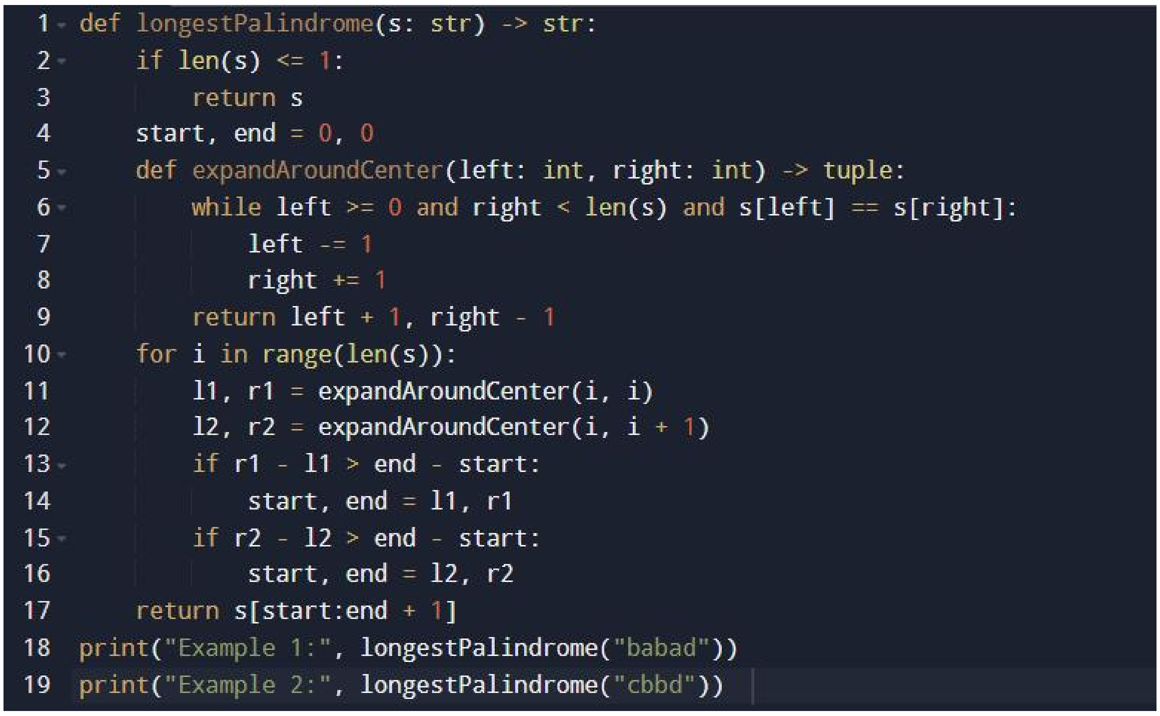
   Output: "bab" Explanation: "aba" is also a valid answer.

   Example 2:

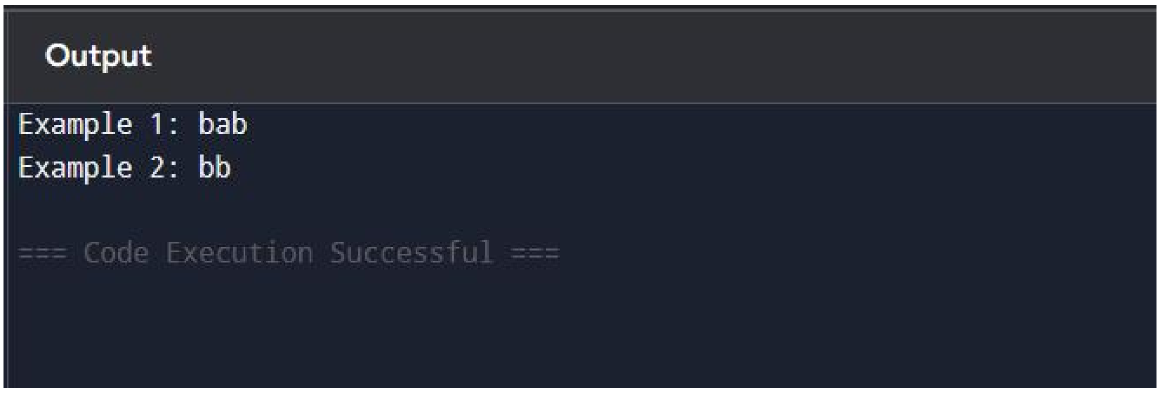
   Input: s = "cbbd"

   Output: "bb"

   Constraints: ● 1 <= s.length <= 1000 ● s consist of only digits and English letters.



```python
def longestPalindrome(s: str) -> str:
    if len(s) <= 1:
        return s
    start, end = 0, 0
    def expandAroundCenter(left: int, right: int) -> tuple:
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return left + 1, right - 1
    for i in range(len(s)):
        l1, r1 = expandAroundCenter(i, i)
        l2, r2 = expandAroundCenter(i, i + 1)
        if r1 - l1 > end - start:
            start, end = l1, r1
        if r2 - l2 > end - start:
            start, end = l2, r2
    return s[start:end + 1]
print("Example 1:", longestPalindrome("babad"))
print("Example 2:", longestPalindrome("cbbd"))
```



Output

```
Example 1: bab
Example 2: bb

=== Code Execution Successful ===
```

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.

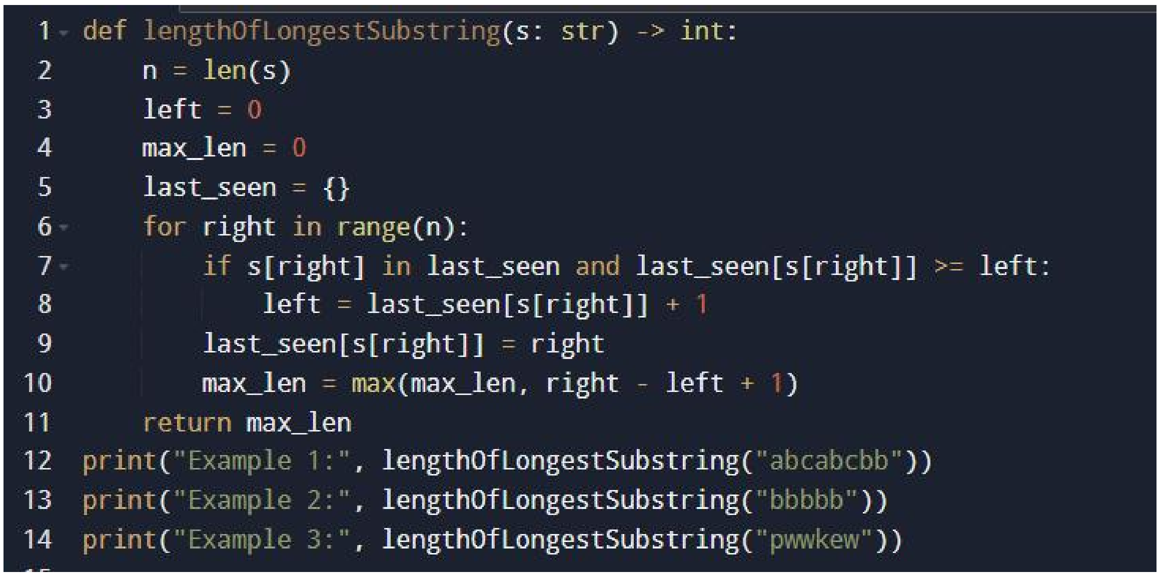
   Example 2: Input: s = "bbbbb" Output: 1

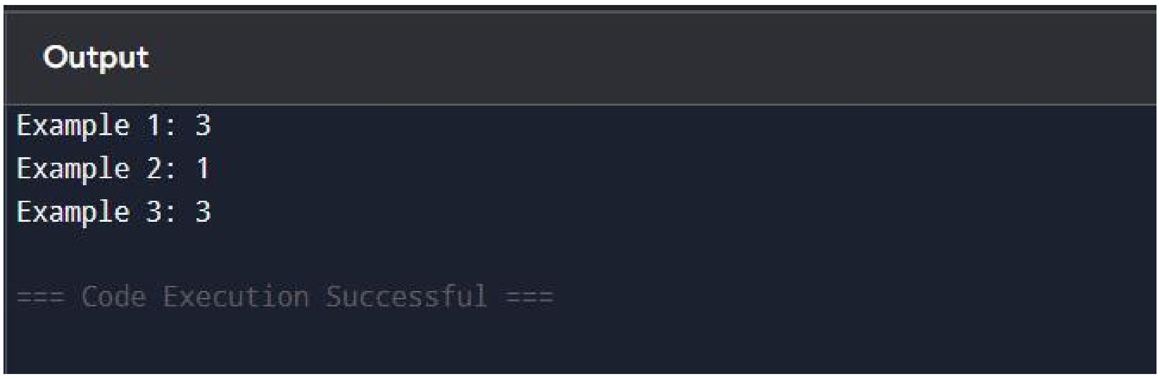
   Explanation: The answer is "b", with the length of 1.

   Example 3: Input: s = "pwwkew" Output: 3

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

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring. Constraints: ● 0 <= s.length <= 5 * 104 ● s consists of English letters, digits, symbols and spaces.


```python
def lengthOfLongestSubstring(s: str) -> int:
    n = len(s)
    left = 0
    max_len = 0
    last_seen = {}
    for right in range(n):
        if s[right] in last_seen and last_seen[s[right]] >= left:
            left = last_seen[s[right]] + 1
        last_seen[s[right]] = right
        max_len = max(max_len, right - left + 1)
    return max_len
print("Example 1:", lengthOfLongestSubstring("abcabcbb"))
print("Example 2:", lengthOfLongestSubstring("bbbbb"))
print("Example 3:", lengthOfLongestSubstring("pwwkew"))
```


Output

```
Example 1: 3
Example 2: 1
Example 3: 3

=== Code Execution Successful ===
```

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

   Explanation: Return true because "leetcode" can be segmented as "leet code".

   Example 2:

   Input: s = "applepenapple", wordDict = ["apple","pen"]

   Output: true

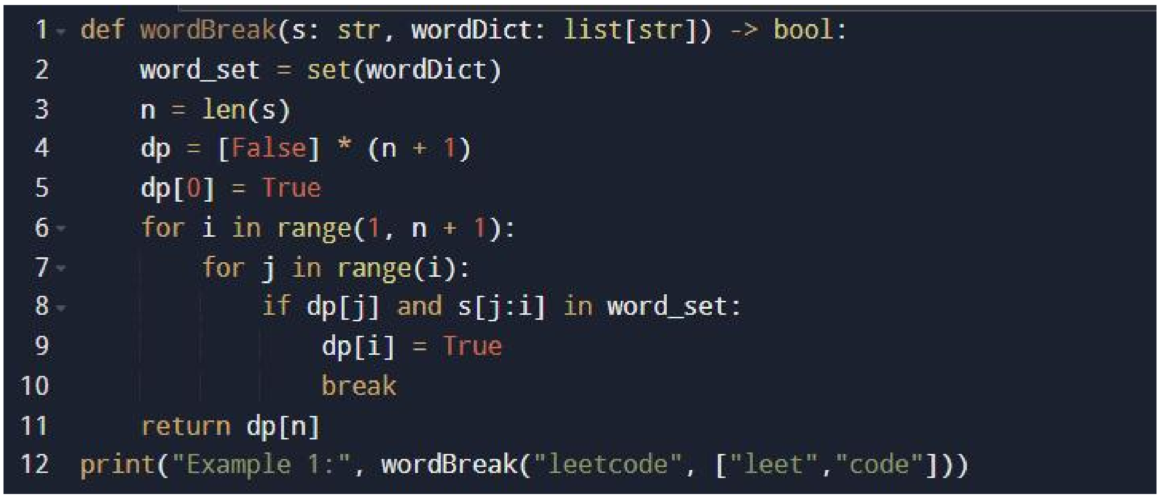
   Explanation: Return true because "applepenapple" can be segmented as "apple pen apple".

   Note that you are allowed to reuse a dictionary word.

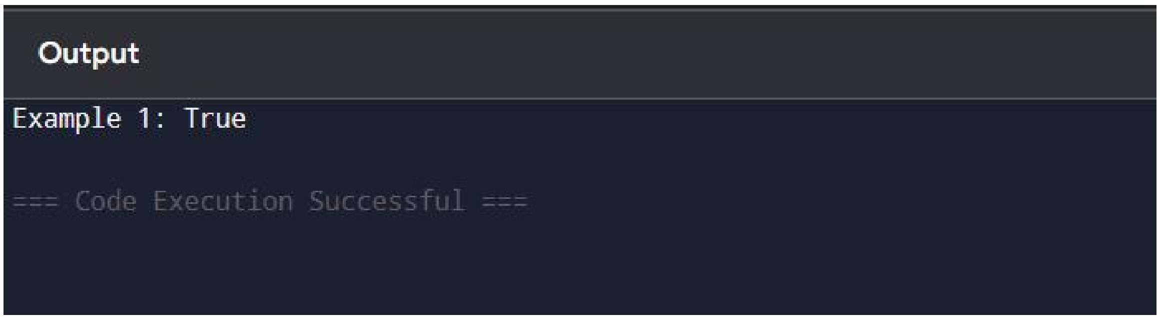
Example 3:

Input: s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]

Output: false



```python
def wordBreak(s: str, wordDict: list[str]) -> bool:
    word_set = set(wordDict)
    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]
print("Example 1:", wordBreak("leetcode", ["leet","code"]))
```



Output

```
Example 1: True

=== Code Execution Successful ===
```

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}

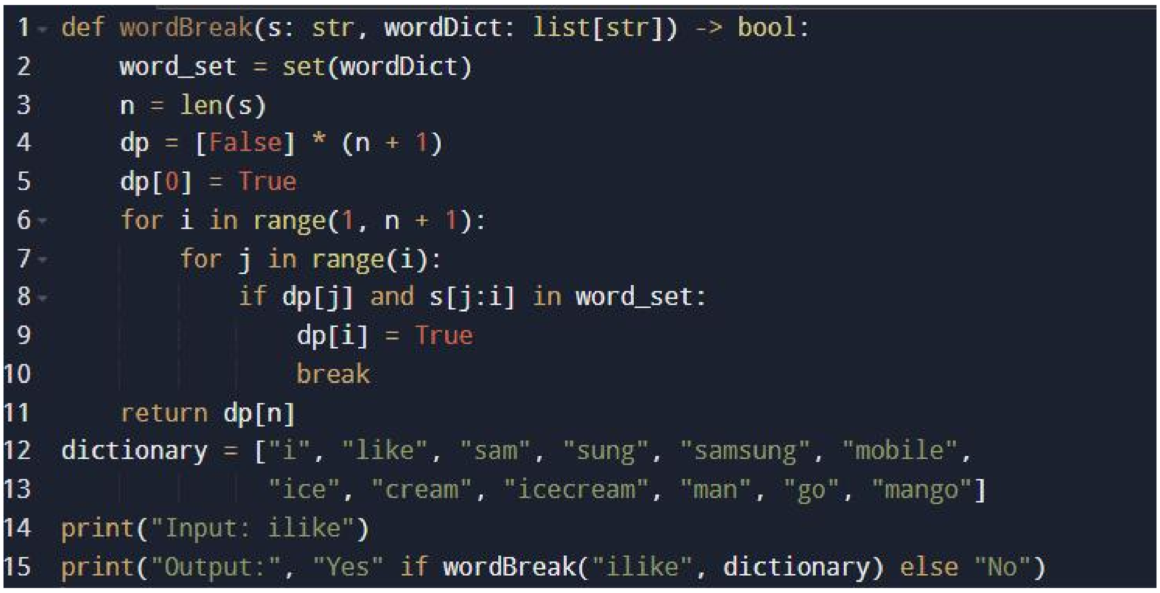
   Input:  ilike

   Output: Yes

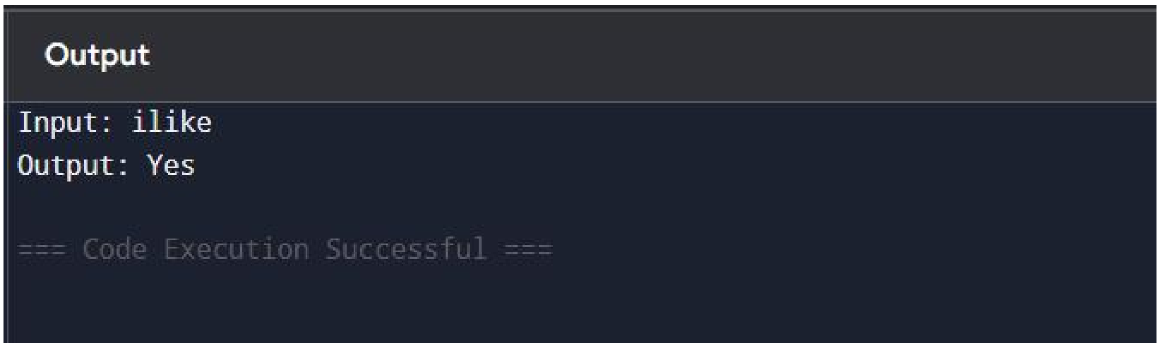
   The string can be segmented as "i like".

   Input:  ilikesamsung

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



```python
def wordBreak(s: str, wordDict: list[str]) -> bool:
    word_set = set(wordDict)
    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]
dictionary = ["i", "like", "sam", "sung", "samsung", "mobile",
              "ice", "cream", "icecream", "man", "go", "mango"]
print("Input: ilike")
print("Output:", "Yes" if wordBreak("ilike", dictionary) else "No")
```



```
Input: ilike
Output: Yes

=== Code Execution Successful ===
```

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.  "
```

]

Example 2:

Input: words = ["What","must","be","acknowledgment","shall","be"], maxWidth = 16

Output:

[

  "What   must   be",

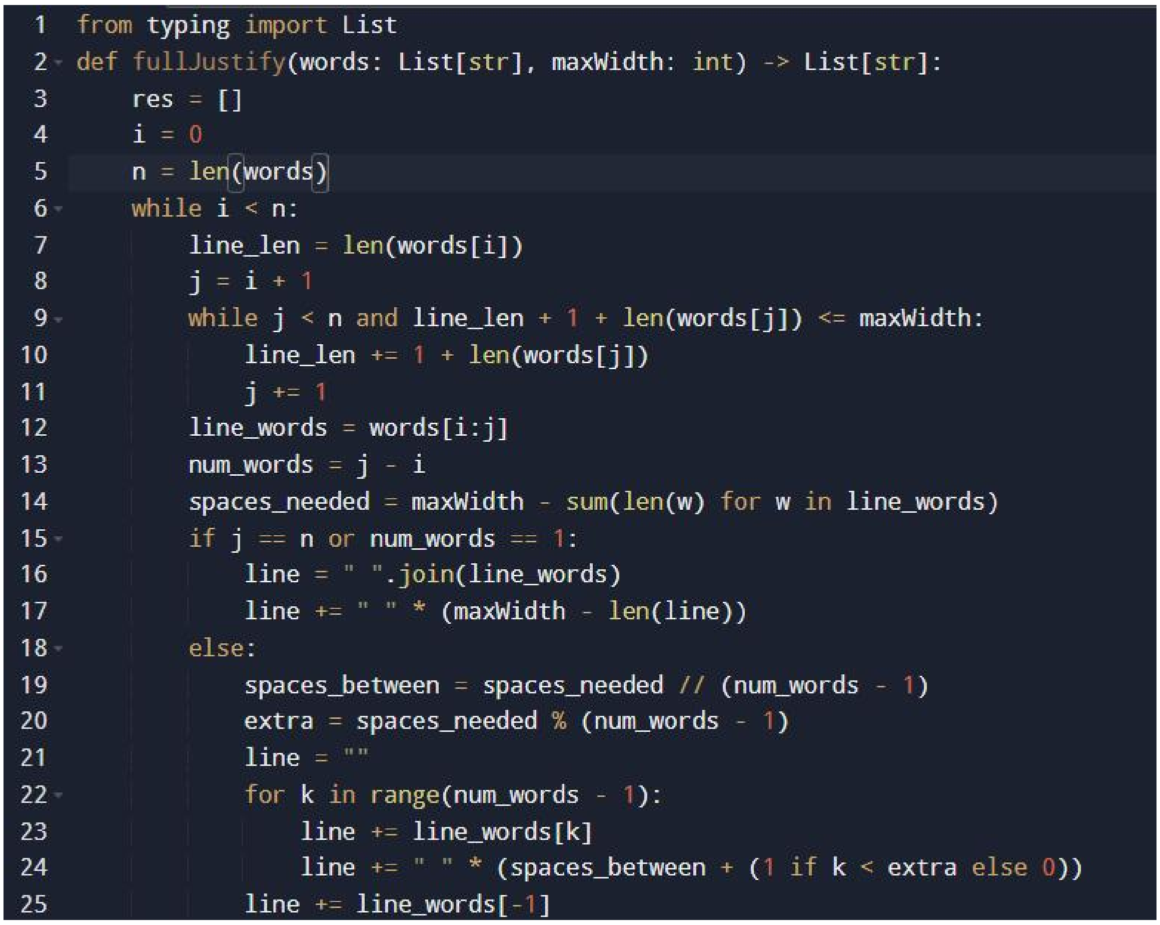
  "acknowledgment  ",

  "shall be        "

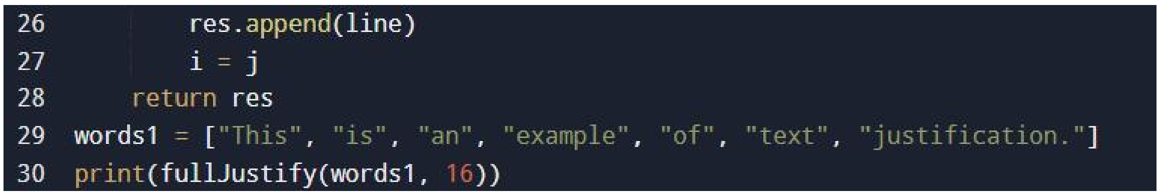
]

Explanation: Note that the last line is "shall be    " instead of "shall     be", because the last line must be left-justified instead of fully-justified.

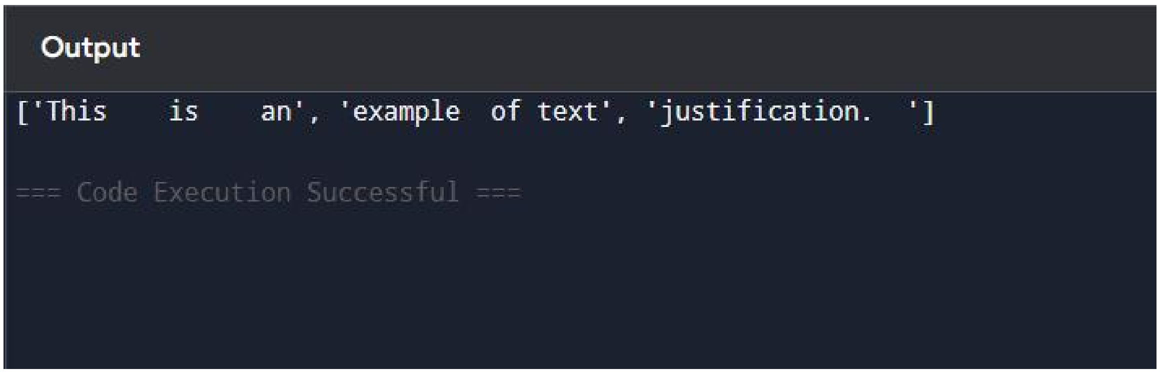
Note that the second line is also left-justified because it contains only one word.



```python
from typing import List
def fullJustify(words: List[str], maxWidth: int) -> List[str]:
    res = []
    i = 0
    n = len(words)
    while i < n:
        line_len = len(words[i])
        j = i + 1
        while j < n and line_len + 1 + len(words[j]) <= maxWidth:
            line_len += 1 + len(words[j])
            j += 1
        line_words = words[i:j]
        num_words = j - i
        spaces_needed = maxWidth - sum(len(w) for w in line_words)
        if j == n or num_words == 1:
            line = " ".join(line_words)
            line += " " * (maxWidth - len(line))
        else:
            spaces_between = spaces_needed // (num_words - 1)
            extra = spaces_needed % (num_words - 1)
            line = ""
            for k in range(num_words - 1):
                line += line_words[k]
                line += " " * (spaces_between + (1 if k < extra else 0))
            line += line_words[-1]
```



```python
        res.append(line)
        i = j
    return res
words1 = ["This", "is", "an", "example", "of", "text", "justification."]
print(fullJustify(words1, 16))
```



```
Output

['This    is    an', 'example  of text', 'justification.  ']

=== Code Execution Successful ===
```

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"]]

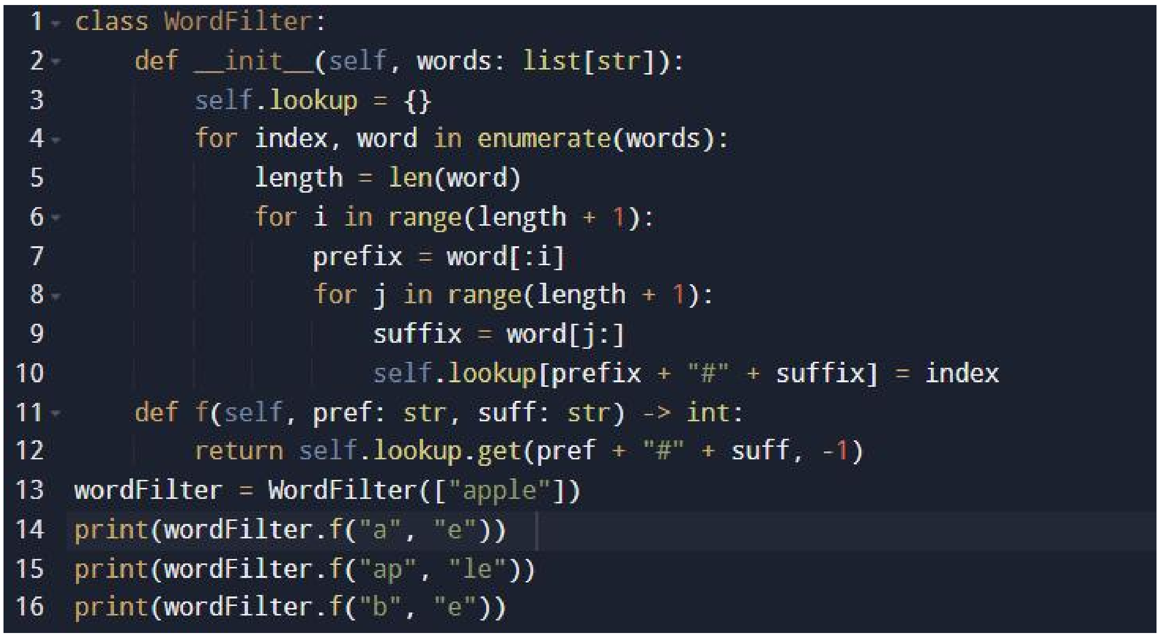
    Output

    [null, 0]

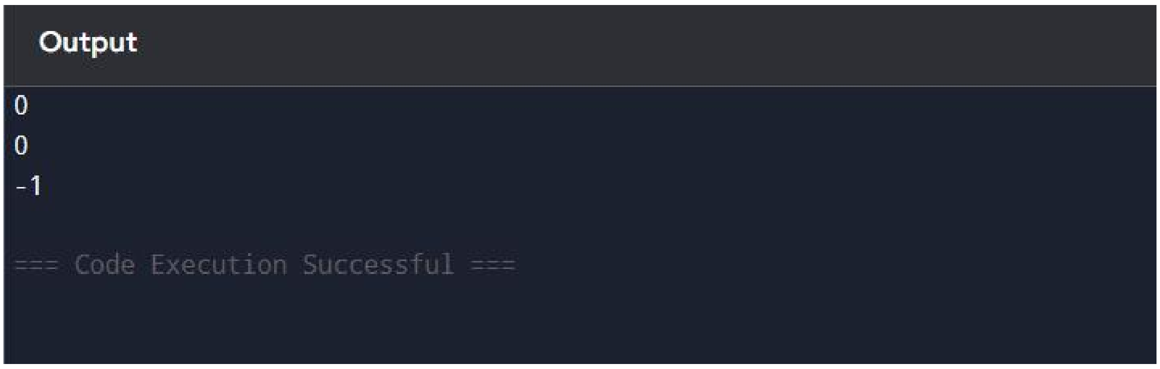
    Explanation

    WordFilter wordFilter = new WordFilter(["apple"]);

    wordFilter.f("a", "e"); // return 0, because the word at index 0 has prefix = "a" and suffix = "e".



```python
class WordFilter:
    def __init__(self, words: list[str]):
        self.lookup = {}
        for index, word in enumerate(words):
            length = len(word)
            for i in range(length + 1):
                prefix = word[:i]
                for j in range(length + 1):
                    suffix = word[j:]
                    self.lookup[prefix + "#" + suffix] = index
    def f(self, pref: str, suff: str) -> int:
        return self.lookup.get(pref + "#" + suff, -1)
wordFilter = WordFilter(["apple"])
print(wordFilter.f("a", "e"))
print(wordFilter.f("ap", "le"))
print(wordFilter.f("b", "e"))
```



```
Output
0
0
-1

=== Code Execution Successful ===
```

12. Implement Floyd's Algorithm to find the shortest path between all pairs of cities. Display the distance matrix before and after applying the algorithm. Identify and print the shortest path

    Input: n = 4, edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]], distanceThreshold = 4

    Output: 3

    Explanation: The figure above describes the graph.

    The neighboring cities at a distanceThreshold = 4 for each city are:

    City 0 -> [City 1, City 2]

    City 1 -> [City 0, City 2, City 3]

    City 2 -> [City 0, City 1, City 3]

    City 3 -> [City 1, City 2]

    Cities 0 and 3 have 2 neighboring cities at a distanceThreshold = 4, but we have to return city 3 since it has the greatest number.

Test cases :

a) You are given a small network of 4 cities connected by roads with the following distances:

City 1 to City 2: 3

City 1 to City 3: 8

City 1 to City 4: -4

City 2 to City 4: 1

City 2 to City 3: 4

City 3 to City 1: 2

City 4 to City 3: -5

City 4 to City 2: 6

Implement Floyd's Algorithm to find the shortest path between all pairs of cities. Display the distance matrix before and after applying the algorithm. Identify and print the shortest path from City 1 to City 3.

Input as above

Output : City 1 to City 3 = -9

b. Consider a network with 6 routers. The initial routing table is as follows:

Router A to Router B: 1

Router A to Router C: 5

Router B to Router C: 2

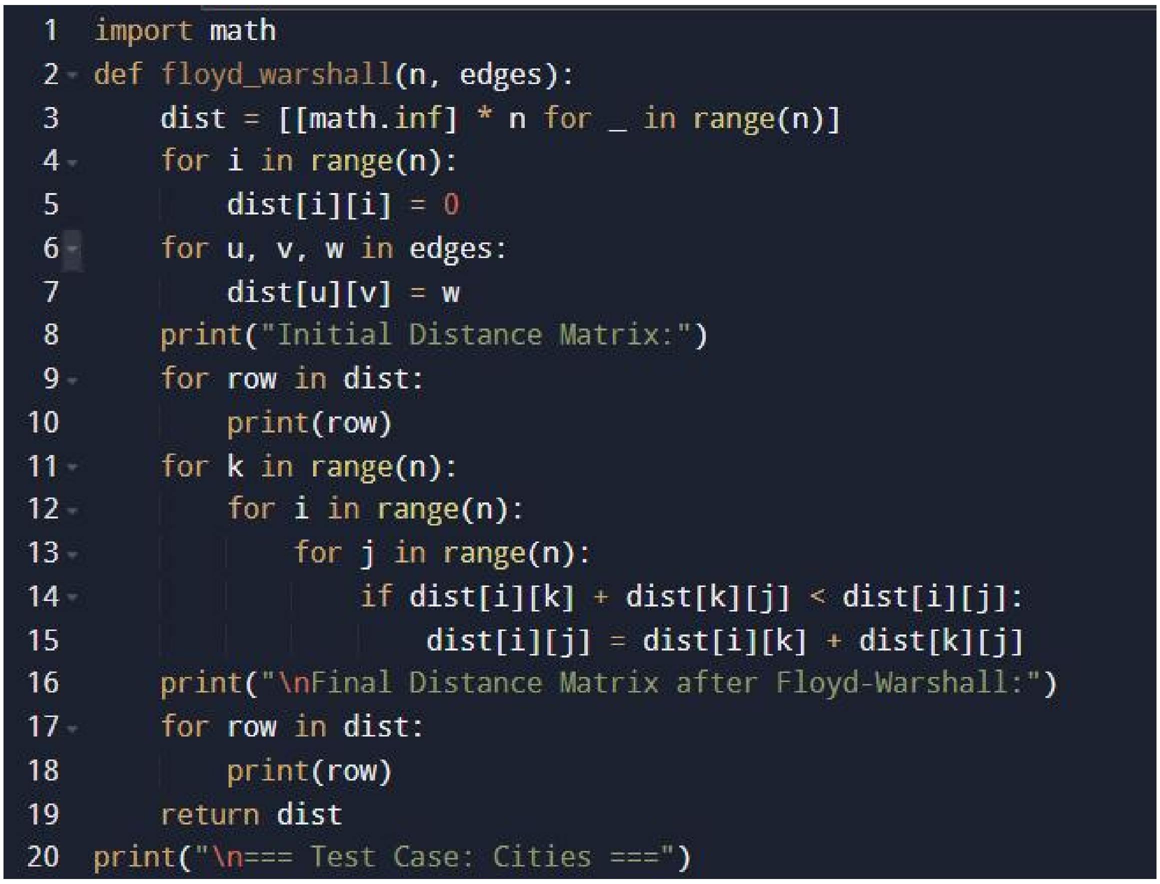
Router B to Router D: 1

Router C to Router E: 3

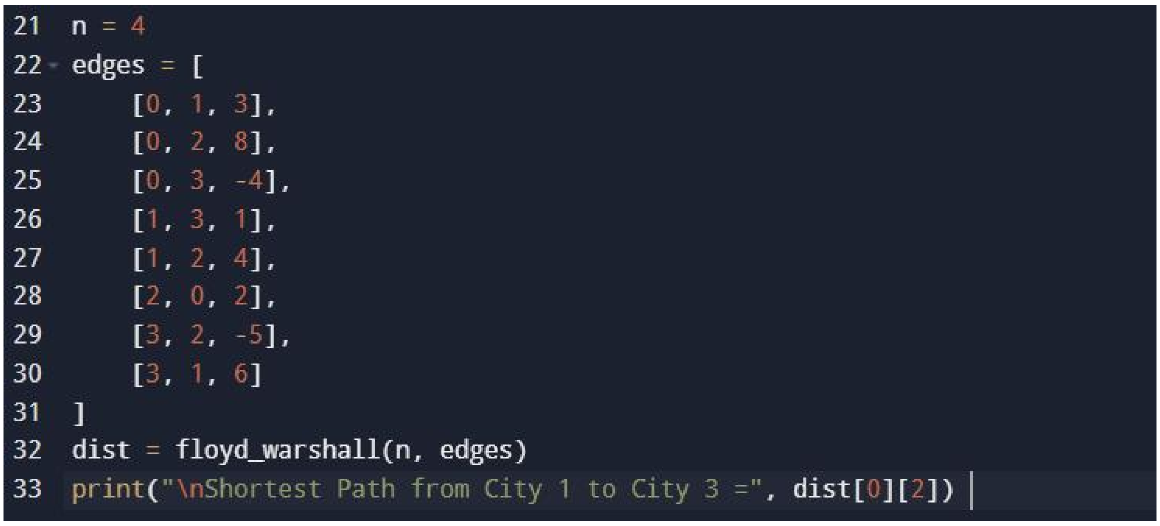
Router D to Router E: 1

Router D to Router F: 6

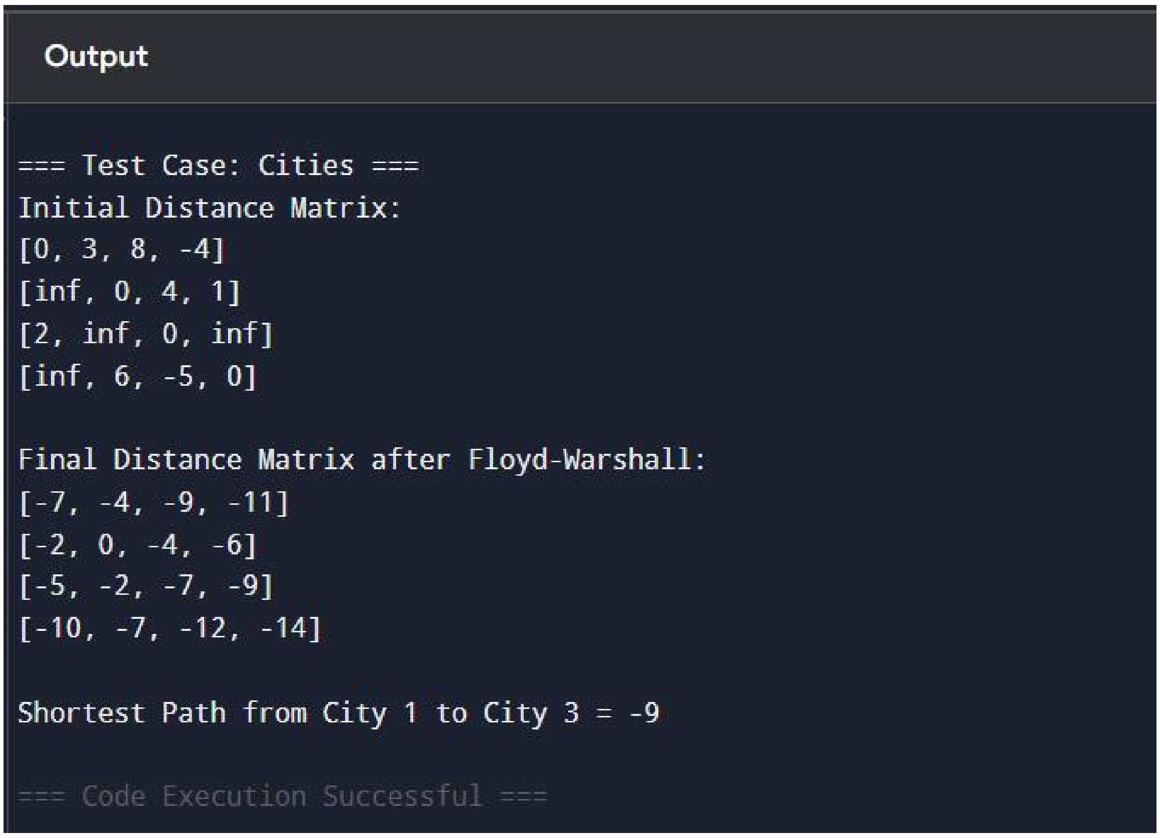
Router E to Router F: 2



```python
import math
def floyd_warshall(n, edges):
    dist = [[math.inf] * n for _ in range(n)]
    for i in range(n):
        dist[i][i] = 0
    for u, v, w in edges:
        dist[u][v] = w
    print("Initial Distance Matrix:")
    for row in dist:
        print(row)
    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]
    print("\nFinal Distance Matrix after Floyd-Warshall:")
    for row in dist:
        print(row)
    return dist
print("\n=== Test Case: Cities ===")
```



```python
n = 4
edges = [
    [0, 1, 3],
    [0, 2, 8],
    [0, 3, -4],
    [1, 3, 1],
    [1, 2, 4],
    [2, 0, 2],
    [3, 2, -5],
    [3, 1, 6]
]
dist = floyd_warshall(n, edges)
print("\nShortest Path from City 1 to City 3 =", dist[0][2])
```

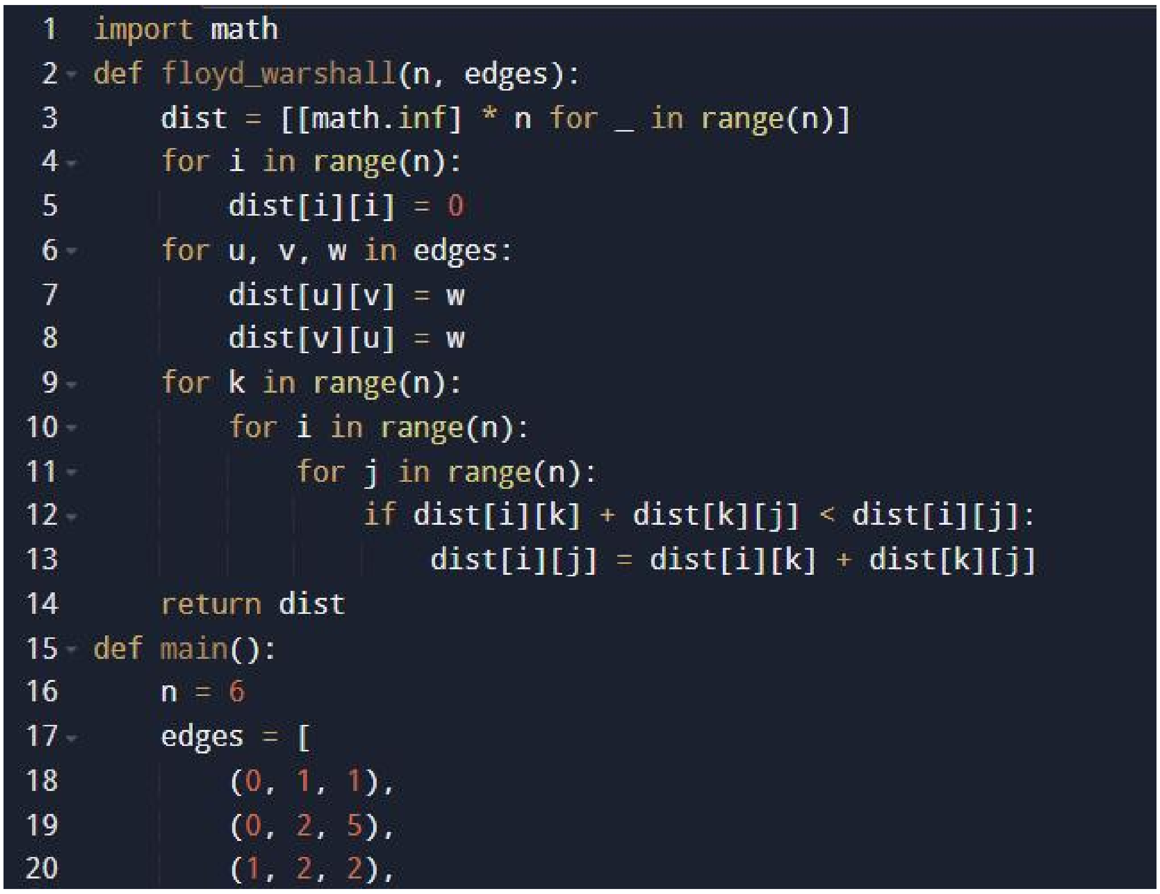

Output

```
=== Test Case: Cities ===
Initial Distance Matrix:
[0, 3, 8, -4]
[inf, 0, 4, 1]
[2, inf, 0, inf]
[inf, 6, -5, 0]

Final Distance Matrix after Floyd-Warshall:
[-7, -4, -9, -11]
[-2, 0, -4, -6]
[-5, -2, -7, -9]
[-10, -7, -12, -14]

Shortest Path from City 1 to City 3 = -9

=== Code Execution Successful ===
```

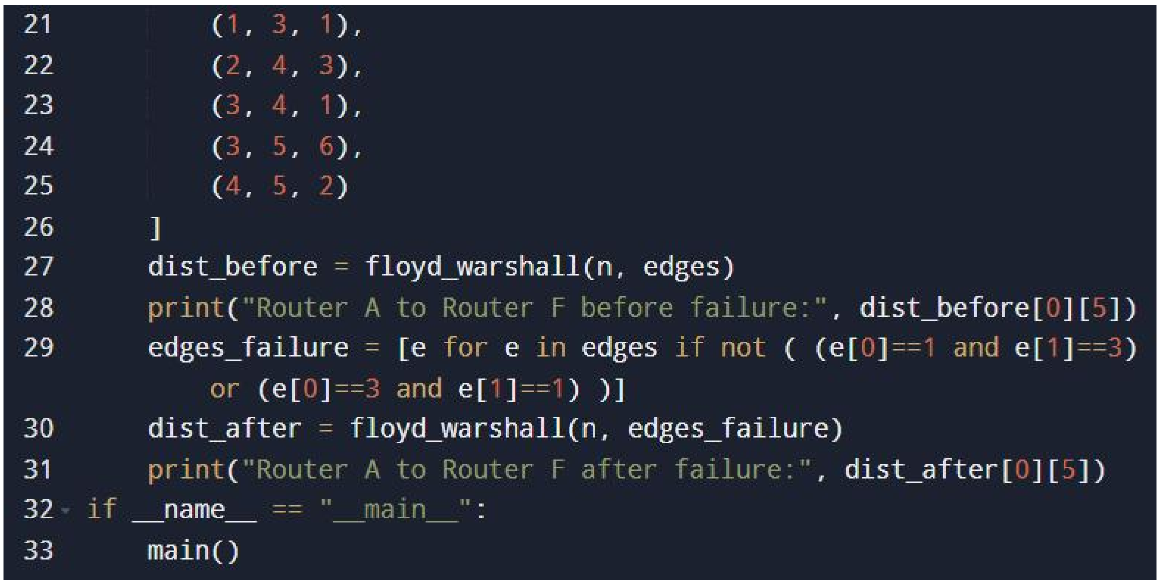
13. Write a Program to implement Floyd's Algorithm to calculate the shortest paths between all pairs of routers. Simulate a change where the link between Router B and Router D fails. Update the distance matrix accordingly. Display the shortest path from Router A to Router F before and after the link failure.

    Input as above

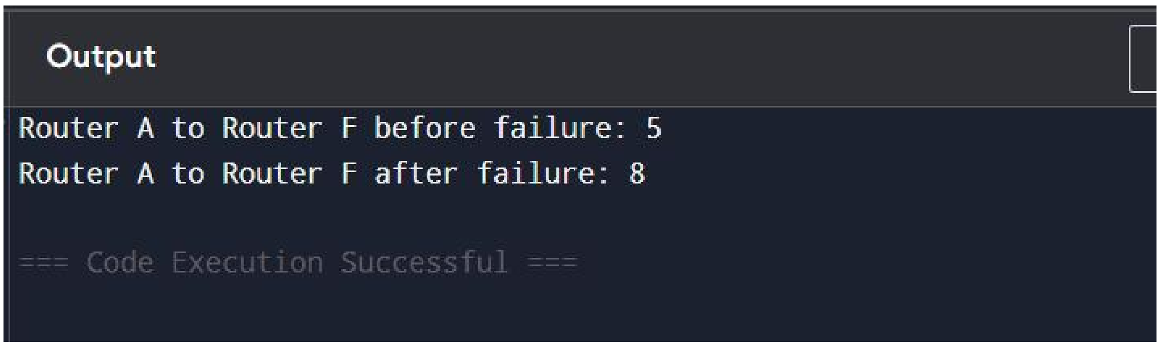
    Output : Router A to Router F = 5



```python
import math
def floyd_warshall(n, edges):
    dist = [[math.inf] * n for _ in range(n)]
    for i in range(n):
        dist[i][i] = 0
    for u, v, w in edges:
        dist[u][v] = w
        dist[v][u] = w
    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
def main():
    n = 6
    edges = [
        (0, 1, 1),
        (0, 2, 5),
        (1, 2, 2),
```



```python
        (1, 3, 1),
        (2, 4, 3),
        (3, 4, 1),
        (3, 5, 6),
        (4, 5, 2)
    ]
    dist_before = floyd_warshall(n, edges)
    print("Router A to Router F before failure:", dist_before[0][5])
    edges_failure = [e for e in edges if not ( (e[0]==1 and e[1]==3) or (e[0]==3 and e[1]==1) )]
    dist_after = floyd_warshall(n, edges_failure)
    print("Router A to Router F after failure:", dist_after[0][5])
if __name__ == "__main__":
    main()
```



**Output**

```
Router A to Router F before failure: 5
Router A to Router F after failure: 8


=== Code Execution Successful ===
```

14. Implement Floyd's Algorithm to find the shortest path between all pairs of cities. Display the distance matrix before and after applying the algorithm. Identify and print the shortest path

    Input: n = 5, edges = [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1],[3,4,1]], distanceThreshold = 2

    Output: 0

    Explanation: The figure above describes the graph.

    The neighboring cities at a distanceThreshold = 2 for each city are:

    City 0 -> [City 1]

    City 1 -> [City 0, City 4]

    City 2 -> [City 3, City 4]

    City 3 -> [City 2, City 4]

    City 4 -> [City 1, City 2, City 3]

    The city 0 has 1 neighboring city at a distanceThreshold = 2.

    a) Test cases :

    B to A 2

    A TO C 3

    C TO D 1

    D TO A 6

    C TO B 7

    Find shortest path from C to A

    Output = 7

    b) Find shortest path from E to C

    C TO A 2

    A TO B 4

    B TO C 1

    B TO E 6

E TO A 1

A TO D 5

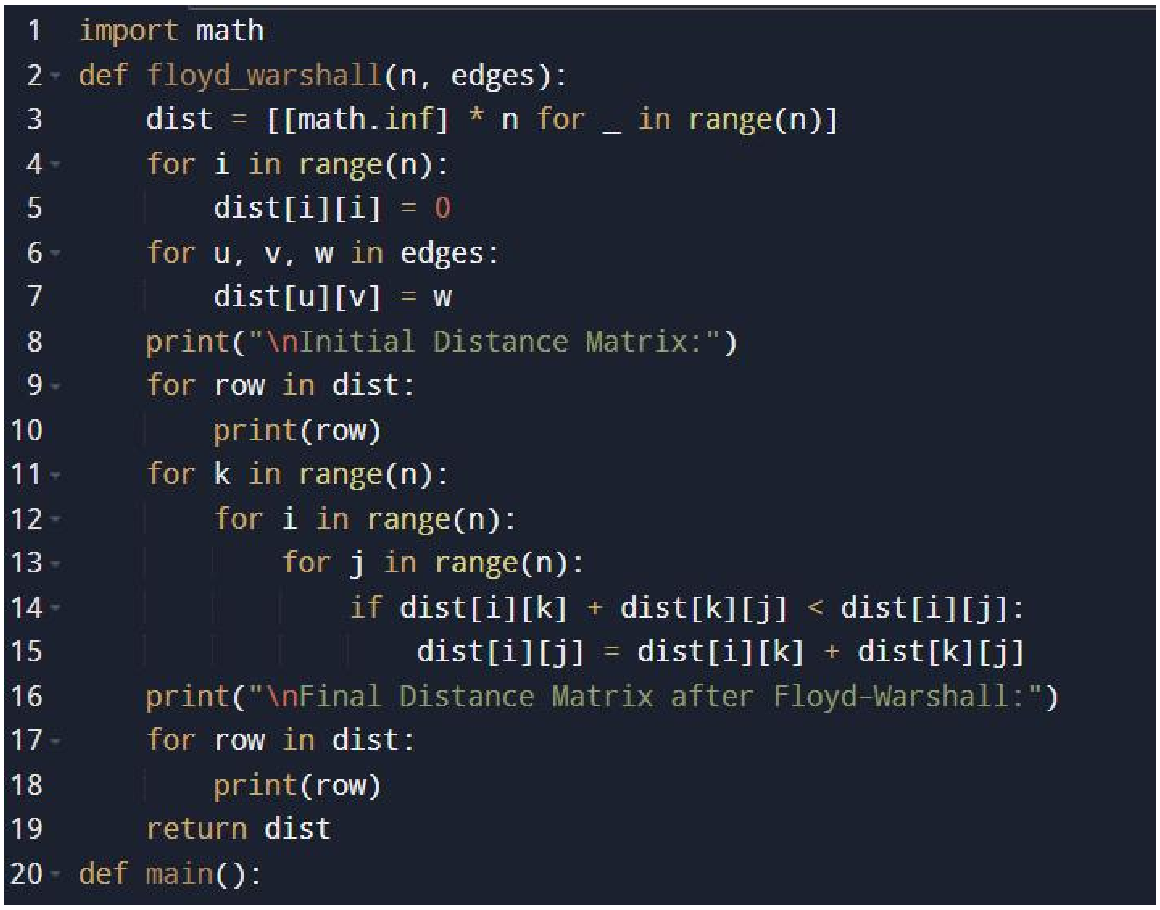
D TO E 2

E TO D 4

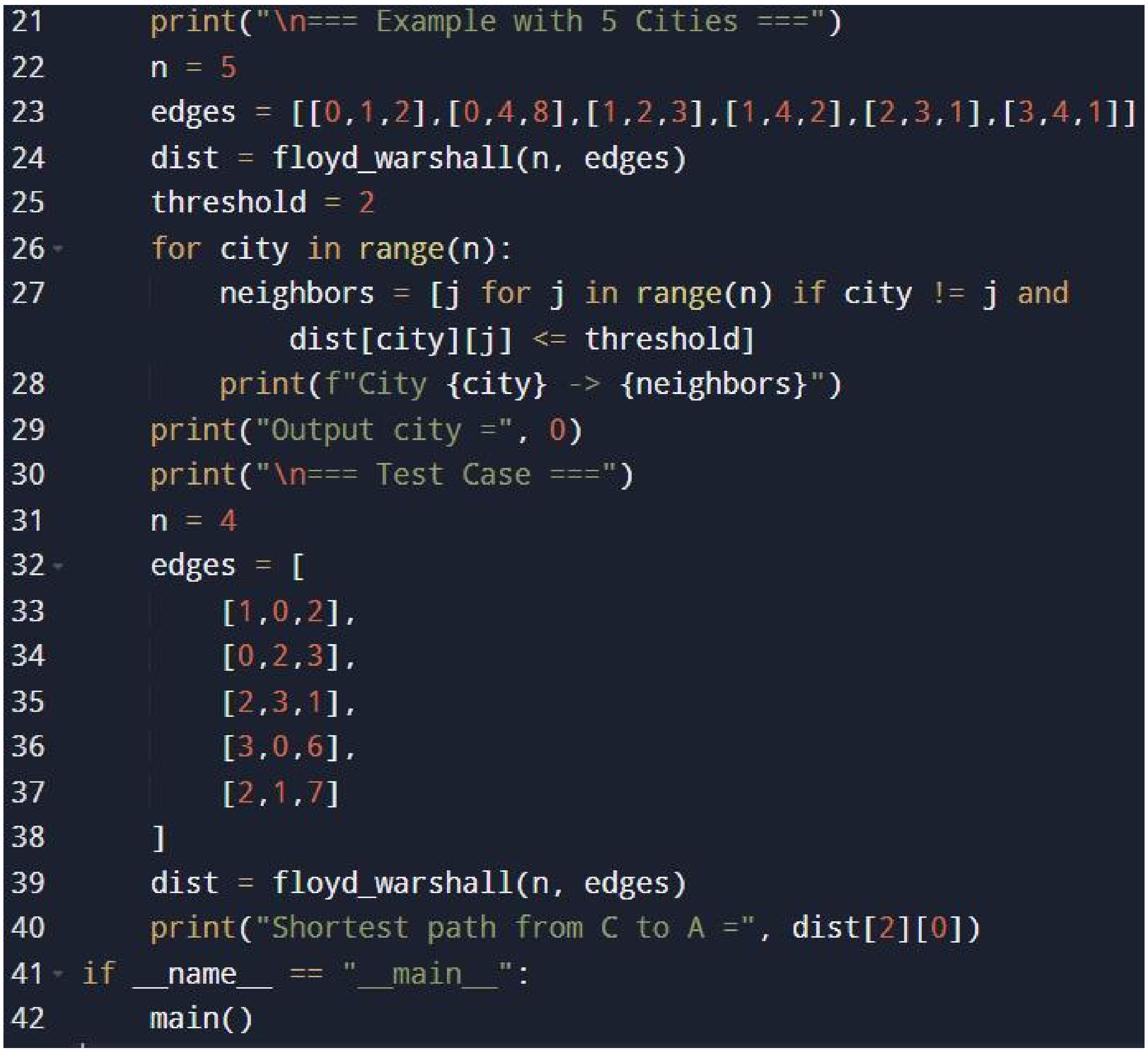
D TO C 1

C TO D 3

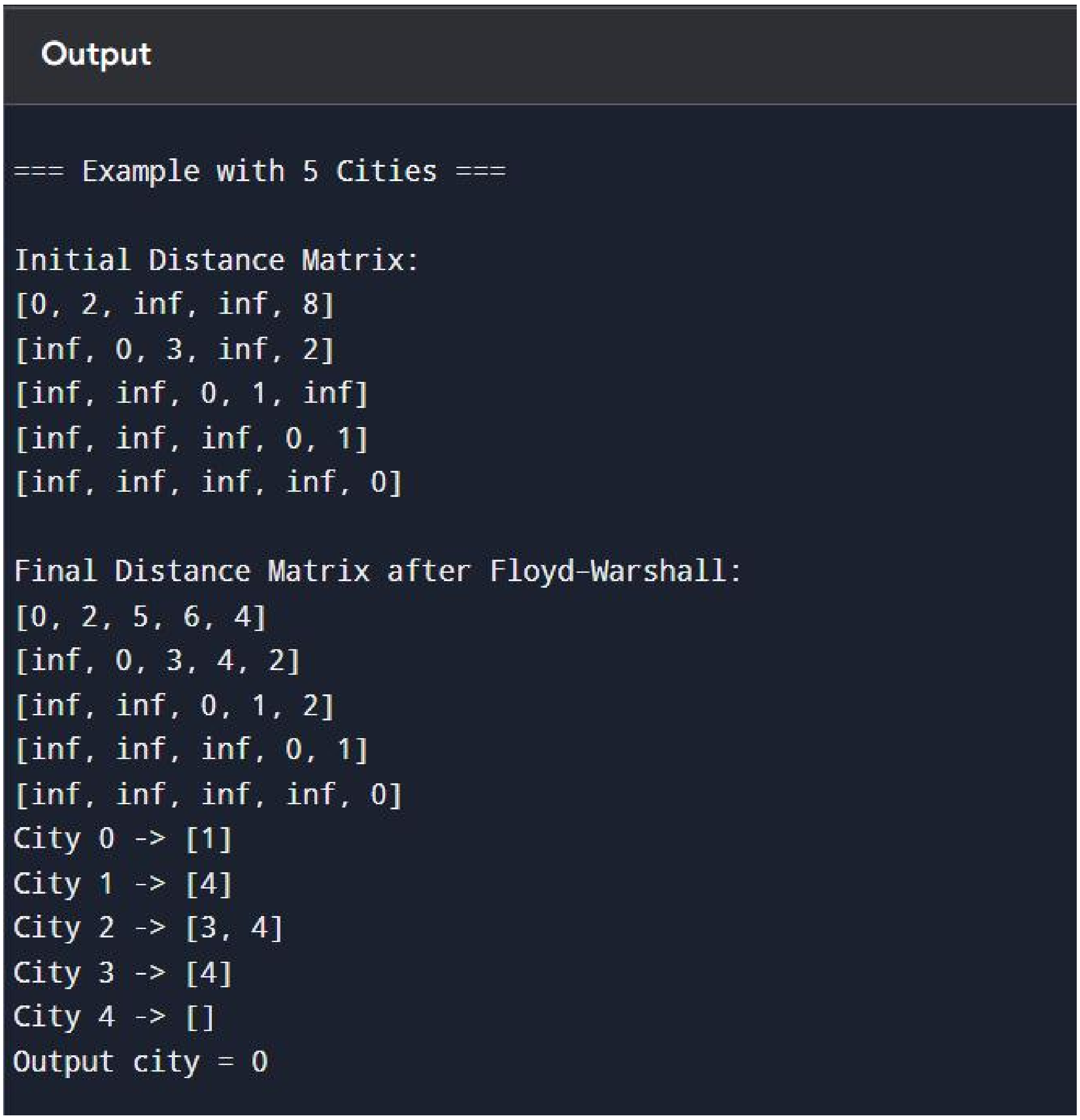
Output : E to C = 5



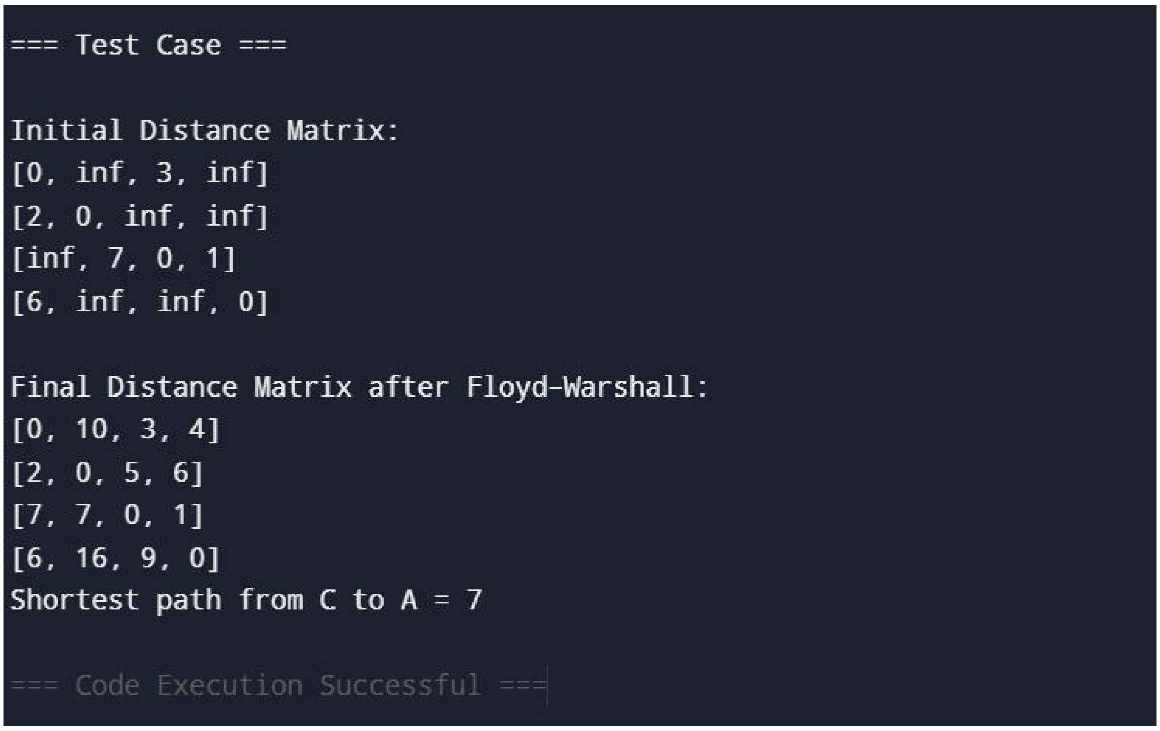
```
import math
def floyd_warshall(n, edges):
    dist = [[math.inf] * n for _ in range(n)]
    for i in range(n):
        dist[i][i] = 0
    for u, v, w in edges:
        dist[u][v] = w
    print("\nInitial Distance Matrix:")
    for row in dist:
        print(row)
    for k in range(n):
        for 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]
    print("\nFinal Distance Matrix after Floyd-Warshall:")
    for row in dist:
        print(row)
    return dist
def main():
```



```python
    print("\n=== Example with 5 Cities ===")
    n = 5
    edges = [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1],[3,4,1]]
    dist = floyd_warshall(n, edges)
    threshold = 2
    for city in range(n):
        neighbors = [j for j in range(n) if city != j and dist[city][j] <= threshold]
        print(f"City {city} -> {neighbors}")
    print("Output city =", 0)
    print("\n=== Test Case ===")
    n = 4
    edges = [
        [1,0,2],
        [0,2,3],
        [2,3,1],
        [3,0,6],
        [2,1,7]
    ]
    dist = floyd_warshall(n, edges)
    print("Shortest path from C to A =", dist[2][0])
if __name__ == "__main__":
    main()
```


Output

```
=== Example with 5 Cities ===

Initial Distance Matrix:
[0, 2, inf, inf, 8]
[inf, 0, 3, inf, 2]
[inf, inf, 0, 1, inf]
[inf, inf, inf, 0, 1]
[inf, inf, inf, inf, 0]

Final Distance Matrix after Floyd-Warshall:
[0, 2, 5, 6, 4]
[inf, 0, 3, 4, 2]
[inf, inf, 0, 1, 2]
[inf, inf, inf, 0, 1]
[inf, inf, inf, inf, 0]
City 0 -> [1]
City 1 -> [4]
City 2 -> [3, 4]
City 3 -> [4]
City 4 -> []
Output city = 0
```



```
=== Test Case ===

Initial Distance Matrix:
[0, inf, 3, inf]
[2, 0, inf, inf]
[inf, 7, 0, 1]
[6, inf, inf, 0]

Final Distance Matrix after Floyd-Warshall:
[0, 10, 3, 4]
[2, 0, 5, 6]
[7, 7, 0, 1]
[6, 16, 9, 0]
Shortest path from C to A = 7

=== Code Execution Successful ===
```

15. Implement the Optimal Binary Search Tree algorithm for the keys A,B,C,D with frequencies 0.1,0.2,0.4,0.3 Write the code using any programming language to construct the OBST for the given keys and frequencies. Execute your code and display the resulting OBST and its cost. Print the cost and root matrix.

Input N =4, Keys = {A,B,C,D} Frequencies = {01.02.,0.3,0.4}

Output : 1.7

Cost Table

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 1 | 0 | 0.1 | 0.4 | 1.1 | 1.7 |
| 2 | | 0 | 0.2 | 0.8 | 0.4 |
| 3 | | | 0 | 0.4 | 1.0 |
| 4 | | | | 0 | 0.3 |
| 5 | | | | | 0 |

Root table

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 1 | 2 | 3 | 3 |
| 2 | | 2 | 3 | 3 |
| 3 | | | 3 | 3 |

4 4

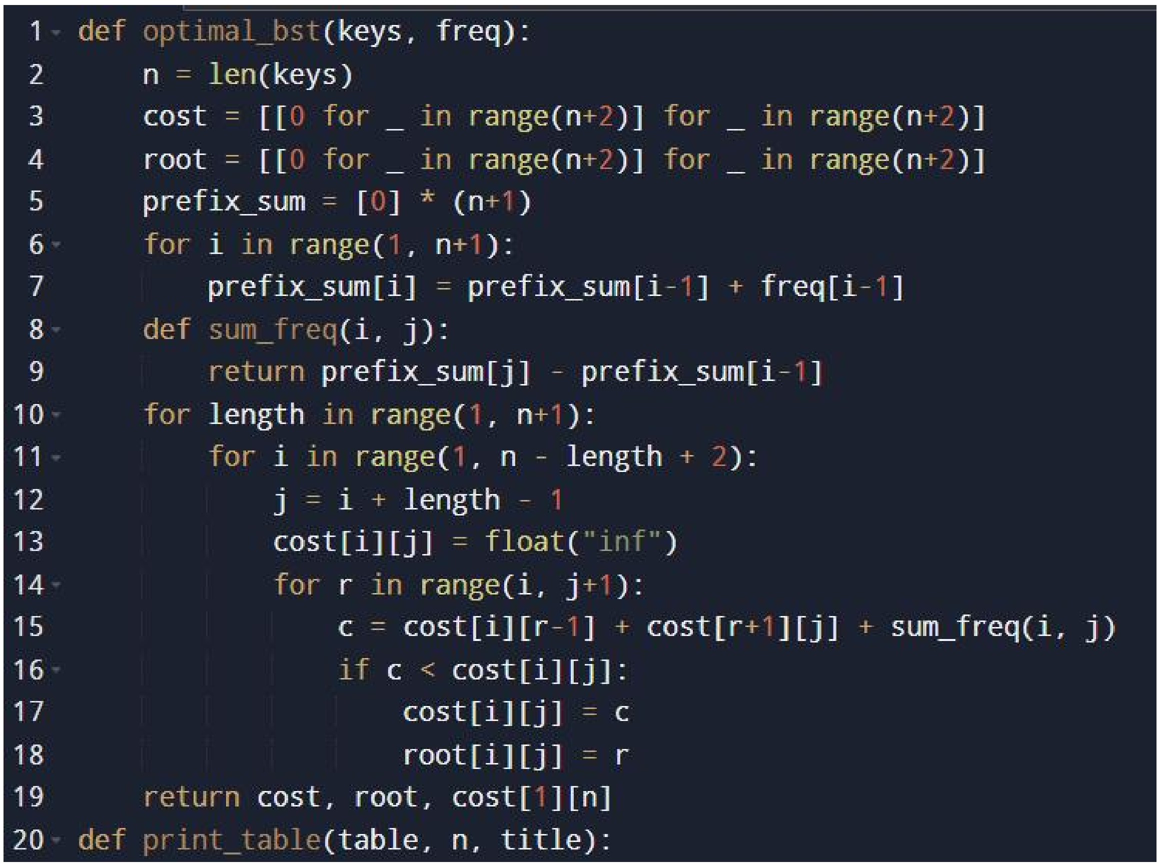
a) Test cases

Input: keys[] = {10, 12}, freq[] = {34, 50}

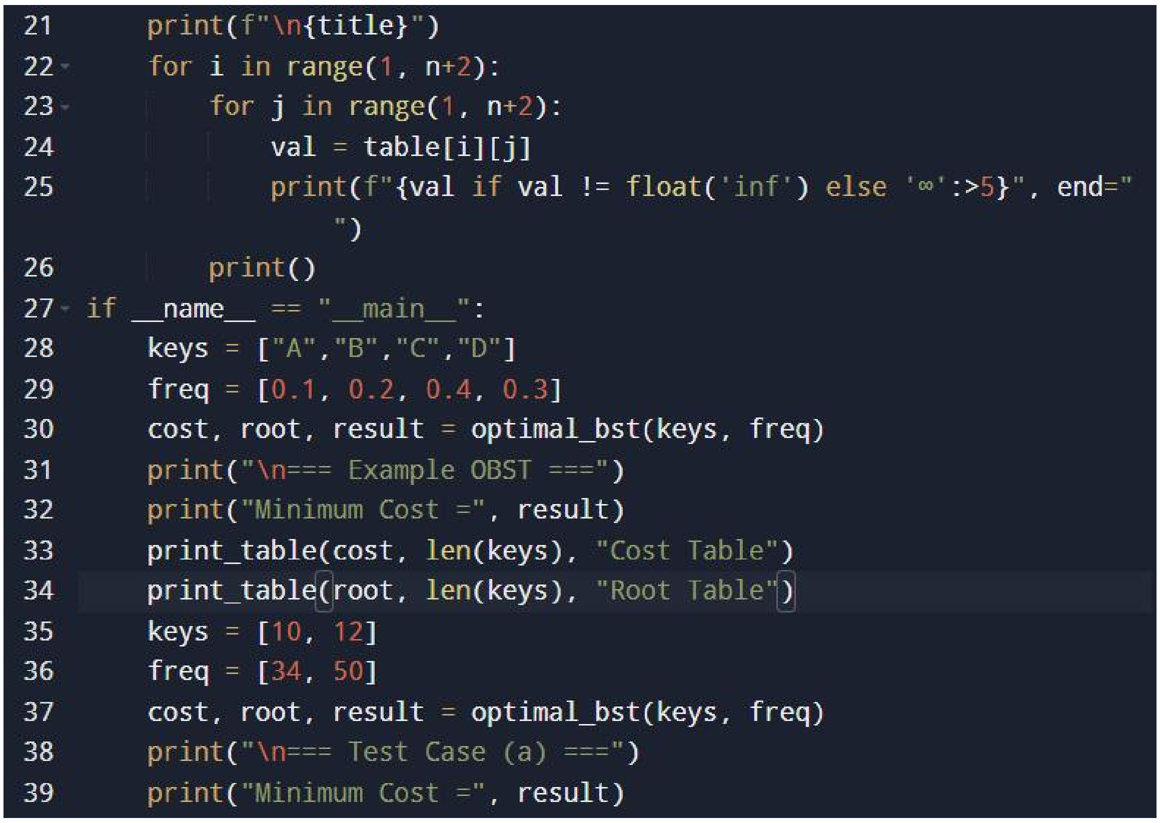
Output = 118

b) Input: keys[] = {10, 12, 20}, freq[] = {34, 8, 50}

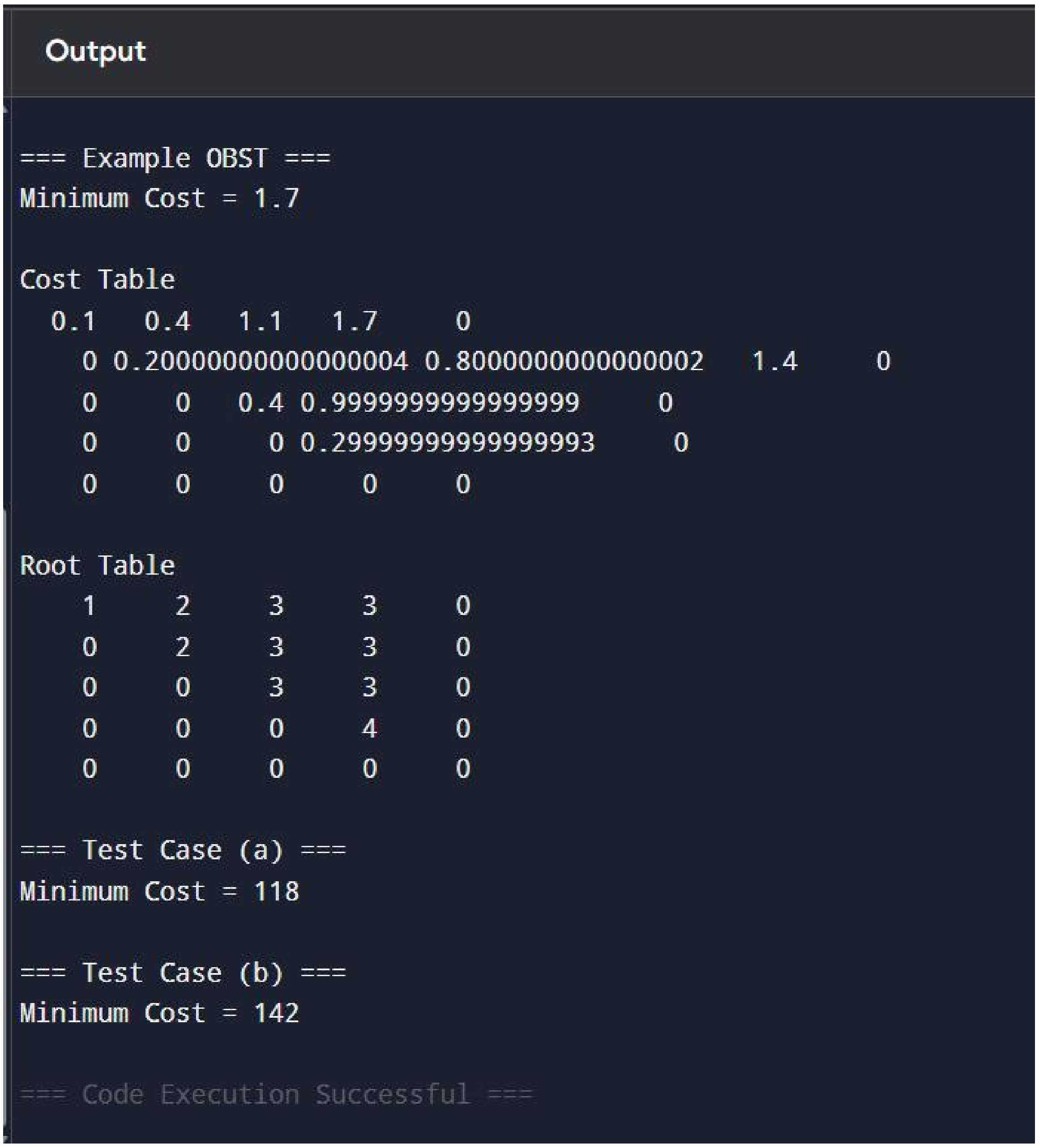
Output = 142



```python
def optimal_bst(keys, freq):
    n = len(keys)
    cost = [[0 for _ in range(n+2)] for _ in range(n+2)]
    root = [[0 for _ in range(n+2)] for _ in range(n+2)]
    prefix_sum = [0] * (n+1)
    for i in range(1, n+1):
        prefix_sum[i] = prefix_sum[i-1] + freq[i-1]
    def sum_freq(i, j):
        return prefix_sum[j] - prefix_sum[i-1]
    for length in range(1, n+1):
        for i in range(1, n - length + 2):
            j = i + length - 1
            cost[i][j] = float("inf")
            for r in range(i, j+1):
                c = cost[i][r-1] + cost[r+1][j] + sum_freq(i, j)
                if c < cost[i][j]:
                    cost[i][j] = c
                    root[i][j] = r
    return cost, root, cost[1][n]
def print_table(table, n, title):
```



```python
    print(f"\n{title}")
    for i in range(1, n+2):
        for j in range(1, n+2):
            val = table[i][j]
            print(f"{val if val != float('inf') else '∞':>5}", end=" ")
        print()
if __name__ == "__main__":
    keys = ["A","B","C","D"]
    freq = [0.1, 0.2, 0.4, 0.3]
    cost, root, result = optimal_bst(keys, freq)
    print("\n=== Example OBST ===")
    print("Minimum Cost =", result)
    print_table(cost, len(keys), "Cost Table")
    print_table(root, len(keys), "Root Table")
    keys = [10, 12]
    freq = [34, 50]
    cost, root, result = optimal_bst(keys, freq)
    print("\n=== Test Case (a) ===")
    print("Minimum Cost =", result)
```



```
Output

=== Example OBST ===
Minimum Cost = 1.7

Cost Table
  0.1   0.4   1.1   1.7     0
    0 0.20000000000000004 0.8000000000000002   1.4     0
    0     0   0.4 0.9999999999999999     0
    0     0     0 0.29999999999999993     0
    0     0     0     0     0

Root Table
    1     2     3     3     0
    0     2     3     3     0
    0     0     3     3     0
    0     0     0     4     0
    0     0     0     0     0

=== Test Case (a) ===
Minimum Cost = 118

=== Test Case (b) ===
Minimum Cost = 142

=== Code Execution Successful ===
```

16. Consider a set of keys 10,12,16,21 with frequencies 4,2,6,3 and the respective probabilities. Write a Program to construct an OBST in a programming language of your choice. Execute your code and display the resulting OBST, its cost and root matrix.

Input N =4, Keys = {10,12,16,21} Frequencies = {4,2,6,3}

Output : 26

|  | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 4 | 80 | 202 | 262 |
| 1 |  | 2 | 102 | 162 |
| 2 |  |  | 6 | 12 |

3 3

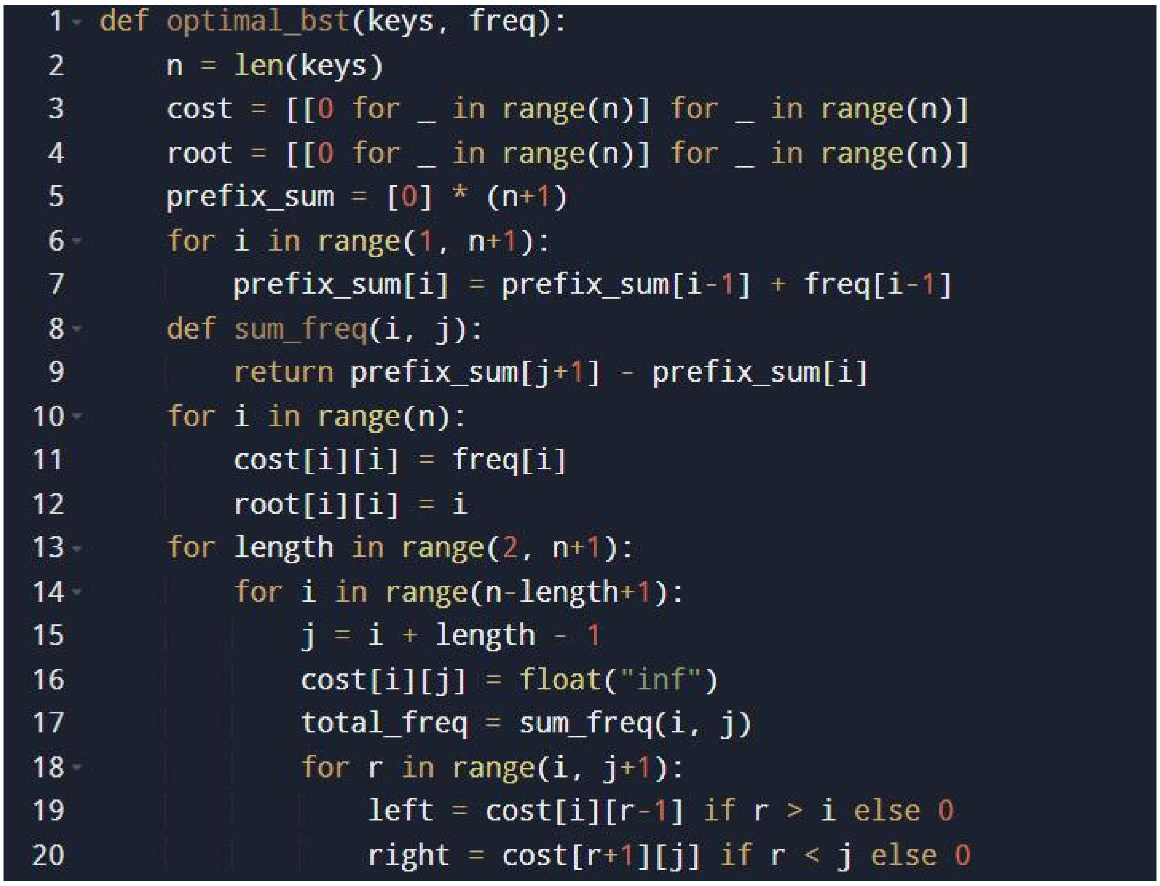
a) Test cases

Input:  keys[] = {10, 12}, freq[] = {34, 50}

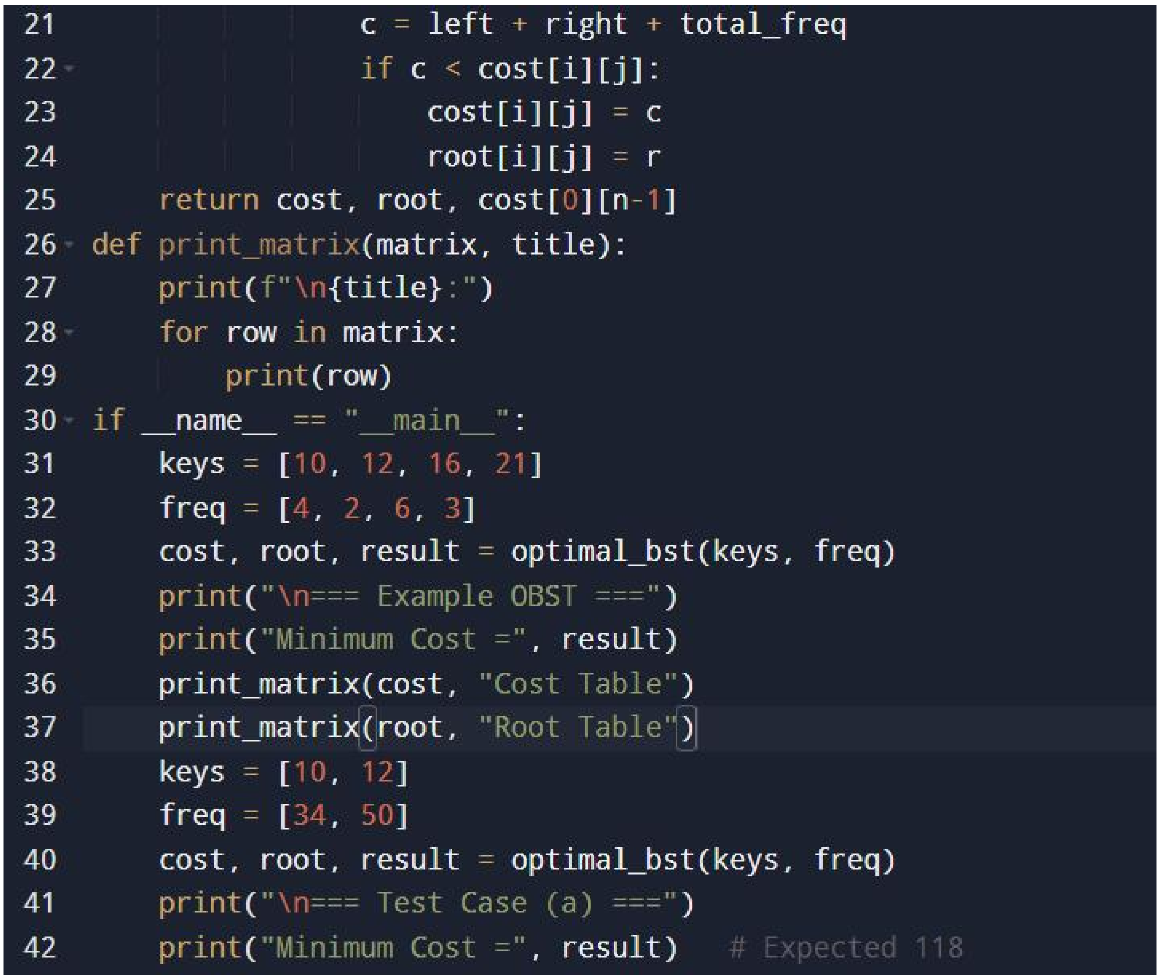
Output = 118

b) Input:  keys[] = {10, 12, 20}, freq[] = {34, 8, 50}

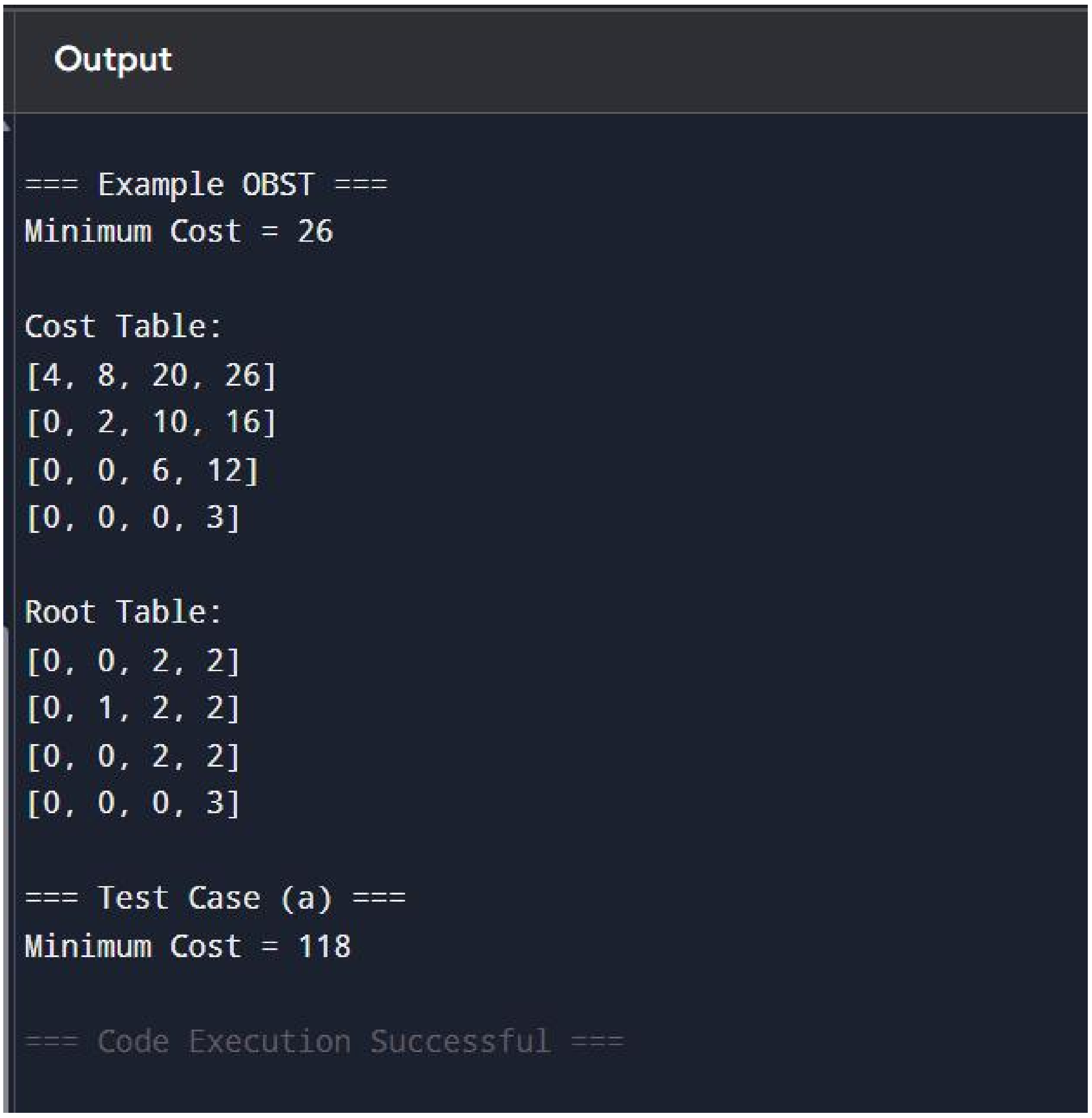
Output = 142



```python
def optimal_bst(keys, freq):
    n = len(keys)
    cost = [[0 for _ in range(n)] for _ in range(n)]
    root = [[0 for _ in range(n)] for _ in range(n)]
    prefix_sum = [0] * (n+1)
    for i in range(1, n+1):
        prefix_sum[i] = prefix_sum[i-1] + freq[i-1]
    def sum_freq(i, j):
        return prefix_sum[j+1] - prefix_sum[i]
    for i in range(n):
        cost[i][i] = freq[i]
        root[i][i] = i
    for length in range(2, n+1):
        for i in range(n-length+1):
            j = i + length - 1
            cost[i][j] = float("inf")
            total_freq = sum_freq(i, j)
            for r in range(i, j+1):
                left = cost[i][r-1] if r > i else 0
                right = cost[r+1][j] if r < j else 0
```



```python
                c = left + right + total_freq
                if c < cost[i][j]:
                    cost[i][j] = c
                    root[i][j] = r
    return cost, root, cost[0][n-1]
def print_matrix(matrix, title):
    print(f"\n{title}:")
    for row in matrix:
        print(row)
if __name__ == "__main__":
    keys = [10, 12, 16, 21]
    freq = [4, 2, 6, 3]
    cost, root, result = optimal_bst(keys, freq)
    print("\n=== Example OBST ===")
    print("Minimum Cost =", result)
    print_matrix(cost, "Cost Table")
    print_matrix(root, "Root Table")
    keys = [10, 12]
    freq = [34, 50]
    cost, root, result = optimal_bst(keys, freq)
    print("\n=== Test Case (a) ===")
    print("Minimum Cost =", result)   # Expected 118
```


Output

```
=== Example OBST ===
Minimum Cost = 26

Cost Table:
[4, 8, 20, 26]
[0, 2, 10, 16]
[0, 0, 6, 12]
[0, 0, 0, 3]

Root Table:
[0, 0, 2, 2]
[0, 1, 2, 2]
[0, 0, 2, 2]
[0, 0, 0, 3]

=== Test Case (a) ===
Minimum Cost = 118

=== Code Execution Successful ===
```

17. A game on an undirected graph is played by two players, Mouse and Cat, who alternate turns. The graph is given as follows: graph[a] is a list of all nodes b such that ab is an edge of the graph. The mouse starts at node 1 and goes first, the cat starts at node 2 and goes second, and there is a hole at node 0. During each player's turn, they must travel along one edge of the graph that meets where they are. For example, if the Mouse is at node 1, it must travel to any node in graph[1]. Additionally, it is not allowed for the Cat to travel to the Hole (node 0).Then, the game can end in three ways:

    If ever the Cat occupies the same node as the Mouse, the Cat wins.

    If ever the Mouse reaches the Hole, the Mouse wins.

    If ever a position is repeated (i.e., the players are in the same position as a previous turn, and it is the same player's turn to move), the game is a draw.

Given a graph, and assuming both players play optimally, return

1 if the mouse wins the game,

2 if the cat wins the game, or

0 if the game is a draw.

Example 1:

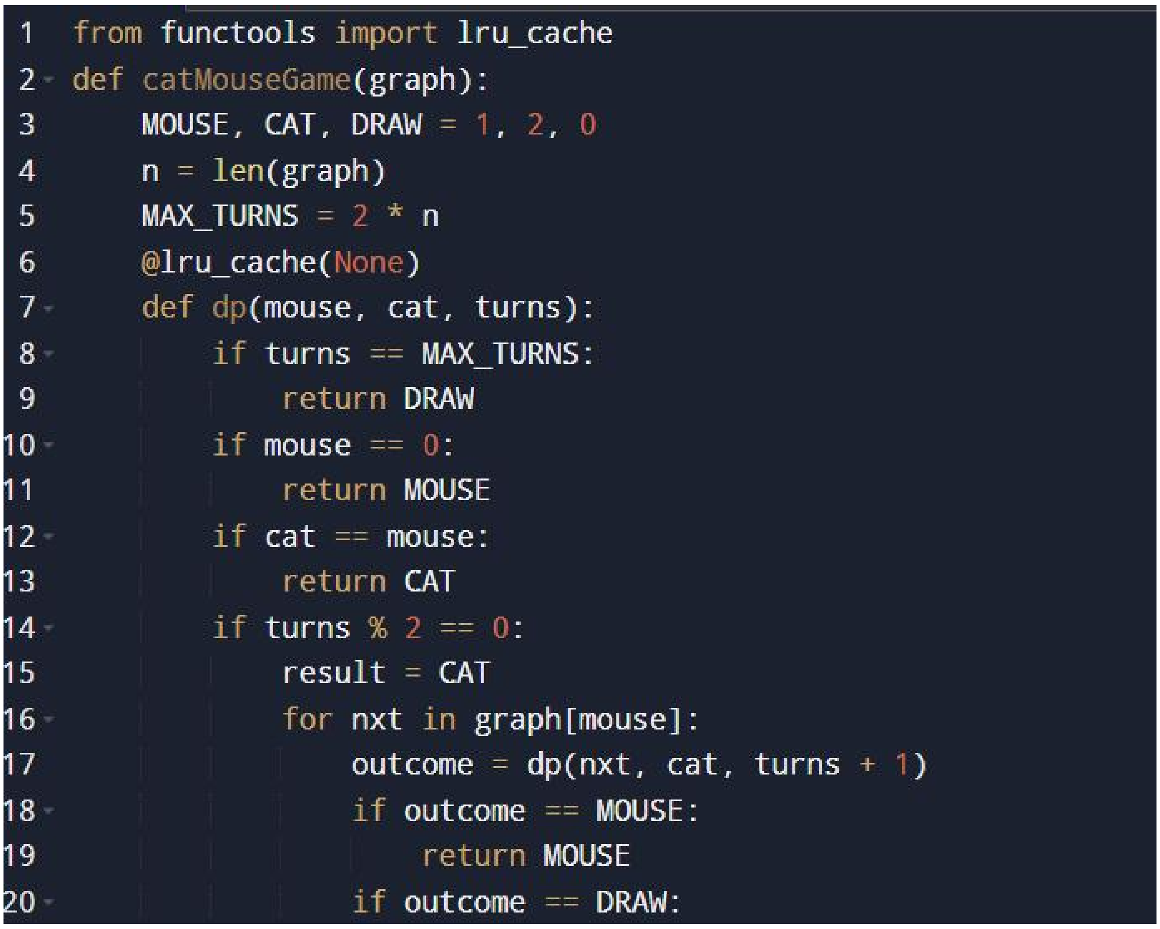
Input: graph = [[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]

Output: 0

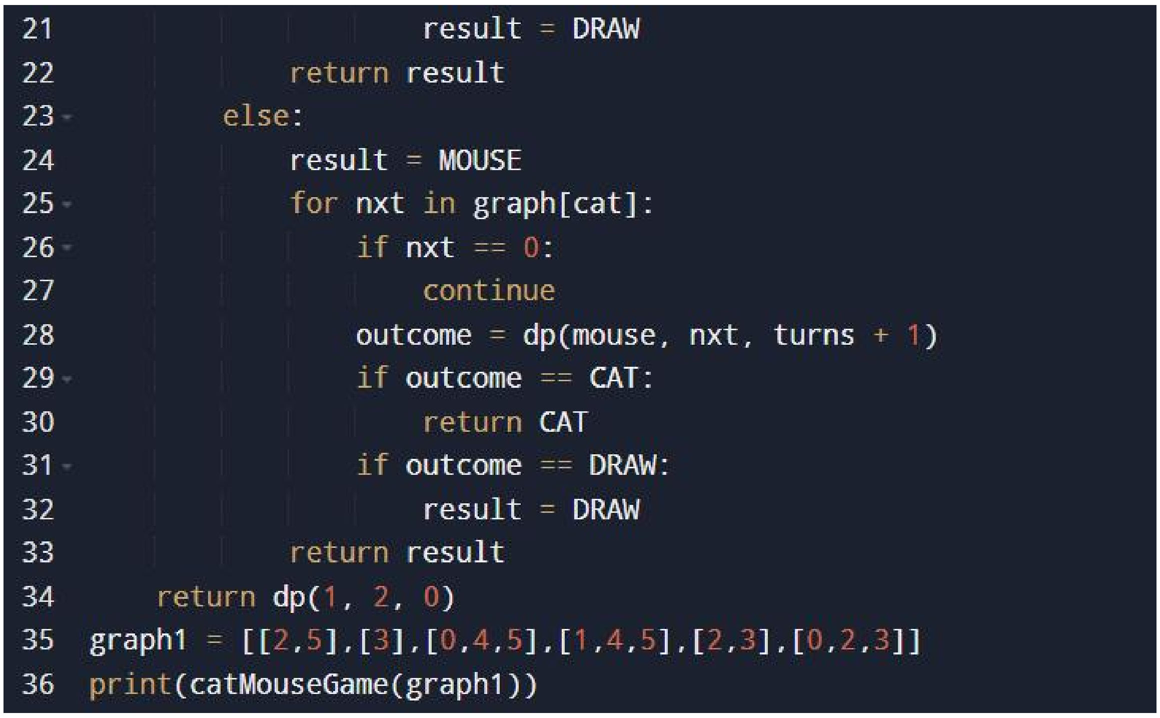
Example 2:

Input: graph = [[1,3],[0],[3],[0,2]]

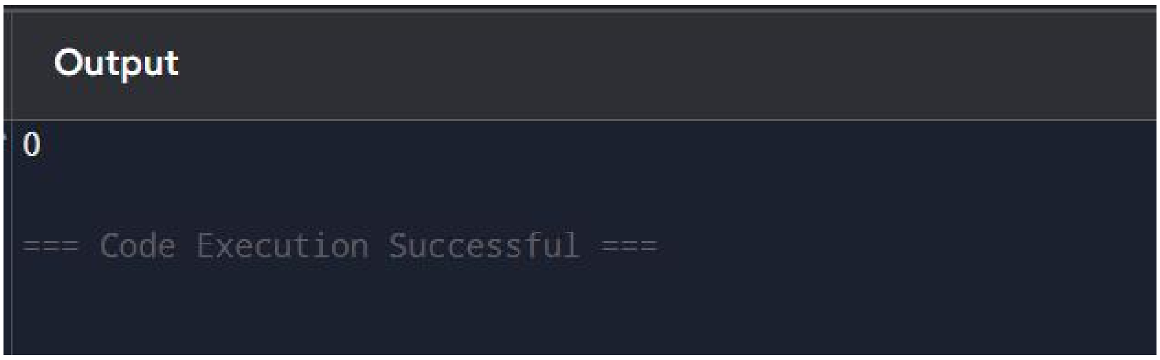
Output: 1



```python
from functools import lru_cache
def catMouseGame(graph):
    MOUSE, CAT, DRAW = 1, 2, 0
    n = len(graph)
    MAX_TURNS = 2 * n
    @lru_cache(None)
    def dp(mouse, cat, turns):
        if turns == MAX_TURNS:
            return DRAW
        if mouse == 0:
            return MOUSE
        if cat == mouse:
            return CAT
        if turns % 2 == 0:
            result = CAT
            for nxt in graph[mouse]:
                outcome = dp(nxt, cat, turns + 1)
                if outcome == MOUSE:
                    return MOUSE
                if outcome == DRAW:
```



```python
                    result = DRAW
            return result
        else:
            result = MOUSE
            for nxt in graph[cat]:
                if nxt == 0:
                    continue
                outcome = dp(mouse, nxt, turns + 1)
                if outcome == CAT:
                    return CAT
                if outcome == DRAW:
                    result = DRAW
            return result
    return dp(1, 2, 0)
graph1 = [[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]
print(catMouseGame(graph1))
```



Output

```
0

=== Code Execution Successful ===
```

18. You are given an undirected weighted graph of n nodes (0-indexed), represented by an edge list where edges[i] = [a, b] is an undirected edge connecting the nodes a and b with a probability of success of traversing that edge succProb[i]. Given two nodes start and end, find the path with the maximum probability of success to go from start to end and return its success probability. If there is no path from start to end, return 0. Your answer will be accepted if it differs from the correct answer by at most 1e-5.

    Example 1:

    Input: n = 3, edges = [[0,1],[1,2],[0,2]], succProb = [0.5,0.5,0.2], start = 0, end = 2

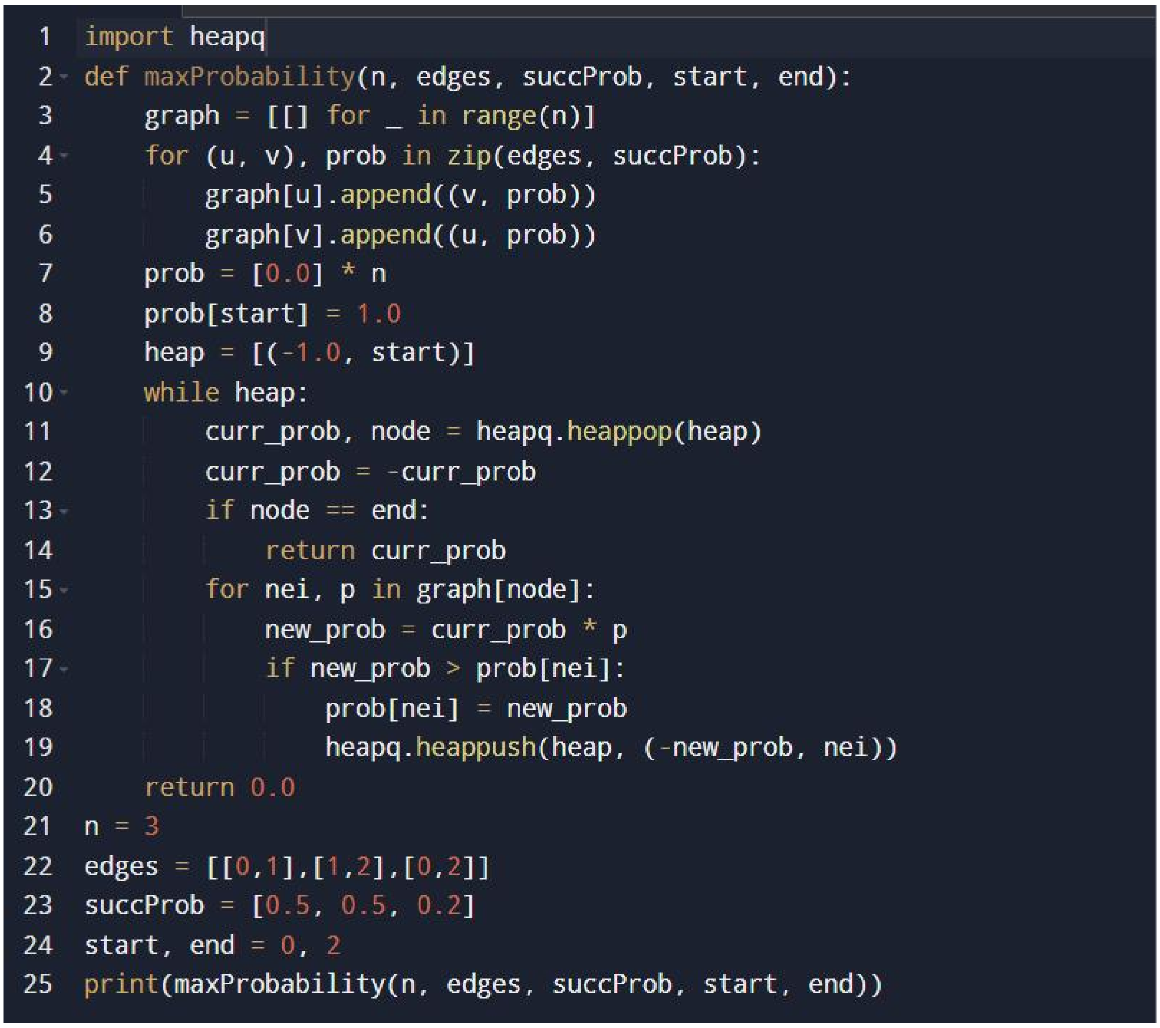
    Output: 0.25000

    Explanation: There are two paths from start to end, one having a probability of success = 0.2 and the other has 0.5 * 0.5 = 0.25.

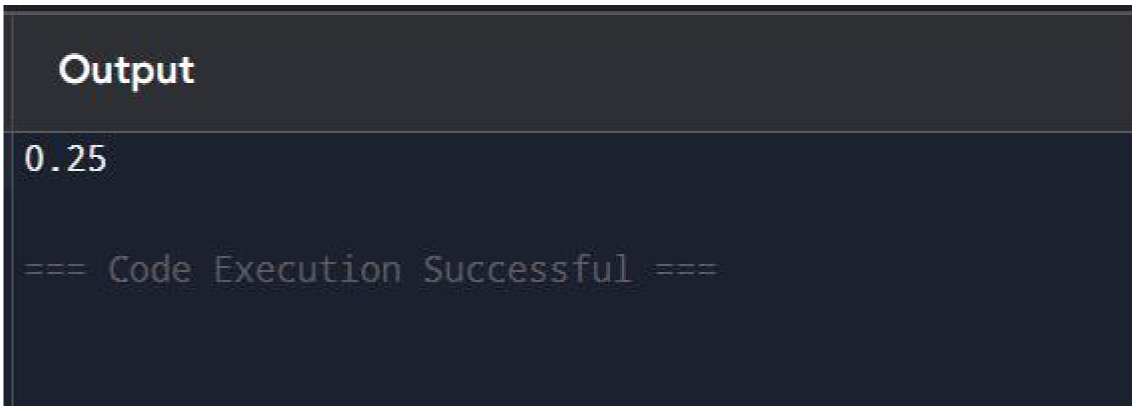
    Example 2:

    Input: n = 3, edges = [[0,1],[1,2],[0,2]], succProb = [0.5,0.5,0.3], start = 0, end = 2

Output: 0.30000



```python
import heapq
def maxProbability(n, edges, succProb, start, end):
    graph = [[] for _ in range(n)]
    for (u, v), prob in zip(edges, succProb):
        graph[u].append((v, prob))
        graph[v].append((u, prob))
    prob = [0.0] * n
    prob[start] = 1.0
    heap = [(-1.0, start)]
    while heap:
        curr_prob, node = heapq.heappop(heap)
        curr_prob = -curr_prob
        if node == end:
            return curr_prob
        for nei, p in graph[node]:
            new_prob = curr_prob * p
            if new_prob > prob[nei]:
                prob[nei] = new_prob
                heapq.heappush(heap, (-new_prob, nei))
    return 0.0
n = 3
edges = [[0,1],[1,2],[0,2]]
succProb = [0.5, 0.5, 0.2]
start, end = 0, 2
print(maxProbability(n, edges, succProb, start, end))
```



Output

```
0.25

=== Code Execution Successful ===
```

19. There is a robot on an m x n grid. The robot is initially located at the top-left corner (i.e., grid[0][0]). The robot tries to move to the bottom-right corner (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time. Given the two integers m and n, return the number of possible unique paths that the

robot can take to reach the bottom-right corner. The test cases are generated so that the answer will be less than or equal to 2 * 10 9.

Example 1:

START

FINISH

Input: m = 3, n = 7

Output: 28

Example 2:

Input: m = 3, n = 2

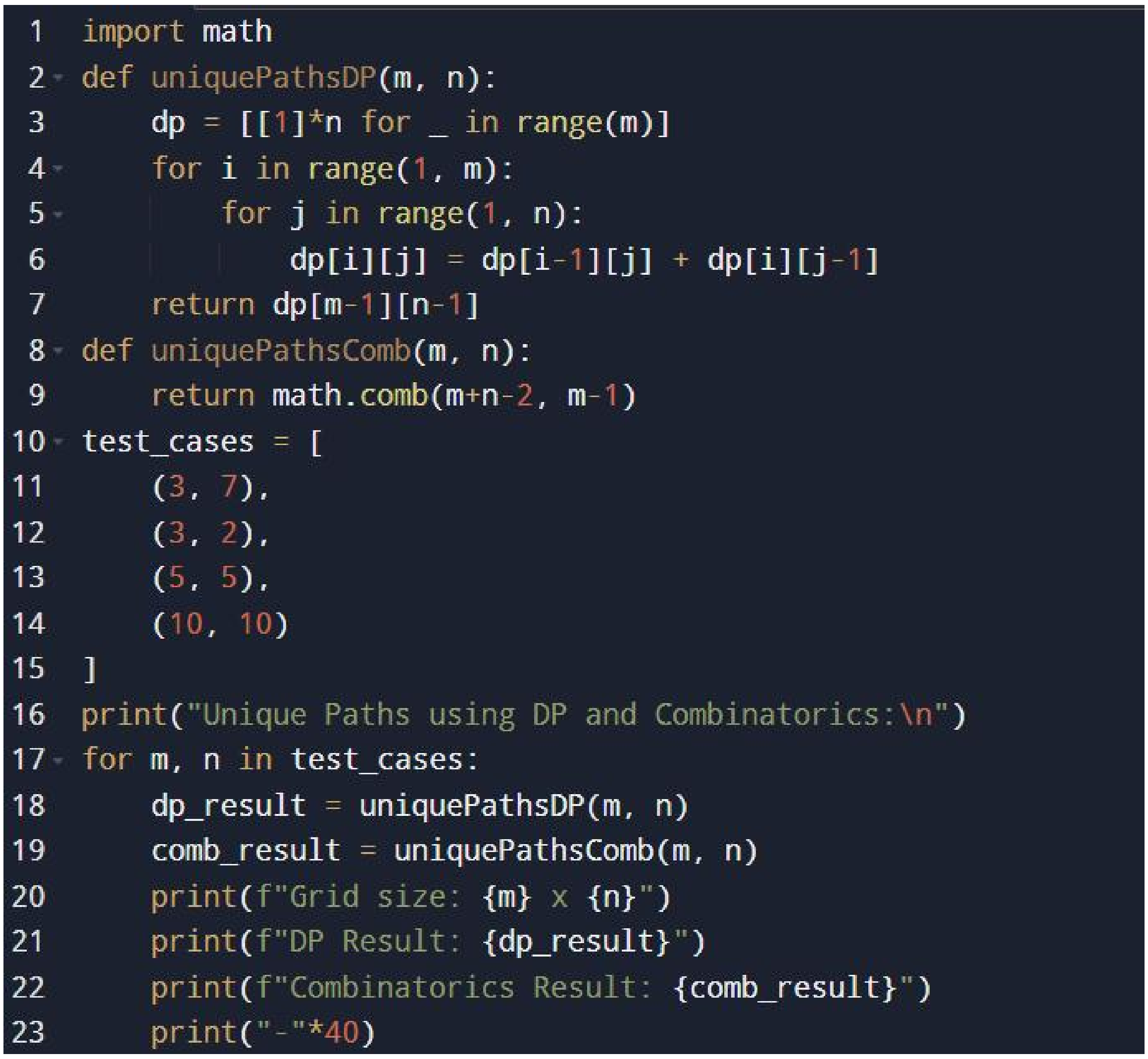
Output: 3

Explanation: From the top-left corner, there are a total of 3 ways to reach the bottom-right corner:

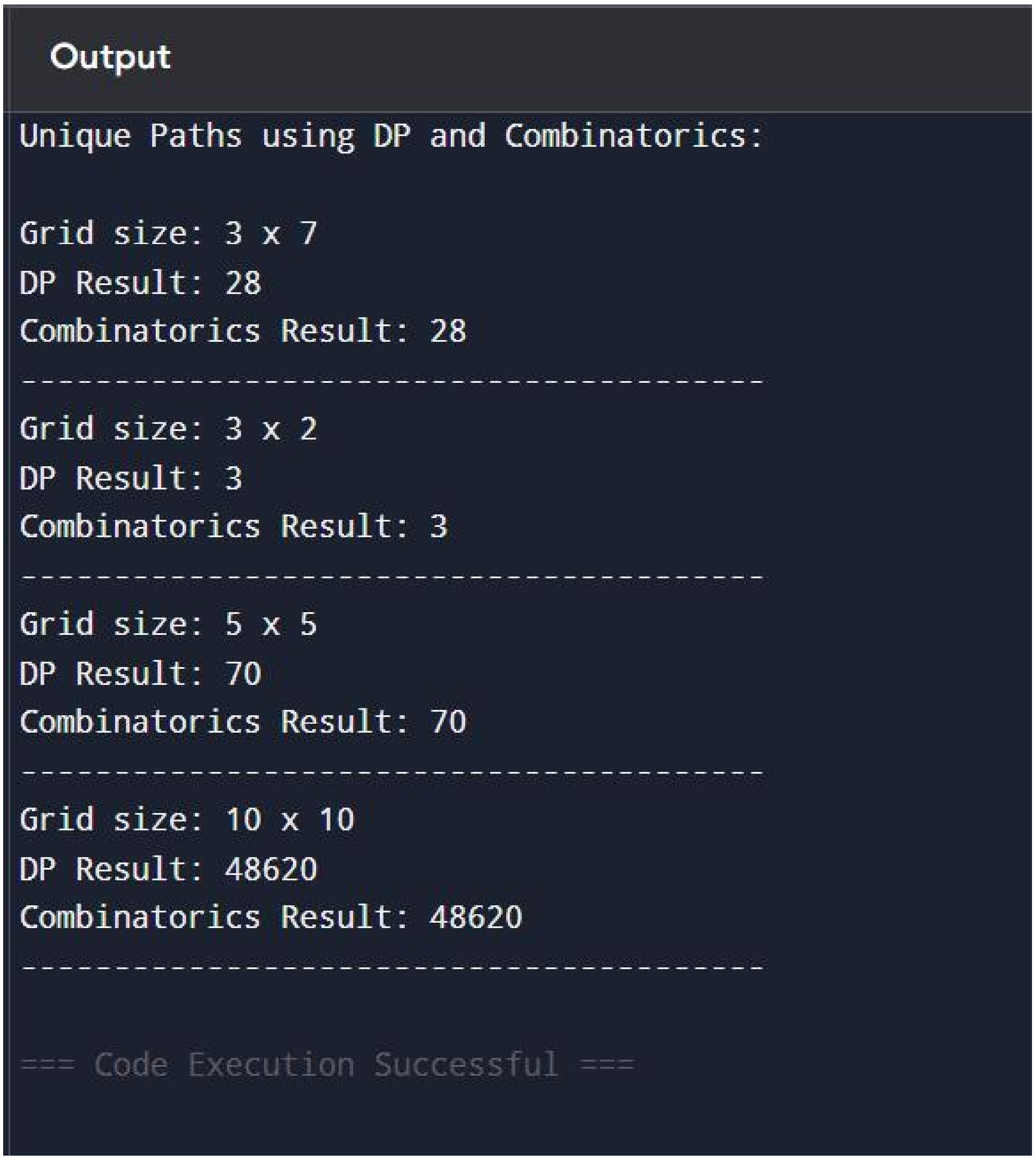
1. Right -> Down -> Down

2. Down -> Down -> Right

3. Down -> Right -> Down



```python
import math
def uniquePathsDP(m, n):
    dp = [[1]*n for _ in range(m)]
    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = dp[i-1][j] + dp[i][j-1]
    return dp[m-1][n-1]
def uniquePathsComb(m, n):
    return math.comb(m+n-2, m-1)
test_cases = [
    (3, 7),
    (3, 2),
    (5, 5),
    (10, 10)
]
print("Unique Paths using DP and Combinatorics:\n")
for m, n in test_cases:
    dp_result = uniquePathsDP(m, n)
    comb_result = uniquePathsComb(m, n)
    print(f"Grid size: {m} x {n}")
    print(f"DP Result: {dp_result}")
    print(f"Combinatorics Result: {comb_result}")
    print("-"*40)
```



```
Unique Paths using DP and Combinatorics:

Grid size: 3 x 7
DP Result: 28
Combinatorics Result: 28
----------------------------------------
Grid size: 3 x 2
DP Result: 3
Combinatorics Result: 3
----------------------------------------
Grid size: 5 x 5
DP Result: 70
Combinatorics Result: 70
----------------------------------------
Grid size: 10 x 10
DP Result: 48620
Combinatorics Result: 48620
----------------------------------------


=== Code Execution Successful ===
```

20. Given an array of integers nums, return the number of good pairs. A pair (i, j) is called good if nums[i] == nums[j] and i < j.

    Example 1:

    Input: nums = [1,2,3,1,1,3]

    Output: 4

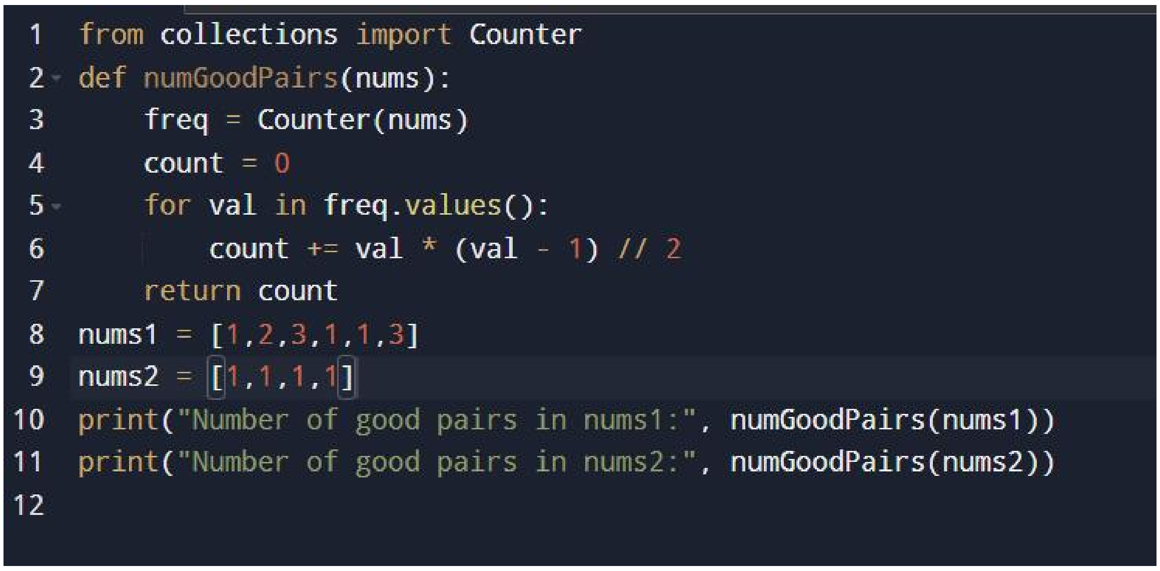
    Explanation: There are 4 good pairs (0,3), (0,4), (3,4), (2,5) 0-indexed.

    Example 2:

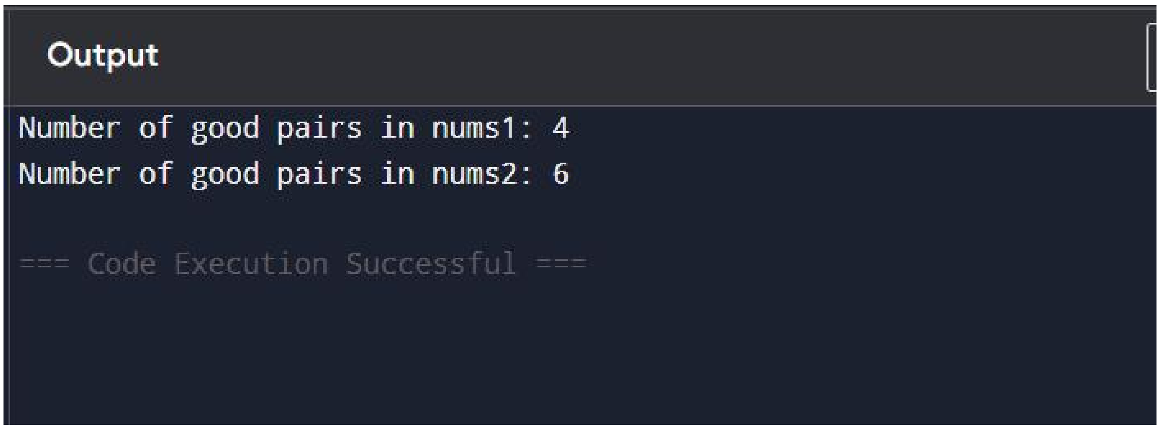
    Input: nums = [1,1,1,1]

Output: 6

Explanation: Each pair in the array are good.



```python
from collections import Counter
def numGoodPairs(nums):
    freq = Counter(nums)
    count = 0
    for val in freq.values():
        count += val * (val - 1) // 2
    return count
nums1 = [1,2,3,1,1,3]
nums2 = [1,1,1,1]
print("Number of good pairs in nums1:", numGoodPairs(nums1))
print("Number of good pairs in nums2:", numGoodPairs(nums2))
```



Output

```
Number of good pairs in nums1: 4
Number of good pairs in nums2: 6

=== Code Execution Successful ===
```

21. There are n cities numbered from 0 to n-1. Given the array edges where edges[i] = [fromi, toi, weighti] represents a bidirectional and weighted edge between cities fromi and toi, and given the integer distanceThreshold. Return the city with the smallest number of cities that are reachable through some path and whose distance is at most distanceThreshold, If there are multiple such cities, return the city with the greatest number. Notice that the distance of a path connecting cities i and j is equal to the sum of the edges' weights along that path.

    **Example 1:**

    Input: n = 4, edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]], distanceThreshold = 4

    Output: 3

    Explanation: The figure above describes the graph.

The neighboring cities at a distanceThreshold = 4 for each city are:

City 0 -> [City 1, City 2]

City 1 -> [City 0, City 2, City 3]

City 2 -> [City 0, City 1, City 3]

City 3 -> [City 1, City 2]

Cities 0 and 3 have 2 neighboring cities at a distance Threshold = 4, but we have to return city 3 since it has the greatest number.

**Example 2:**

Input: n = 5, edges = [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1],[3,4,1]], distance Threshold = 2

Output: 0

Explanation: The figure above describes the graph.

The neighboring cities at a distance Threshold = 2 for each city are:

City 0 -> [City 1]

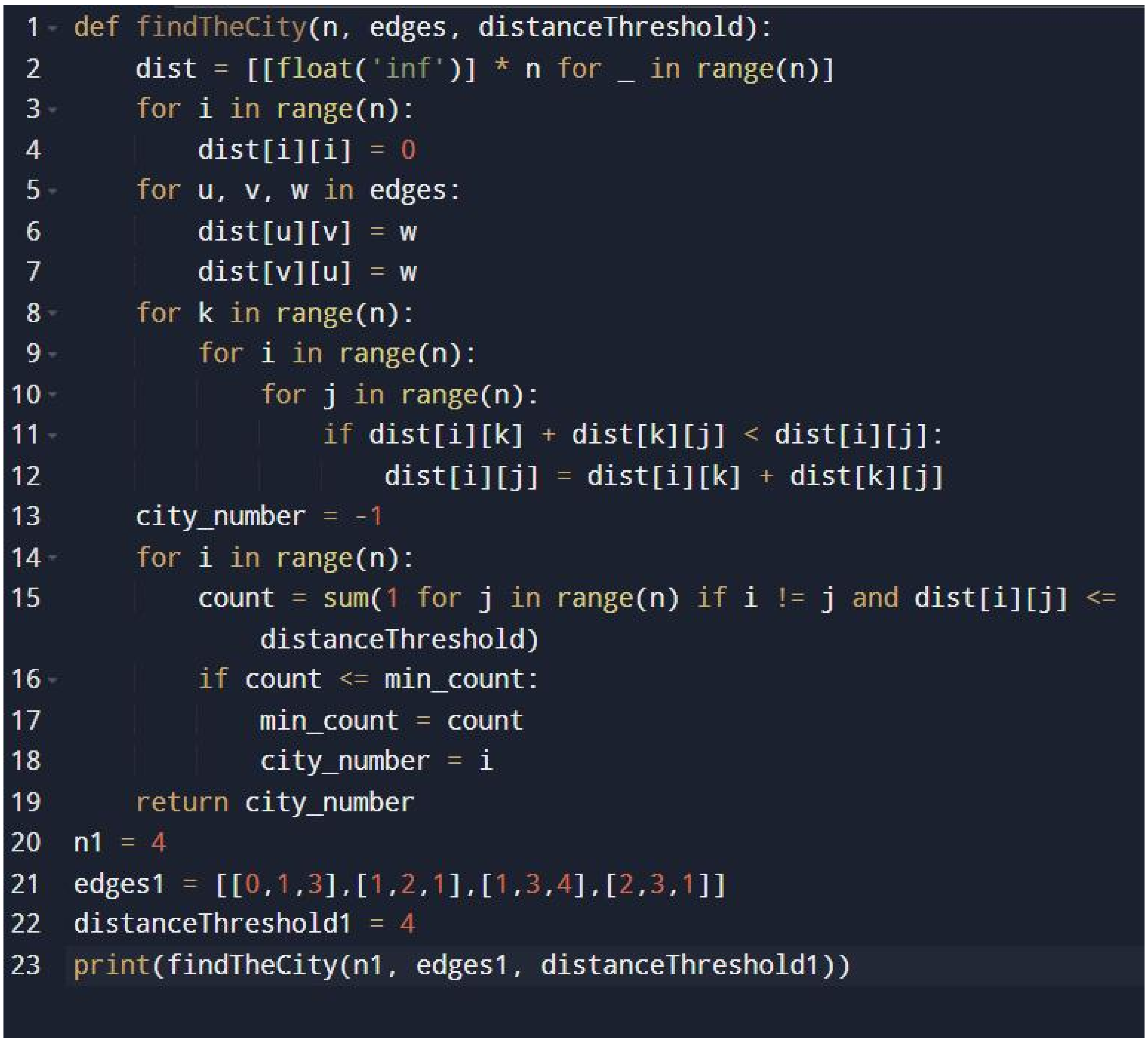
City 1 -> [City 0, City 4]

City 2 -> [City 3, City 4]

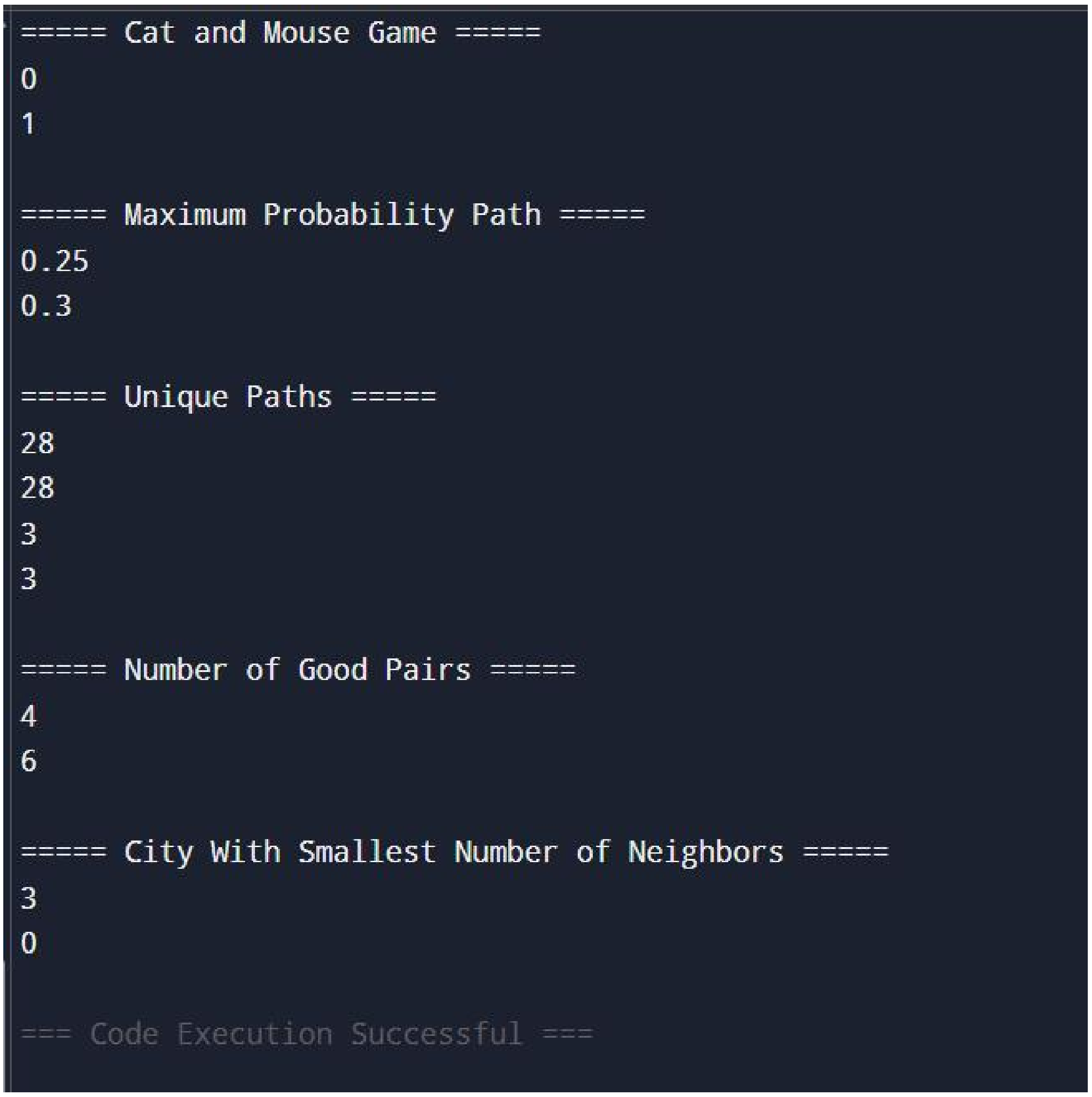
City 3 -> [City 2, City 4]

City 4 -> [City 1, City 2, City 3]

The city 0 has 1 neighboring city at a distanceThreshold = 2.



```python
def findTheCity(n, edges, distanceThreshold):
    dist = [[float('inf')] * n for _ in range(n)]
    for i in range(n):
        dist[i][i] = 0
    for u, v, w in edges:
        dist[u][v] = w
        dist[v][u] = w
    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]
    city_number = -1
    for i in range(n):
        count = sum(1 for j in range(n) if i != j and dist[i][j] <=
            distanceThreshold)
        if count <= min_count:
            min_count = count
            city_number = i
    return city_number
n1 = 4
edges1 = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]]
distanceThreshold1 = 4
print(findTheCity(n1, edges1, distanceThreshold1))
```



```
===== Cat and Mouse Game =====
0
1

===== Maximum Probability Path =====
0.25
0.3

===== Unique Paths =====
28
28
3
3

===== Number of Good Pairs =====
4
6

===== City With Smallest Number of Neighbors =====
3
0

=== Code Execution Successful ===
```

22. You are given a network of n nodes, labeled from 1 to n. You are also given times, a list of travel times as directed edges times[i] = (ui, vi, wi), where ui is the source node, vi is the target node, and wi is the time it takes for a signal to travel from source to target. We will send a signal from a given node k. Return the minimum time it takes for all the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1.

    Example 1:

    Input: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2

    Output: 2

    Example 2:

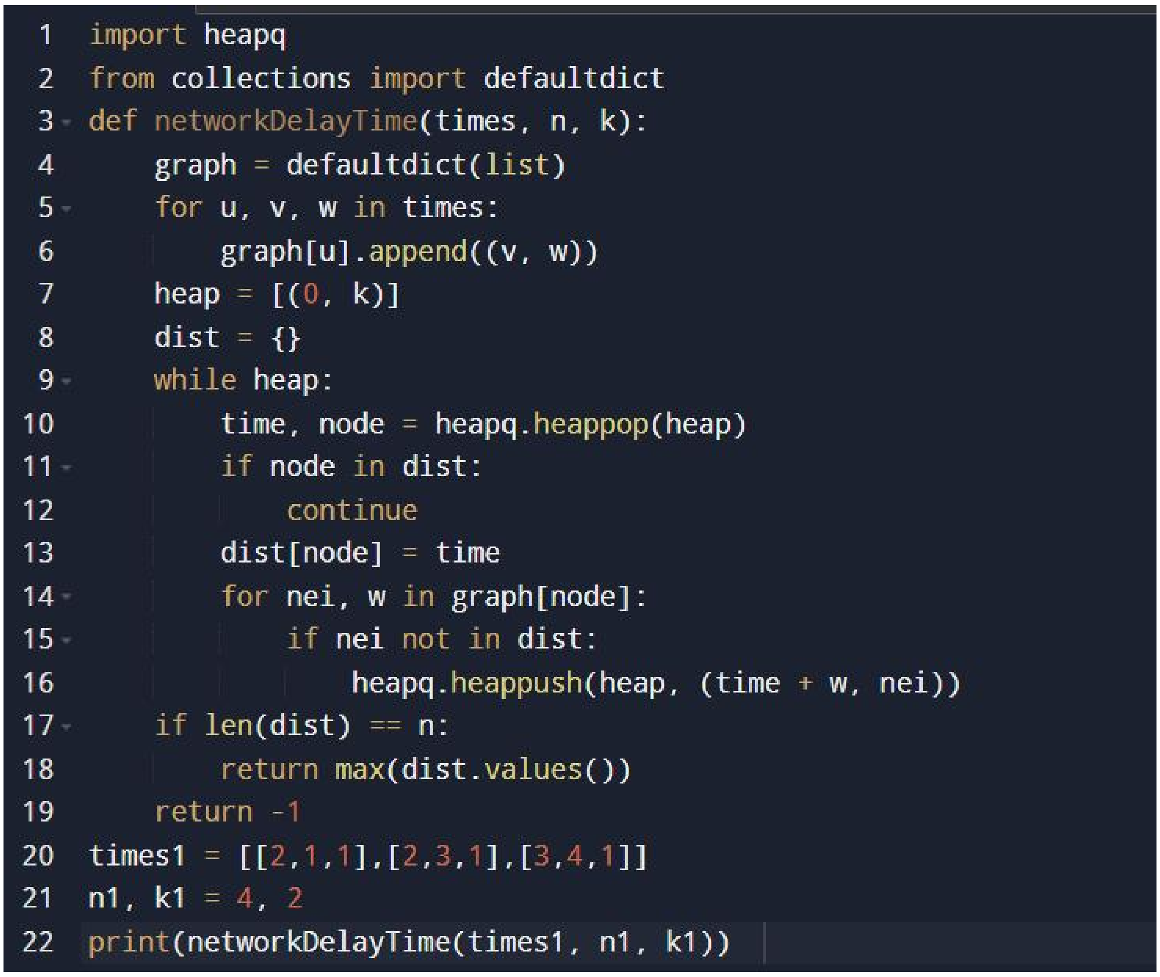
    Input: times = [[1,2,1]], n = 2, k = 1

Output: 1

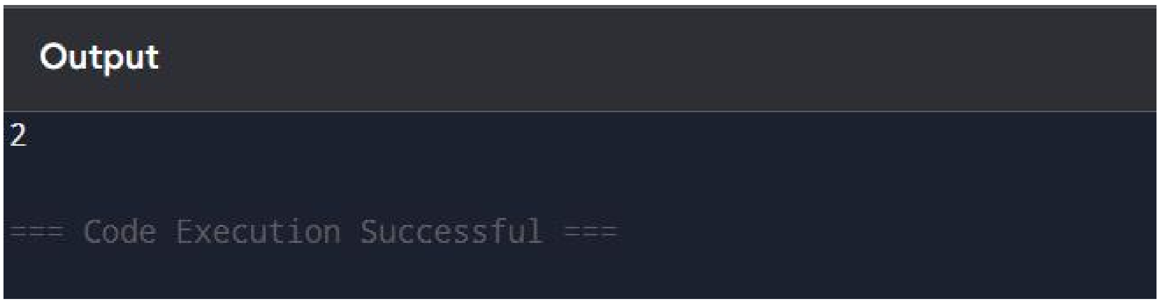
Example 3:

Input: times = [[1,2,1]], n = 2, k = 2

Output: -1



```python
import heapq
from collections import defaultdict
def networkDelayTime(times, n, k):
    graph = defaultdict(list)
    for u, v, w in times:
        graph[u].append((v, w))
    heap = [(0, k)]
    dist = {}
    while heap:
        time, node = heapq.heappop(heap)
        if node in dist:
            continue
        dist[node] = time
        for nei, w in graph[node]:
            if nei not in dist:
                heapq.heappush(heap, (time + w, nei))
    if len(dist) == n:
        return max(dist.values())
    return -1
times1 = [[2,1,1],[2,3,1],[3,4,1]]
n1, k1 = 4, 2
print(networkDelayTime(times1, n1, k1))
```



Output

```
2


=== Code Execution Successful ===
```