NAME: SHUBAS GM
REG NO: 192210058
COURSE CODE: CSA0614
COURSE NAME: Design and Analysis of Algorithms for Approximation Problems

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.

Aim:

To find the number of possible ways to obtain a given target sum by throwing a specified number of dice, each having a fixed number of sides, using dynamic programming.

Algorithm:

- Initialize a 2D table dp where dp[i][j] represents the number of ways to get sum j using i dice.
- Set the base condition: dp[0][0] = 1.
- For each die from 1 to num_dice:
  - For each possible sum from 1 to target:
    - Add the number of ways from previous dice considering all possible face values.
- The final answer is stored in dp[num_dice][target].

Program:

```
num_sides = 6
num_dice = 2
target = 7

dp = [[0] * (target + 1) for _ in range(num_dice + 1)]

dp[0][0] = 1

for dice in range(1, num_dice + 1):
    for curr_sum in range(1, target + 1):
        for face in range(1, num_sides + 1):
```
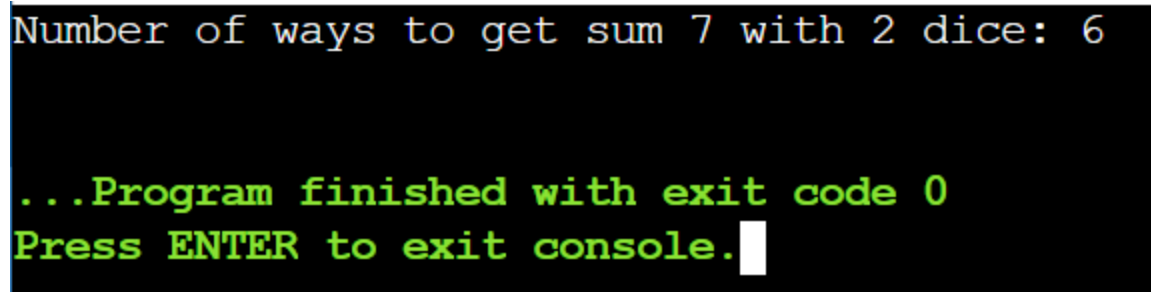
```
        if curr_sum - face >= 0:
            dp[dice][curr_sum] += dp[dice - 1][curr_sum - face]

print("Number of ways to get sum", target, "with",
    num_dice, "dice:", dp[num_dice][target])
```

Sample Input:
Simple Case:
•Number of sides: 6
•Number of dice: 2
•Target sum: 7

Output:


```
Number of ways to get sum 7 with 2 dice: 6


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
Number of ways = 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.

Aim:
To determine the minimum time required to process a product through two assembly lines with multiple stations, considering station processing times, transfer times between lines, and entry and exit times, using dynamic programming.

Algorithm:

- Initialize two arrays f1 and f2 to store the minimum time to reach each station on assembly line 1 and line 2.
- Add entry times to the first station processing times.
- For each station from 2 to n, compute the minimum time by choosing either to stay on the same line or switch from the other line including transfer time.
- Add exit times to the final station times.
- The minimum of the two final values is the required processing time.

Program:

n = 4

a1 = [4, 5, 3, 2]
a2 = [2, 10, 1, 4]

t1 = [0, 7, 4, 5]
t2 = [0, 9, 2, 8]

e1 = 10
e2 = 12

x1 = 18
x2 = 7

f1 = [0] * n
f2 = [0] * n

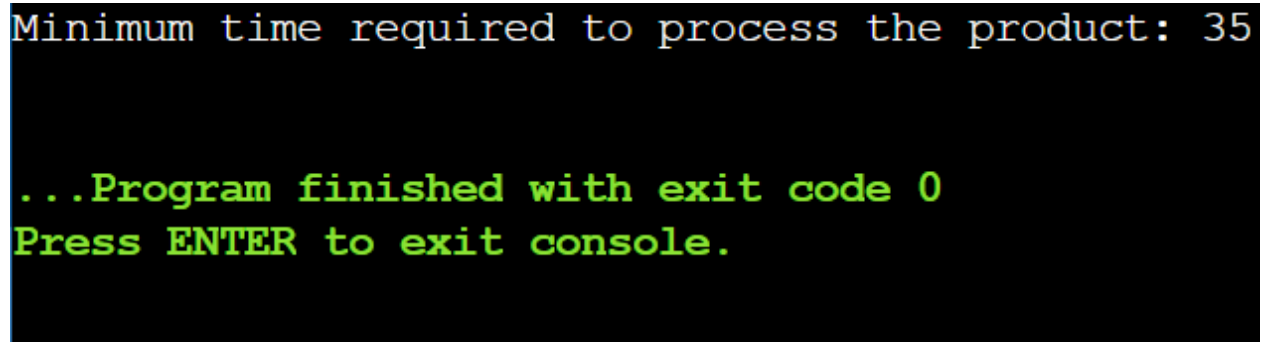f1[0] = e1 + a1[0]
f2[0] = e2 + a2[0]

for i in range(1, n):
    f1[i] = min(
        f1[i - 1] + a1[i],
        f2[i - 1] + t2[i] + a1[i]
)
    f2[i] = min(
        f2[i - 1] + a2[i],
        f1[i - 1] + t1[i] + a2[i]
)

result = min(f1[n - 1] + x1, f2[n - 1] + x2)
print("Minimum time required to process the product:", result)

Sample 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:

```
Minimum time required to process the product: 35



...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
Minimum time = 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.

Aim:
To minimize total production time across three assembly lines considering transfer times and dependencies.

Algorithm:

- Initialize a DP table where each entry stores the minimum time to complete a station on a particular assembly line.
- Set the first station time directly from the station processing times of each line.
- For each subsequent station, compute the minimum time by selecting the least of all possible previous lines plus transfer time and current station time.
- Continue this process while respecting the station order dependencies.
- The minimum value at the final station gives the optimal total production time.

Program:

```
n = 3

line_times = [
    [5, 9, 3],
    [6, 8, 4],
    [7, 6, 5]
]

transfer = [
    [0, 2, 3],
    [2, 0, 4],
    [3, 4, 0]
]

dp = [[0] * 3 for _ in range(n)]

for line in range(3):
    dp[0][line] = line_times[line][0]

for station in range(1, n):
    for curr_line in range(3):
        dp[station][curr_line] = min(
            dp[station - 1][prev_line] +
            transfer[prev_line][curr_line] +
            line_times[curr_line][station]
            for prev_line in range(3)
        )
result = min(dp[n - 1])
print("Minimum total production time:", result)
```

Sample Input:
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]
]

Output:

```
Minimum total production time: 17



...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
Minimum time = 17

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

Aim:
To find the minimum path distance to visit all cities exactly once and return to the start using a distance matrix.

Algorithm:

- ● Represent cities and distances in a matrix.
- ● Initialize DP table for visited cities.
- ● Start from the first city.
- ● Update DP table for all unvisited cities.
- ● Return to start and take minimum distance.

Program:

```
import sys

matrix = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

n = len(matrix)
VISITED_ALL = (1 << n) - 1

dp = [[sys.maxsize] * n for _ in range(1 << n)]

dp[1][0] = 0

for mask in range(1 << n):
    for i in range(n):
        if mask & (1 << i):
            for j in range(n):
                if not mask & (1 << j):
                    dp[mask | (1 << j)][j] = min(
                        dp[mask | (1 << j)][j],
                        dp[mask][i] + matrix[i][j]
                    )

ans = sys.maxsize
for i in range(1, n):
    ans = min(ans, dp[VISITED_ALL][i] + matrix[i][0])

print("Minimum path distance:", ans)
```

Sample Input:
{0,10,15,20}
{10,0,35,25}
{15,35,0,30}
{20,25,30,0}

Output:

```
Minimum path distance: 80


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:

Minimum 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.

Aim:

To determine the shortest possible route that visits all five cities exactly once and returns to the starting city, minimizing the total travel distance.

Algorithm:

- Store distances between all cities.
- Generate all possible routes starting and ending at the first city.
- Calculate total distance for each route.
- Keep track of the route with minimum distance.
- Output the shortest route and its total distance.

Program:

```
import itertools

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

dist = {
    ('A','B'): 10, ('B','A'): 10,
    ('A','C'): 15, ('C','A'): 15,
    ('A','D'): 20, ('D','A'): 20,
    ('A','E'): 25, ('E','A'): 25,
    ('B','C'): 35, ('C','B'): 35,
    ('B','D'): 25, ('D','B'): 25,
```

```
    ('B','E'): 30, ('E','B'): 30,
    ('C','D'): 30, ('D','C'): 30,
    ('C','E'): 20, ('E','C'): 20,
    ('D','E'): 15, ('E','D'): 15
}

perms = itertools.permutations(['B','C','D','E'])

min_distance = float('inf')
best_route = []

for perm in perms:
    route = ['A'] + list(perm) + ['A']
    distance = 0
    for i in range(len(route)-1):
        distance += dist[(route[i], route[i+1])]
    if distance < min_distance:
        min_distance = distance
        best_route = route

print("Shortest route:", ' -> '.join(best_route))
print("Total distance:", min_distance)
```

Sample Input:
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

Output:

```
Shortest route: A -> B -> D -> E -> C -> A
Total distance: 85



...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
Shortest distance = 85

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

Aim:
To find the longest palindromic substring in a given string s.

Algorithm:
- Input the string s.
- Initialize variables to store the starting index and maximum length of the palindrome.
- For each character in the string, expand around it to check odd-length palindromes.
- Expand between consecutive characters to check even-length palindromes.
- Update the longest palindrome found and display the result.

Program:
```
s = "babad"

start = 0
max_len = 1
n = len(s)

for i in range(n):
    left = i
    right = i
    while left >= 0 and right < n and s[left] == s[right]:
        if right - left + 1 > max_len:
            start = left
            max_len = right - left + 1

        left -= 1
        right += 1
    left = i
    right = i + 1
    while left >= 0 and right < n and s[left] == s[right]:
        if right - left + 1 > max_len:
            start = left
            max_len = right - left + 1
        left -= 1
        right += 1
```
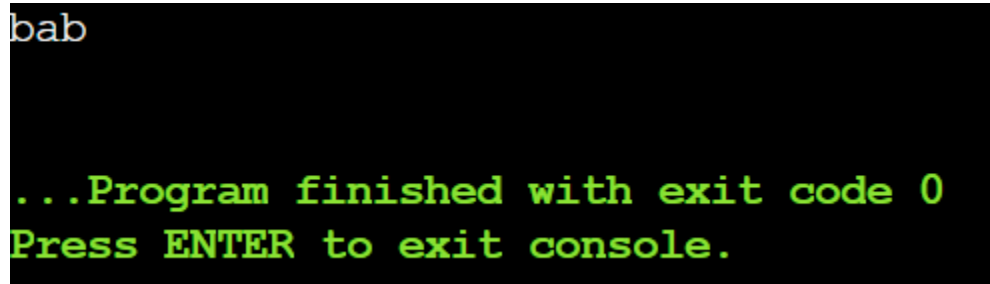
```
result = s[start:start + max_len]
print(result)
```

Sample Input:
s = "babad"

Output:

```
bab




...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
The algorithm efficiently finds the longest palindromic substring in $O(n^2)$ time and $O(1)$ extra space.

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

Aim:
To find the length of the longest substring without repeating characters in a given string.

Algorithm:
- Read the input string s.
- Initialize an empty set, two pointers, and a variable for maximum length.
- Move the right pointer through the string.
- Remove characters from the left when repetition occurs.
- Update the maximum length.
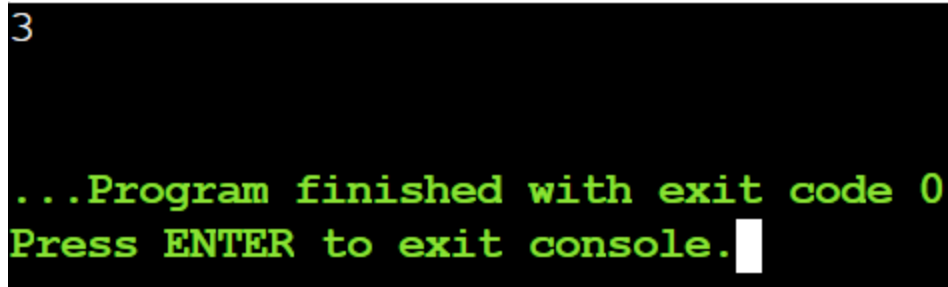
Program:
s = "abcabcbb"

char_set = set()
left = 0
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])
    max_length = max(max_length, right - left + 1)

print(max_length)
```

Sample Input:
Input: s = "abcabcbb"

Output:

Result:
The length of the longest substring without repeating characters is 3 (for input "abcabcbb").

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.

Aim:
To determine whether a given string can be segmented into a sequence of one or more dictionary words.

Algorithm:
- Read the input string s and the dictionary wordDict.
- Create a boolean array dp to store segmentation possibilities.
- Set dp[0] as true to represent an empty string.
- For each position in the string, check all dictionary words.
- Mark dp[n] as true if the full string can be segmented and display the result.

Program:
s = "leetcode"
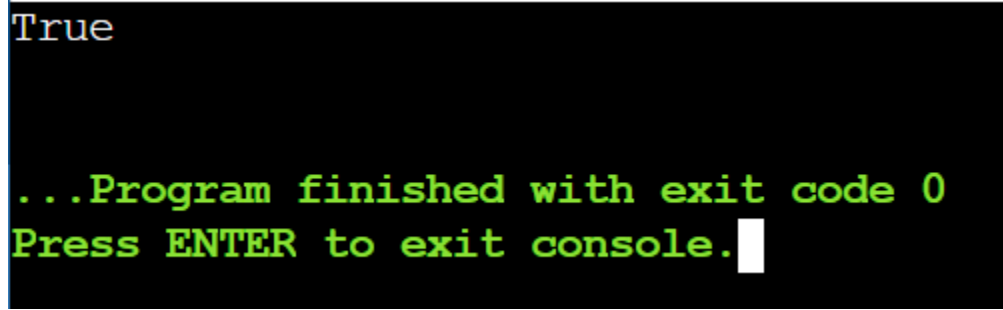wordDict = ["leet", "code"]

n = len(s)
dp = [False] * (n + 1)
dp[0] = True # Empty string can be segmented

for i in range(1, n + 1):
    for word in wordDict:
        if i >= len(word) and dp[i - len(word)] and s[i - len(word):i] == word:
            dp[i] = True
            break

print(dp[n])

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

Output:

```
True


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
The string can be segmented using the given dictionary words.

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}

Aim:
To determine whether a given string can be segmented into a sequence of dictionary words and display the segmented string if possible.

Algorithm:

- Read the input string and dictionary.
- Initialize a DP array to track segmentation.
- Check each substring against dictionary words.
- Mark positions as segmentable and store split points.
- Print result and segmented string if possible.

Program:

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

n = len(s)
dp = [False] * (n + 1)
dp[0] = True  # Empty string can be segmented
backtrack = [0] * (n + 1)

for i in range(1, n + 1):
    for word in wordDict:
        if i >= len(word) and dp[i - len(word)] and s[i - len(word):i] == word:
            dp[i] = True
            backtrack[i] = i - len(word)
            break

if dp[n]:
    print("Yes")

    words = []
    idx = n
    while idx > 0:
        start = backtrack[idx]
        words.append(s[start:idx])
        idx = start
    words.reverse()
    print("Segmented string:", " ".join(words))
else:
    print("No")
```

Sample Input:
Ilike

Output:

```
Yes
Segmented string: i like


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
Yes, the string can be segmented.

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.

Aim:
To format a list of words into lines of a given width such that each line is fully justified (left and right) and spaces are distributed evenly.

Algorithm:
- Read words and maxWidth.
- Pack as many words as fit in a line.
- Distribute spaces evenly between words.
- Left-justify the last line or single-word lines.
- Append each line to the result.

Program:

```python
words = ["This", "is", "an", "example", "of", "text", "justification."]
maxWidth = 16

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
    line = ""

    if j == n or num_words == 1:
        line = " ".join(line_words)
        line += " " * (maxWidth - len(line))
    else:
        total_spaces = maxWidth - sum(len(word) for word in line_words)
        space_between = total_spaces // (num_words - 1)
        extra_spaces = total_spaces % (num_words - 1)

        for k in range(num_words - 1):
            line += line_words[k]
            line += " " * (space_between + (1 if k < extra_spaces else 0))
        line += line_words[-1]

    res.append(line)
    i = j

for l in res:
    print(f"'{l}'")
```
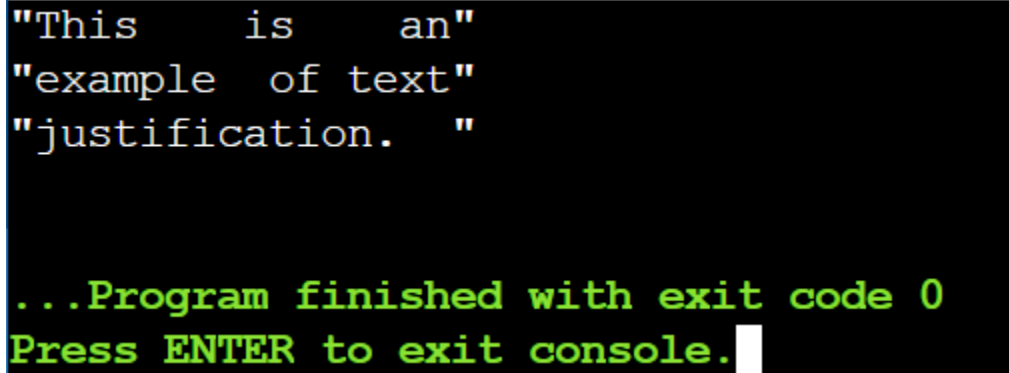
Sample Input:
words = ["This", "is", "an", "example", "of", "text", "justification."],
maxWidth = 16

Output:
```
"This      is      an"
"example  of text"
"justification.   "



...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
Text is fully justified.

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.

Aim:
To design a dictionary that allows searching for words by a given prefix and suffix, returning the largest index of a matching word.

Algorithm:

- Read the list of words and store each word with its index.
- For each query, get the prefix and suffix.
- Check each word to see if it starts with the prefix and ends with the suffix.
- Keep track of the largest index of matching words.
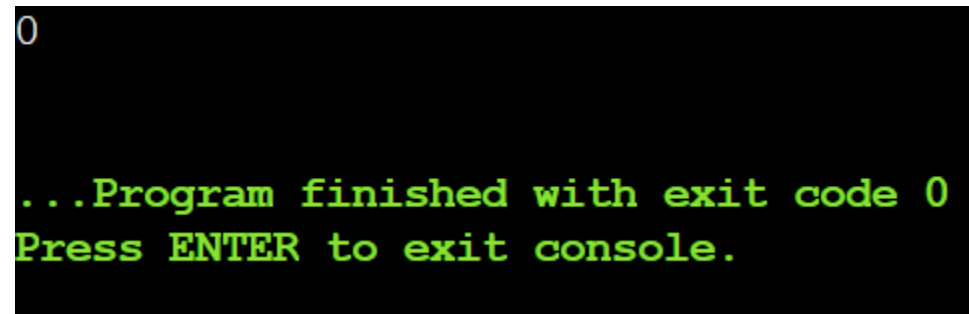- Return the largest index if found, otherwise return -1.

Program:
```
words = ["apple"]
queries = [("a", "e")]  # List of (prefix, suffix) queries

word_dict = {}
for i, word in enumerate(words):
    word_dict[word] = i

for pref, suff in queries:
    max_index = -1
    for i, word in enumerate(words):
        if word.startswith(pref) and word.endswith(suff):
            max_index = max(max_index, i)
    print(max_index)
```

Sample Input:
["WordFilter", "f"]
[[["apple"]], ["a", "e"]]

Output:


```
0

...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
Returns 0, as "apple" matches the prefix "a" and suffix "e".

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.

Aim:
To find the shortest path between all pairs of cities using Floyd-Warshall Algorithm, display the distance matrix before and after, and identify cities reachable within a given distance threshold.

Algorithm:

- Read number of cities, edges, and distance threshold.
- Initialize distance matrix with inf and 0 for self-loops.
- Fill distance matrix with given edge weights.
- Update distances using Floyd-Warshall for all pairs via intermediate cities.
- Display distance matrix and count reachable cities within threshold.

Program:

```
import math
n = 4
edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]]
distanceThreshold = 4

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

print("Distance matrix before Floyd-Warshall:")
for row in dist:
    print(row)

for k in range(n):
    for i in range(n):
        for j in range(n):
            if dist[i][j] > dist[i][k] + dist[k][j]:
                dist[i][j] = dist[i][k] + dist[k][j]
print("\nDistance matrix after Floyd-Warshall:")
for row in dist:
    print(row)
reachable_count = []
for i in range(n):
    count = sum(1 for d in dist[i] if 0 < d <= distanceThreshold)
    reachable_count.append(count)
    print(f"City {i} can reach {count} cities within distance {distanceThreshold}")
print("\nOutput:", max(reachable_count))
```

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

Output:

```
Distance matrix before Floyd-Warshall:
[0, 3, inf, inf]
[3, 0, 1, 4]
[inf, 1, 0, 1]
[inf, 4, 1, 0]

Distance matrix after Floyd-Warshall:
[0, 3, 4, 5]
[3, 0, 1, 2]
[4, 1, 0, 1]
[5, 2, 1, 0]
City 0 can reach 2 cities within distance 4
City 1 can reach 3 cities within distance 4
City 2 can reach 3 cities within distance 4
City 3 can reach 2 cities within distance 4

Output: 3



...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
Maximum number of cities reachable within distance 4 is 3.

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.

Aim:

To implement Floyd-Warshall Algorithm to find the shortest paths between all pairs of routers, simulate a link failure, and update the shortest path accordingly.

Algorithm:

- Read the number of routers and initialize the distance matrix with inf and 0 for self-loops.
- Fill the distance matrix with the given link weights between routers.
- Apply Floyd-Warshall Algorithm to compute shortest paths between all pairs.
- Display the shortest path from Router A to Router F.
- Simulate link failure by setting the failed link's distance to inf and reapply Floyd-Warshall.
- Display the updated shortest path from Router A to Router F.

Program:

```
import math

n = 6
edges = [
    [0,1,2],
    [0,2,5],
    [1,2,4],
    [1,3,6],
    [2,3,2],
    [2,4,3],
    [3,4,1],
    [4,5,2],
    [3,5,5]
]

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

print("Distance matrix before link failure:")
for row in dist:
```

```python
        print(row)

for k in range(n):
    for i in range(n):
        for j in range(n):
            if dist[i][j] > dist[i][k] + dist[k][j]:
                dist[i][j] = dist[i][k] + dist[k][j]
print("\nShortest path from Router A to Router F before link failure:", dist[0][5])

dist[1][3] = math.inf
dist[3][1] = math.inf

for k in range(n):
    for i in range(n):
        for j in range(n):
            if dist[i][j] > dist[i][k] + dist[k][j]:
                dist[i][j] = dist[i][k] + dist[k][j]

print("Shortest path from Router A to Router F after link failure:", dist[0][5])
```

Sample Input:
as above

Output:

```
Distance matrix before link failure:
[0, 2, 5, inf, inf, inf]
[2, 0, 4, 6, inf, inf]
[5, 4, 0, 2, 3, inf]
[inf, 6, 2, 0, 1, 5]
[inf, inf, 3, 1, 0, 2]
[inf, inf, inf, 5, 2, 0]

Shortest path from Router A to Router F before link failure: 10
Shortest path from Router A to Router F after link failure: 10


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:

Shortest path from Router A to Router F = 10 (before failure) and updated path after failure.

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.

Aim:
To find the shortest paths between all pairs of cities using Floyd-Warshall Algorithm, display the distance matrix before and after, and identify cities reachable within a given distance threshold.

Algorithm:

- Read the number of cities, edges, and distance threshold.
- Initialize a distance matrix with inf and 0 for self-loops.
- Fill the distance matrix with the given edge weights.
- Apply Floyd-Warshall Algorithm to update shortest paths between all pairs.
- Display the distance matrix after computation.
- Count and identify cities reachable within the distance threshold.

Program:
```
import math

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

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

print("Distance matrix before Floyd-Warshall:")
for row in dist:
    print(row)

for k in range(n):
    for i in range(n):
```

```python
        for j in range(n):
            if dist[i][j] > dist[i][k] + dist[k][j]:
                dist[i][j] = dist[i][k] + dist[k][j]

print("\nDistance matrix after Floyd-Warshall:")
for row in dist:
    print(row)

reachable_count = []
for i in range(n):
    count = sum(1 for d in dist[i] if 0 < d <= distanceThreshold)
    reachable_count.append(count)
    print(f"City {i} can reach {count} cities within distance {distanceThreshold}")

min_reachable = min(reachable_count)
print("\nOutput:", min_reachable)
```

Sample 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:

```
Distance matrix before Floyd-Warshall:
[0, 2, inf, inf, 8]
[2, 0, 3, inf, 2]
[inf, 3, 0, 1, inf]
[inf, inf, 1, 0, 1]
[8, 2, inf, 1, 0]

Distance matrix after Floyd-Warshall:
[0, 2, 5, 5, 4]
[2, 0, 3, 3, 2]
[5, 3, 0, 1, 2]
[5, 3, 1, 0, 1]
[4, 2, 2, 1, 0]
City 0 can reach 1 cities within distance 2
City 1 can reach 2 cities within distance 2
City 2 can reach 2 cities within distance 2
City 3 can reach 2 cities within distance 2
City 4 can reach 3 cities within distance 2

Output: 1


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
Minimum number of cities reachable within distance 2 is 1.

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.

Aim:

To construct an Optimal Binary Search Tree (OBST) for given keys and their frequencies such that the expected search cost is minimized, and display its cost and structure.

Algorithm:

- Input keys and their frequencies.
- Initialize cost and root matrices.
- Set cost[i][i] = freq[i] and root[i][i] = i.
- For chain lengths 2 to n, calculate cost[i][j] using all possible roots r and update root[i][j].
- Use root matrix to construct OBST and display cost.

Program:

```
import pprint

keys = ['A', 'B', 'C', 'D']
freq = [0.1, 0.2, 0.4, 0.3]
n = len(keys)

cost = [[0 for _ in range(n)] for _ in range(n)]
root = [[0 for _ in range(n)] for _ in range(n)]

for i in range(n):
    cost[i][i] = freq[i]
    root[i][i] = i

for l in range(2, n+1):
    for i in range(n-l+1):
        j = i + l - 1
        cost[i][j] = float('inf')

        fsum = sum(freq[i:j+1])

        for r in range(i, j+1):
            c = (0 if r==i else cost[i][r-1]) + \
                (0 if r==j else cost[r+1][j]) + fsum
            if c < cost[i][j]:
                cost[i][j] = c
                root[i][j] = r
```

```python
def print_obst(root, keys, i, j, parent=None, side='root'):
    if i > j:
        return
    r = root[i][j]
    node = keys[r]
    if parent is None:
        print(f"{node} is {side}")
    else:
        print(f"{node} is {side} child of {parent}")
    print_obst(root, keys, i, r-1, node, 'left')
    print_obst(root, keys, r+1, j, node, 'right')

print("Cost Matrix:")
pprint.pprint(cost)
print("\nRoot Matrix:")
pprint.pprint(root)

print("\nOptimal BST Structure:")
print_obst(root, keys, 0, n-1)

print("\nCost of Optimal BST:", round(cost[0][n-1],2))
```

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

Output:

```
Cost Matrix:
[[0.1, 0.4, 1.1, 1.7], [0, 0.2, 0.8, 1.4], [0, 0, 0.4, 1.0], [0, 0, 0, 0.3]]

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

Optimal BST Structure:
C is root
B is left child of C
A is left child of B
D is right child of C

Cost of Optimal BST: 1.7


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:

Optimal BST Cost = 1.7

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.

Aim:
To construct an Optimal Binary Search Tree (OBST) for given keys and frequencies with minimum search cost.

Algorithm:

1. Input keys and their frequencies.
2. Initialize cost and root matrices.
3. Set cost[i][i] = freq[i] and root[i][i] = i.
4. Use dynamic programming to fill cost[i][j] and root[i][j] for all subarrays.
5. Construct OBST from root matrix and display cost.

Program:
```
import pprint

keys = [10, 12, 16, 21]
freq = [4, 2, 6, 3]
n = len(keys)

cost = [[0 for _ in range(n)] for _ in range(n)]
root = [[0 for _ in range(n)] for _ in range(n)]

for i in range(n):
    cost[i][i] = freq[i]
    root[i][i] = i

for l in range(2, n+1):
    for i in range(n - l + 1):
        j = i + l - 1
        cost[i][j] = float('inf')
        fsum = sum(freq[i:j+1])
        for r in range(i, j+1):
            left = 0 if r == i else cost[i][r-1]
```

```python
            right = 0 if r == j else cost[r+1][j]
            c = left + right + fsum
            if c < cost[i][j]:
                cost[i][j] = c
                root[i][j] = r


print("Cost Matrix:")
pprint.pprint(cost)

print("\nRoot Matrix:")
pprint.pprint(root)

stack = [(0, n-1, None, 'root')]  # (i,j,parent,side)
while stack:
    i, j, parent, side = stack.pop()
    if i > j:
        continue
    r = root[i][j]
    node = keys[r]
    if parent is None:
        print(f"{node} is {side}")
    else:
        print(f"{node} is {side} child of {parent}")

    stack.append((r+1, j, node, 'right'))
    stack.append((i, r-1, node, 'left'))

print("\nCost of Optimal BST:", cost[0][n-1])
```

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

Output:

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

Root Matrix:
[[0, 0, 2, 2], [0, 1, 2, 2], [0, 0, 2, 2], [0, 0, 0, 3]]
16 is root
10 is left child of 16
12 is right child of 10
21 is right child of 16

Cost of Optimal BST: 26


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:

OBST cost = 26, root = 16, with 12 as left child, 10 as left of 12, and 21 as right child.

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:

Aim:

To determine the outcome of the Cat and Mouse game on a graph assuming both play optimally: mouse wins, cat wins, or draw.

Algorithm:

- Represent the graph and initialize a memo table for all (mouse, cat, turn) states.
- Set base cases: mouse at hole → mouse wins, cat catches mouse → cat wins.
- Use BFS to propagate win/loss results through all previous positions.
- For each turn, check next possible positions for mouse or cat.

Program:
```
from collections import deque

graph = [[1,3],[0],[3],[0,2]]
n = len(graph)
 wins)
memo = [[[-1]*2 for _ in range(n)] for _ in range(n)]

queue = deque()

for c in range(n):
    if c != 0:
        memo[0][c][0] = 1
        memo[0][c][1] = 1
        queue.append((0,c,0))
        queue.append((0,c,1))

for m in range(1,n):
    memo[m][m][0] = 2
    memo[m][m][1] = 2
    queue.append((m,m,0))
    queue.append((m,m,1))

def next_positions(mouse, cat, turn):
    if turn == 0:
        for mnext in graph[mouse]:
            yield (mnext, cat, 1)
    else:
        for cnext in graph[cat]:
            if cnext != 0:
                yield (mouse, cnext, 0)

while queue:
    mouse, cat, turn = queue.popleft()
    result = memo[mouse][cat][turn]
    prev_turn = 1 - turn
    for pmouse, pcat, _ in next_positions(mouse, cat, prev_turn):
        if memo[pmouse][pcat][prev_turn] == -1:
            if result == (prev_turn + 1):
                memo[pmouse][pcat][prev_turn] = result
```
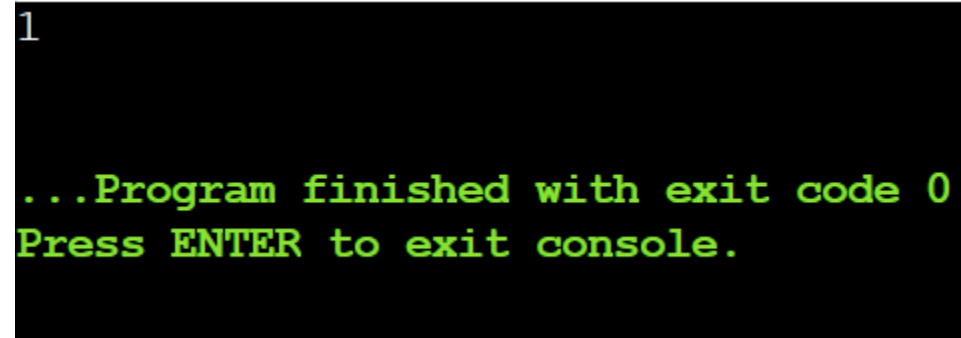
```
            queue.append((pmouse, pcat, prev_turn))
res = memo[1][2][0]
print(res)
```

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

Output:

Result:
1 → The game is a draw.

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.

Aim:
To find the path with the maximum probability of success between two nodes in an undirected weighted graph and return that probability.

Algorithm:
- Build adjacency list with probabilities.
- Use a max-heap starting from the start node with probability 1.
- Pop the node with the highest probability and update neighbors' probabilities.
- If end node is reached, return its probability.
- If not reachable, return 0.

Program:

```python
import heapq

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

graph = [[] for _ in range(n)]
for (u, v), prob in zip(edges, succProb):
    graph[u].append((v, prob))
    graph[v].append((u, prob))

heap = [(-1.0, start)]
visited = [0.0] * n
visited[start] = 1.0

while heap:
    prob, node = heapq.heappop(heap)
    prob = -prob
    if node == end:
        print(f"{prob:.5f}")
        break
    for nei, p in graph[node]:
        new_prob = prob * p
        if new_prob > visited[nei]:
            visited[nei] = new_prob
            heapq.heappush(heap, (-new_prob, nei))
else:
    print("0.00000")
```

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

Output:

```
0.25000



...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
Maximum probability from node 0 to 2 = 0.25

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.

Aim:
To find the total number of unique paths a robot can take from the top-left corner to the bottom-right corner of an m x n grid, moving only right or down.

Algorithm:

- Initialize a m x n DP table with all values as 1 (first row and column have 1 path).
- For each cell (i,j) starting from (1,1):
  - Set dp[i][j] = dp[i-1][j] + dp[i][j-1].
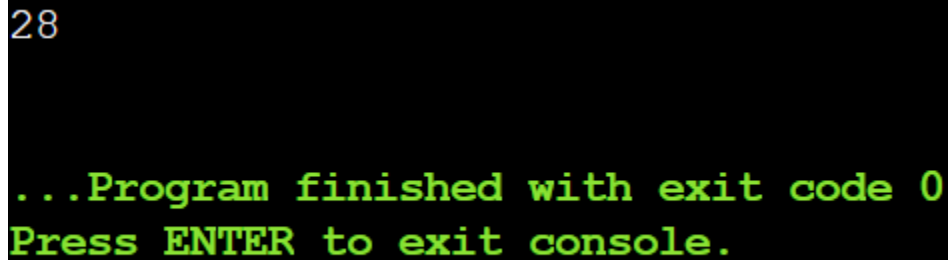- The value in dp[m-1][n-1] gives the total number of unique paths.

Program:
```
m = 3
n = 7

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]
print(dp[m-1][n-1])
```

Sample Input:
m = 3, n = 7

Output:

```
28


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
Number of unique paths = 28.

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.

Aim:
To count the number of good pairs in an array where a pair (i, j) is good if nums[i] == nums[j] and i < j.

Algorithm:

- Initialize a counter to 0.
- Loop through all elements of the array with index i.
- For each i, loop through all indices j > i.
- If nums[i] == nums[j], increment the counter.
- Return the counter as the number of good pairs.

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

count = 0
for i in range(len(nums)):
    for j in range(i+1, len(nums)):
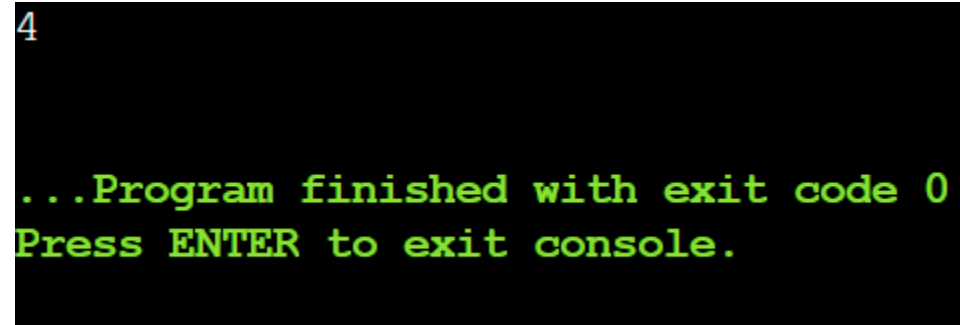   if nums[i] == nums[j]:
            count += 1
print(count)

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

Output:


```
4



...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
Number of good pairs = 4

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.

Aim:

To find the city with the smallest number of other cities reachable within a given distance threshold, and return the greatest city number in case of a tie.

Algorithm:

- Initialize an n x n distance matrix with INF and 0 for self-distances.
- Fill the matrix with given edge weights.
- Use Floyd-Warshall algorithm to compute shortest paths between all pairs of cities.
- For each city, count how many other cities are reachable within distanceThreshold.
- Return the city with the minimum count; if tied, choose the city with the greatest number.

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

INF = float('inf')
dist = [[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][j] > dist[i][k] + dist[k][j]:
                dist[i][j] = dist[i][k] + dist[k][j]

min_count = n
result_city = -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
        result_city = i # pick greatest city in case of tie

print(result_city)

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

Output:

```
3

...Program finished with exit code 0
Press ENTER to exit console.
```

Result:

City with smallest reachable cities within threshold = 3.

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.

Aim:

To find the minimum time it takes for a signal to reach all nodes in a network from a given starting node, or return -1 if any node is unreachable.

Algorithm:

- Build adjacency list for the network with travel times.
- Initialize distances from start node k (0 for k, infinity for others).
- Use Dijkstra's algorithm to update shortest distances.
- Find the maximum distance among all nodes.
- Return -1 if any node is unreachable, else return the maximum distance.

Program:

import heapq

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

graph = [[] for _ in range(n+1)]
for u, v, w in times:

```python
        graph[u].append((v, w))

dist = [float('inf')] * (n + 1)
dist[k] = 0
heap = [(0, k)]

while heap:
    time, node = heapq.heappop(heap)
    if time > dist[node]:
        continue
    for nei, w in graph[node]:
        if dist[nei] > time + w:
            dist[nei] = time + w
            heapq.heappush(heap, (dist[nei], nei))

res = max(dist[1:])
print(res if res != float('inf') else -1)
```
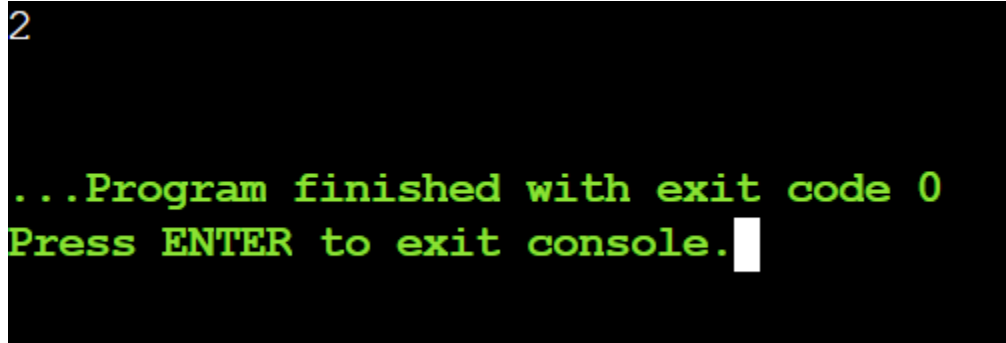
Sample Input:
times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2

Output:

Result:
Minimum time for all nodes to receive the signal = 2.