**DAA PROGRAMS:**

**Question 1: Handling Edge Cases in Selection Sort**

Discuss how your selection\_sort function handles edge cases. Specifically, consider and explain the outcomes for the following cases:

* + An empty list
  + A list with one element
  + A list with all identical elements
  + A list with negative numbers

**Test Cases:**

1. **Input:** []
   * **Expected Output:** []
2. **Input:** [1]
   * **Expected Output:** [1]
3. **Input:** [7, 7, 7, 7]
   * **Expected Output:** [7, 7, 7, 7]
4. **Input:** [-5, -1, -3, -2, -4]
   * **Expected Output:** [-5, -4, -3, -2, -1]

**Code:**

def selection\_sort(arr):  
 n = len(arr)  
 for i in range(n):  
 min\_index = i  
 for j in range(i+1,n):  
 if arr[j]<arr[min\_index]:  
 min\_index = j;  
 arr[i],arr[min\_index] = arr[min\_index],arr[i]  
 return arr  
arr = [1,4,3,2,7]  
print(selection\_sort(arr))

INPUT: [1,4,3,2,7]

OUTPUT:[1,2,3,4,7]

**Bubble Sort**

**Question-1 Optimizing Bubble Sort**

Modify your bubble\_sort function to stop early if the list becomes sorted before all passes are completed. Explain why this optimization improves performance and how it affects the time complexity in the best case.

**Test Cases:**

* Test your optimized function with the following lists:
  1. **Input:** [64, 25, 12, 22, 11]
     + **Expected Output:** [11, 12, 22, 25, 64]
  2. **Input:** [29, 10, 14, 37, 13]
     + **Expected Output:** [10, 13, 14, 29, 37]
  3. **Input:** [3, 5, 2, 1, 4]
     + **Expected Output:** [1, 2, 3, 4, 5]
  4. **Input:** [1, 2, 3, 4, 5] (Already sorted list)
     + **Expected Output:** [1, 2, 3, 4, 5]
  5. **Input:** [5, 4, 3, 2, 1] (Reverse sorted list)
     + **Expected Output:** [1, 2, 3, 4, 5]

CODE:

def bubble\_sort(arr):

n = len(arr)

for i in range(n):

swapped = False

for j in range(0, n-i-1):

if arr[j] > arr[j+1]:

arr[j], arr[j+1] = arr[j+1], arr[j]

swapped = True

if not swapped:

break

return arr

print(bubble\_sort([5, 2, 9, 1, 5, 6]))

print(bubble\_sort([10, 8, 6, 4, 2]))

print(bubble\_sort([1, 2, 3, 4, 5]))

INPUT: [5,2,9,1,5,6]

OUTPUT: [1,2,5,5,6,9]

**Question 1: Handling Duplicates in Insertion Sort**

Describe how Insertion Sort manages arrays with duplicate elements during the sorting process. Explain the algorithm's behavior when encountering duplicate values, including whether it preserves the relative order of duplicates and how it affects the overall sorting outcome. Provide specific examples with arrays containing duplicate integers, demonstrating how Insertion Sort sorts the array while ensuring duplicates are correctly positioned, and discuss any considerations or adjustments that might be necessary.

**Example Scenarios**:

1. **Array with Duplicates**:
   * **Input**: [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
   * **Output**: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]
2. **All Identical Elements**:
   * **Input**: [5, 5, 5, 5, 5]
   * **Output**: [5, 5, 5, 5, 5]
3. **Mixed Duplicates**:
   * **Input**: [2, 3, 1, 3, 2, 1, 1, 3]
   * **Output**: [1, 1, 1, 2, 2, 3, 3, 3]

CODE:

def insertion\_sort(arr):  
 n = len(arr)  
 for i in range(1,n):  
 temp = arr[i]  
 j = i-1  
 while j>=0 and arr[j]>temp:  
 arr[j+1] = arr[j]  
 j=j-1  
 arr[j+1] = temp  
 return arr  
arr = [3,2,1,5,6]  
print(insertion\_sort(arr))

**Input**: [2, 3, 1, 3, 2, 1, 1, 3]

**Output**: [1, 1, 1, 2, 2, 3, 3, 3]

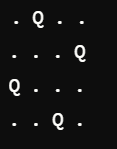
**N-Queens Problem**

**Question 1: Visualizing Solutions for the N-Queens Problem**

Discuss the importance of visualizing the solutions of the N-Queens Problem to understand the placement of queens better. Use a graphical representation to show how queens are placed on the board for different values of N. Explain how visual tools can help in debugging the algorithm and gaining insights into the problem's complexity. Provide examples of visual representations for N = 4, N = 5, and N = 8, showing different valid solutions.

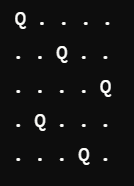
**Example Scenarios**:

1. **Visualization for 4-Queens**:
   * **Input**: N = 4
   * **Output**:

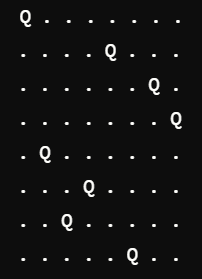


* + **Explanation**: Each 'Q' represents a queen, and '.' represents an empty space.

1. **Visualization for 5-Queens**:
   * **Input**: N = 5
   * **Output**:



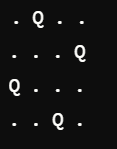
1. **Visualization for 8-Queens**:
   * **Input**: N = 8
   * **Output**:



CODE:

def is\_safe(board, row, col, n):  
  
 for i in range(row):  
 if board[i][col] == 'Q':  
 return False  
  
 for i, j in zip(range(row, -1, -1), range(col, -1, -1)):  
 if board[i][j] == 'Q':  
 return False  
  
  
 for i, j in zip(range(row, -1, -1), range(col, n)):  
 if board[i][j] == 'Q':  
 return False  
  
 return True  
  
def solve\_n\_queens\_util(board, row, n, solutions):  
  
 if row >= n:  
  
 solution = [''.join(row) for row in board]  
 solutions.append(solution)  
 return  
  
  
 for col in range(n):  
 if is\_safe(board, row, col, n):  
  
 board[row][col] = 'Q'  
  
 solve\_n\_queens\_util(board, row + 1, n, solutions)  
  
 board[row][col] = '.'  
  
def solve\_n\_queens(n):  
 board = [['.' for \_ in range(n)] for \_ in range(n)]  
 solutions = []  
 solve\_n\_queens\_util(board, 0, n, solutions)  
 return solutions  
   
n = 4  
solutions = solve\_n\_queens(n)  
for index, solution in enumerate(solutions):  
 print(f"Solution {index + 1}:")  
 for row in solution:  
 print(row)  
 print()

input: 4

output: 

**1. Sequential Search**

Given an array arr of positive integers sorted in a strictly increasing order, and an integer k.

Return the kth positive integer that is missing from this array.

Example 1:

Input: arr = [2,3,4,7,11], k = 5

Output: 9

Explanation: The missing positive integers are [1,5,6,8,9,10,12,13,...]. The 5th missing positive integer is 9.

Example 2:

Input: arr = [1,2,3,4], k = 2

Output: 6

Explanation: The missing positive integers are [5,6,7,...]. The 2nd missing positive integer is 6.

CODE:

def find\_kth\_positive(arr, k):

missing\_count = 0

prev = 0

for num in arr:

missing = num - prev - 1

if missing\_count + missing >= k:

return prev + k - missing\_count

missing\_count += missing

prev = num

return prev + (k - missing\_count)

arr1 = [2, 3, 4, 7, 11]

k1 = 5

print(find\_kth\_positive(arr1, k1))

arr2 = [1, 2, 3, 4]

k2 = 2

print(find\_kth\_positive(arr2, k2))

Input: arr = [1,2,3,4], k = 2

Output: 6

**2. Sequential Search**

A peak element is an element that is strictly greater than its neighbors.

Given a 0-indexed integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks.

You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

You must write an algorithm that runs in O(log n) time.

Example 1:

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

Output: 2

Explanation: 3 is a peak element and your function should return the index number 2.

Example 2:

Input: nums = [1,2,1,3,5,6,4]

Output: 5

Explanation: Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.

CODE:

def find\_peak\_element(nums):

left, right = 0, len(nums) - 1

while left < right:

mid = (left + right) // 2

if nums[mid] < nums[mid + 1]:

# Move to the right half

left = mid + 1

else:

right = mid

return left

nums1 = [1, 2, 3, 1]

print(find\_peak\_element(nums1))

nums2 = [1, 2, 1, 3, 5, 6, 4]

print(find\_peak\_element(nums2))

Input: nums = [1,2,1,3,5,6,4]

Output: 5

**3. Brute-Force String Matching**

Given two strings needle and haystack, return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

Example 1:

Input: haystack = "sadbutsad", needle = "sad"

Output: 0

Explanation: "sad" occurs at index 0 and 6.

The first occurrence is at index 0, so we return 0.

Example 2:

Input: haystack = "leetcode", needle = "leeto"

Output: -1

Explanation: "leeto" did not occur in "leetcode", so we return -1.

CODE:

def str\_str(haystack, needle):

if not needle:

return 0

needle\_len = len(needle)

haystack\_len = len(haystack)

for i in range(haystack\_len - needle\_len + 1):

if haystack[i:i + needle\_len] == needle:

return I

return -1

haystack1 = "sadbutsad"

needle1 = "sad"

print(str\_str(haystack1, needle1))

haystack2 = "leetcode"

needle2 = "leeto"

print(str\_str(haystack2, needle2))

Input: haystack = "leetcode", needle = "leeto"

Output: -1

**4. Brute-Force String Matching**

Given an array of string words, return all strings in words that is a substring of another word. You can return the answer in any order.

A substring is a contiguous sequence of characters within a string

Example 1:

Input: words = ["mass","as","hero","superhero"]

Output: ["as","hero"]

Explanation: "as" is substring of "mass" and "hero" is substring of "superhero".

["hero","as"] is also a valid answer.

Example 2:

Input: words = ["leetcode","et","code"]

Output: ["et","code"]

Explanation: "et", "code" are substring of "leetcode".

Example 3:

Input: words = ["blue","green","bu"]

Output: []

Explanation: No string of words is substring of another string.

CODE:

def string\_matching(words):

result = []

for i in range(len(words)):

for j in range(len(words)):

if i != j and words[i] in words[j]:

result.append(words[i])

break

return result

words1 = ["mass", "as", "hero", "superhero"]

print(string\_matching(words1))

words2 = ["leetcode", "et", "code"]

print(string\_matching(words2

words3 = ["blue", "green", "bu"]

print(string\_matching(words3))

Input: words = ["blue","green","bu"]

Output: []

**5. Word Break Problem**

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

CODE:

def wordBreak(s, wordDict):

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

dp[0] = True

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

for j in range(i):

if dp[j] and s[j:i] in wordDict:

dp[i] = True

break

return dp[-1]

s1 = "leetcode"

wordDict1 = ["leet", "code"]

print(wordBreak(s1, wordDict1 )

s2 = "applepenapple"

wordDict2 = ["apple", "pen"]

print(wordBreak(s2, wordDict2)) # Output: True

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

Output: true

**6. Word Break Problem**

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".

CODE:

def wordBreak(s, wordDict):

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

dp[0] = True

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

for j in range(i):

if dp[j] and s[j:i] in wordDict:

dp[i] = True

break

return "Yes" if dp[-1] else "No"

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

s1 = "ilike"

print(wordBreak(s1, wordDict))

s2 = "ilikesamsung"

print(wordBreak(s2, wordDict))

Input: ilikesamsung

Output: Yes

**7. Word Wrap Problem**

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.

Note:

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.

Example 3:

Input: words = ["Science","is","what","we","understand","well","enough","to","explain","to","a","computer.","Art","is","everything","else","we","do"], maxWidth = 20

Output:

[

"Science is what we", "understand well", "enough to explain to",

"a computer. Art is", "everything else we", "do"

]

CODE:

def fullJustify(words, maxWidth):

result, cur\_line, num\_of\_letters = [], [], 0

for word in words:

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

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

cur\_line[i % (len(cur\_line) - 1 or 1)] += ' '

result.append(''.join(cur\_line))

cur\_line, num\_of\_letters = [], 0

cur\_line.append(word)

num\_of\_letters += len(word)

result.append(' '.join(cur\_line).ljust(maxWidth))

return result

words1 = ["This", "is", "an", "example", "of", "text", "justification."]

maxWidth1 = 16

print(fullJustify(words1, maxWidth1))

words2 = ["What","must","be","acknowledgment","shall","be"]

maxWidth2 = 16

print(fullJustify(words2, maxWidth2))

Input: words = ["Science","is","what","we","understand","well","enough","to","explain","to","a","computer.","Art","is","everything","else","we","do"], maxWidth = 20

Output:

[

"Science is what we", "understand well", "enough to explain to",

"a computer. Art is", "everything else we", "do"

]

**8. Word Wrap Problem**

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".

CODE:

class WordFilter:

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

self.prefix\_suffix\_map = {}

for index, word in enumerate(words):

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

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

self.prefix\_suffix\_map[word[:i], word[j:]] = index

def f(self, pref, suff):

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

wordFilter = WordFilter(["apple"])

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

Input

["WordFilter", "f"]

[[["apple"]], ["a", "e"]]

Output: [null, 0]

**9. Warshall’s & Floyd’s Algorithm**

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

CODE:

def uniquePaths(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]

m1, n1 = 3, 7

print(uniquePaths(m1, n1))

m2, n2 = 3, 2

print(uniquePaths(m2, n2))

Input: m = 3, n = 7

Output: 28

10.

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.

Example 3:

Input: nums = [1,2,3]

Output: 0

CODE:

def numIdenticalPairs(nums):

count = 0

freq = {}

for num in nums:

if num in freq:

count += freq[num]

freq[num] += 1

else:

freq[num] = 1

return count

nums1 = [1, 2, 3, 1, 1, 3]

print(numIdenticalPairs(nums1))

nums2 = [1, 1, 1, 1]

print(numIdenticalPairs(nums2))

Input: nums = [1,2,3]

Output: 0

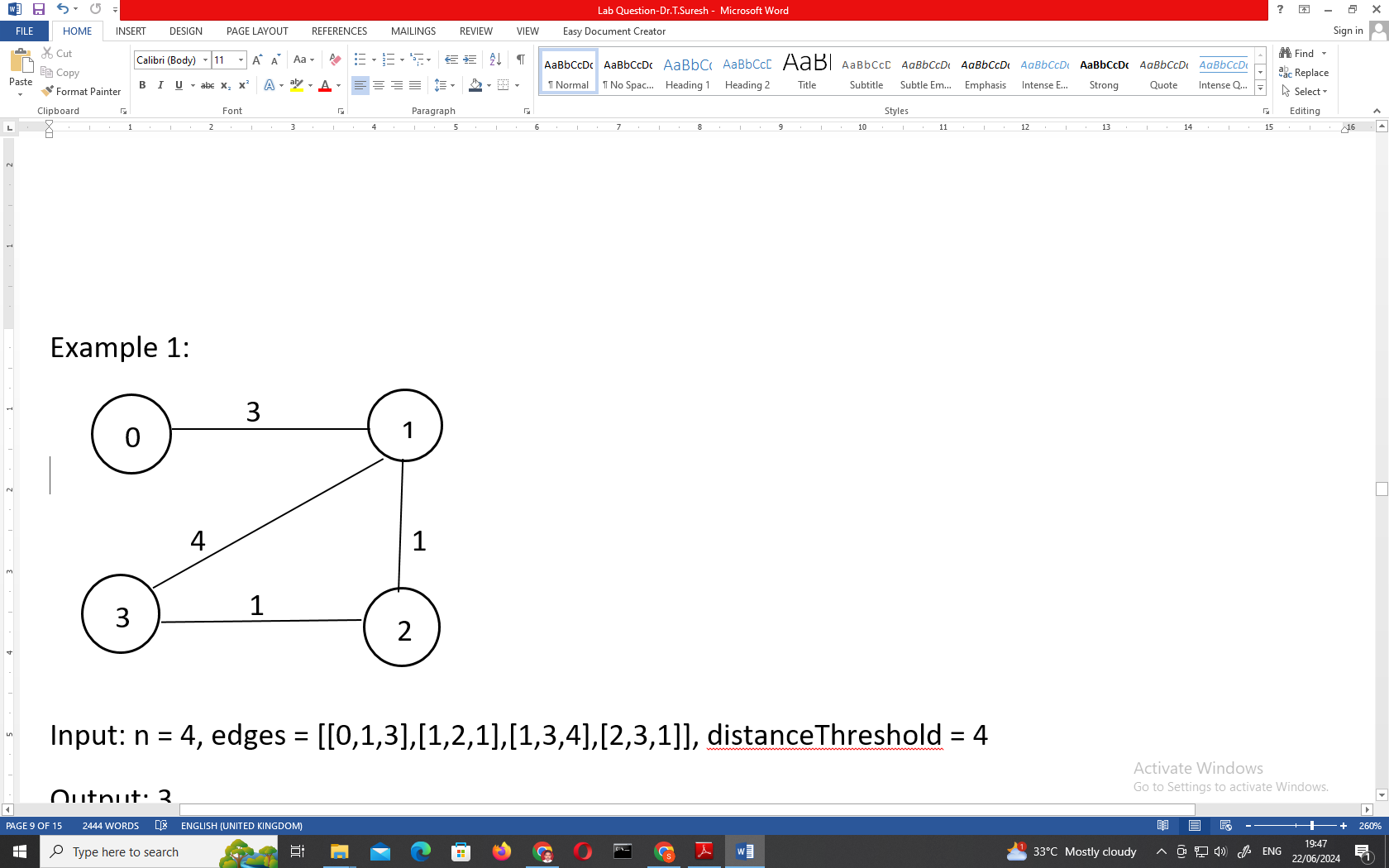
**11. Warshall’s & Floyd’s Algorithm**

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]

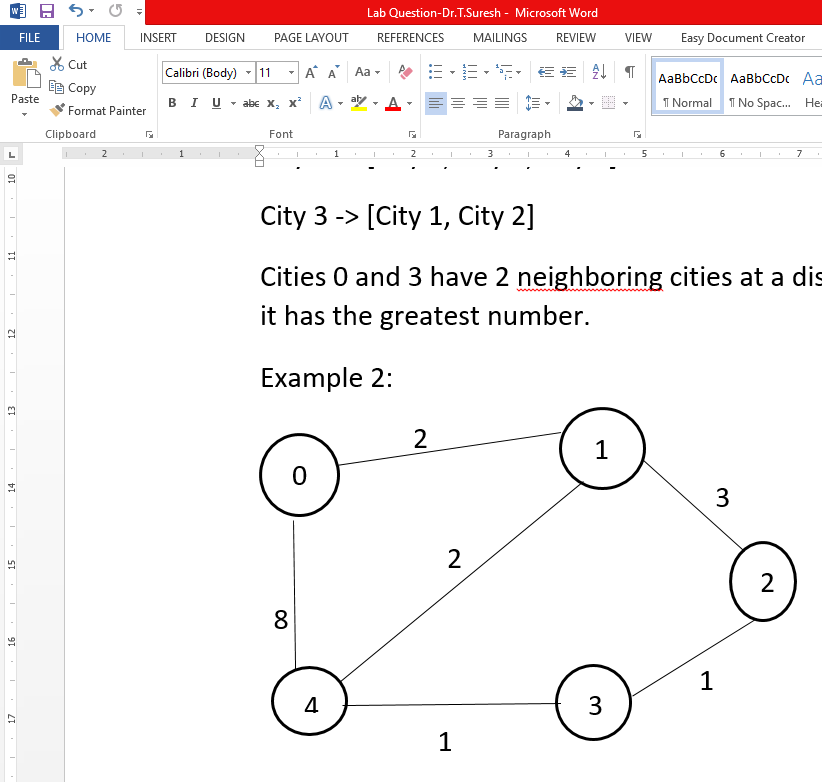
City 1 -> [City 0, City 2, City 3]

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.

CODE:

def findTheCity(n, edges, distanceThreshold):

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\_reachable = inf

city\_with\_min\_reachable = -1

for i in range(n):

reachable\_count = sum(1 for j in range(n) if dist[i][j] <= distanceThreshold)

if (reachable\_count < min\_reachable) or (reachable\_count == min\_reachable and i > city\_with\_min\_reachable):

min\_reachable = reachable\_count

city\_with\_min\_reachable = i

return city\_with\_min\_reachable

n1 = 4

edges1 = [[0, 1, 3], [1, 2, 1], [1, 3, 4], [2, 3, 1]]

distanceThreshold1 = 4

print(findTheCity(n1, edges1, distanceThreshold1))

n2 = 5

edges2 = [[0, 1, 2], [0, 4, 8], [1, 2, 3], [1, 4, 2], [2, 3, 1], [3, 4, 1]]

distanceThreshold2 = 2

print(findTheCity(n2, edges2, distanceThreshold2))

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

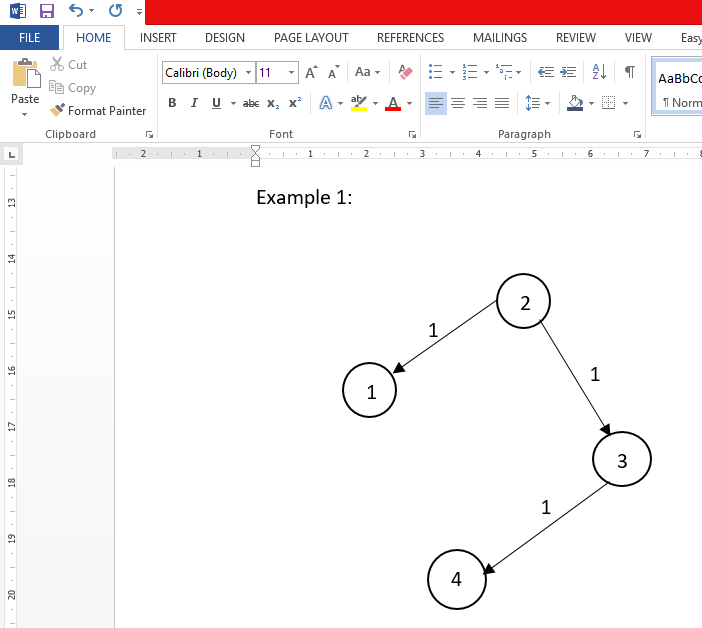
Output: 0

**12. Warshall’s & Floyd’s Algorithm**

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

CODE:

def networkDelayTime(times, n, k):

inf = float('inf')

dist = [inf] \* n

dist[k - 1] = 0

for \_ in range(n - 1):

for u, v, w in times:

if dist[u - 1] != inf and dist[u - 1] + w < dist[v - 1]:

dist[v - 1] = dist[u - 1] + w

max\_dist = max(dist)

return max\_dist if max\_dist != inf else -1

times1 = [[2, 1, 1], [2, 3, 1], [3, 4, 1]]

n1 = 4

k1 = 2

print(networkDelayTime(times1, n1, k1))

times2 = [[1, 2, 1]]

n2 = 2

k2 = 1

print(networkDelay Time(times2, n2, k2))

times3 = [[1, 2, 1]]

n3 = 2

k3 = 2

print(networkDelayTime(times3, n3, k3))

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

Output: -1

**13. Bellman-Ford Algorithm**

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.

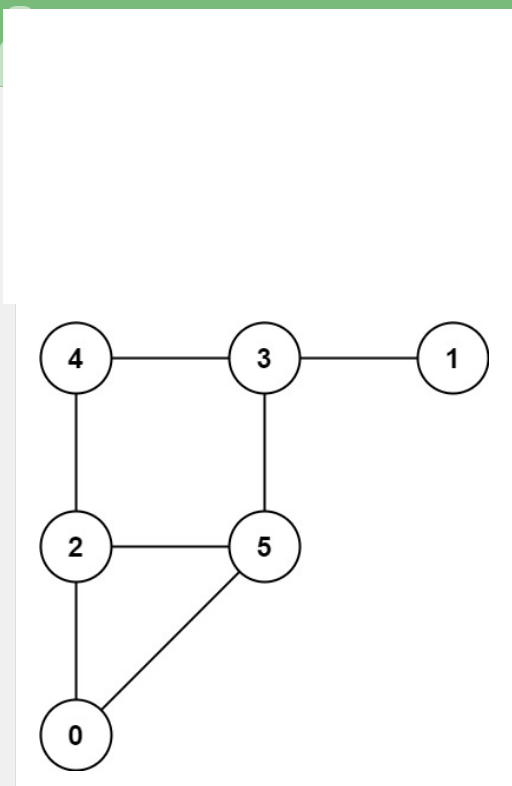
Given a graph, and assuming both players play optimally, return

1 if the mouse wins the game,

2 if the cat wins the game, or

0 if the game is a draw.

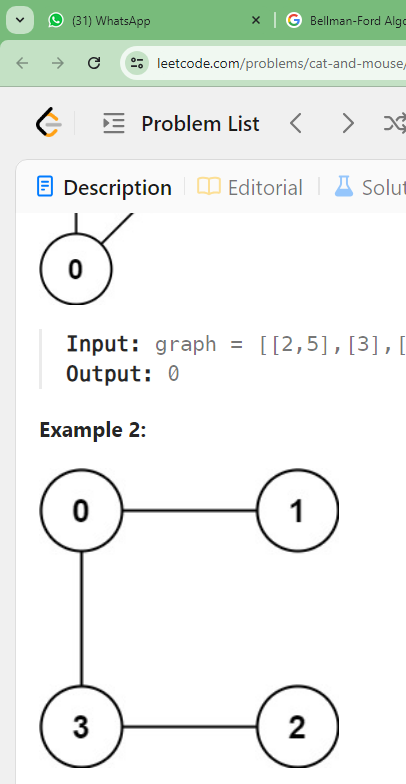
Example 1:



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

CODE:

def catMouseGame(graph):

n = len(graph)

dp = [[[0] \* n for \_ in range(n)] for \_ in range(2 \* n)]

def dfs(t, mouse, cat):

if t == 2 \* n:

return 0

if dp[t][mouse][cat] != 0:

return dp[t][mouse][cat]

if mouse == 0:

return 1

if mouse == cat:

return 2

if t % 2 == 0:

result = 2

for nei in graph[mouse]:

res = dfs(t + 1, nei, cat)

if res == 1:

dp[t][mouse][cat] = 1

return 1

if res == 0:

result = 0

dp[t][mouse][cat] = result

return result

else:

result = 1

for nei in graph[cat]:

if nei == 0:

continue

res = dfs(t + 1, mouse, nei)

if res == 2:

dp[t][mouse][cat] = 2

return 2

if res == 0:

result = 0

dp[t][mouse][cat] = result

return result

return dfs(0, 1, 2)

graph1 = [[2, 5], [3], [0, 4, 5], [1, 4, 5], [2, 3], [0, 2, 3]]

print(catMouseGame(graph1))

graph2 = [[1, 3], [0], [3], [0, 2]]

print(catMouseGame(graph2))

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

Output: 1

**14. Bellman-Ford Algorithm**

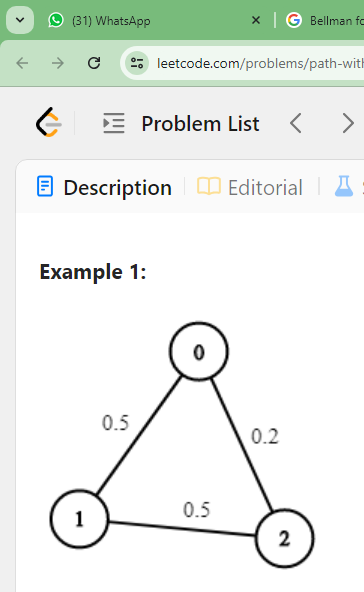
**Path with Maximum Probability**

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:

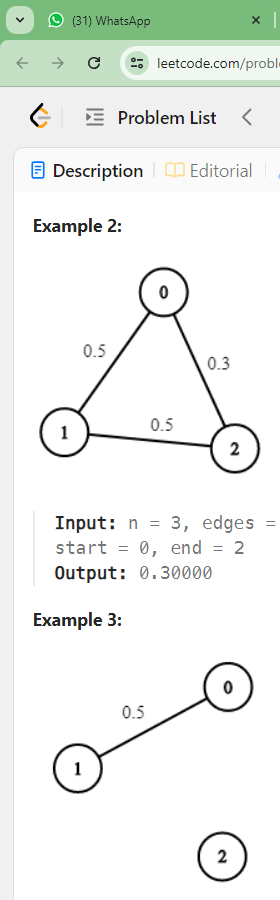


Input: n = 3, edges = [[0,1],[1,2],[0,2]], succProb = [0.5,0.5,0.2], start = 0, end = 2

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.

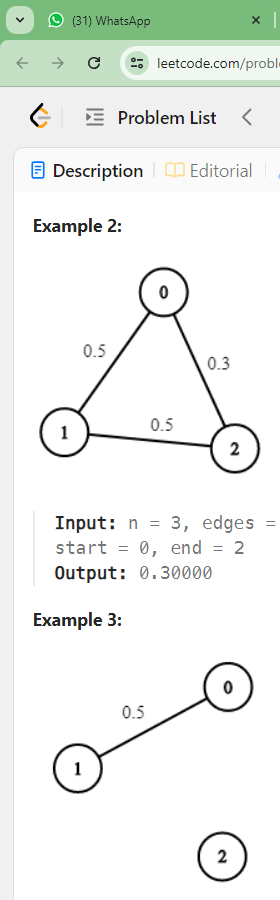
Example 2:



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

Output: 0.30000

Example 3:



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

Output: 0.00000

Explanation: There is no path between 0 and 2.

CODE:

import heapq

def maxProbability(n, edges, succProb, start, end):

graph = [[] for \_ in range(n)]

for i, (u, v) in enumerate(edges):

graph[u].append((v, succProb[i]))

graph[v].append((u, succProb[i]))

prob = [0.0] \* n

prob[start] = 1.0

pq = [(-1.0, start)]

while pq:

curr\_prob, node = heapq.heappop(pq)

curr\_prob = -curr\_prob

if node == end:

return curr\_prob

for neighbor, edge\_prob in graph[node]:

new\_prob = curr\_prob \* edge\_prob

if new\_prob > prob[neighbor]:

prob[neighbor] = new\_prob

heapq.heappush(pq, (-new\_prob, neighbor))

return 0.0

n1 = 3

edges1 = [[0, 1], [1, 2], [0, 2]]

succProb1 = [0.5, 0.5, 0.2]

start1 = 0

end1 = 2

print(maxProbability(n1, edges1, succProb1, start1, end1))

n2 = 3

edges2 = [[0, 1], [1, 2], [0, 2]]

succProb2 = [0.5, 0.5, 0.3]

start2 = 0

end2 = 2

print(maxProbability(n2, edges2, succProb2, start2, end2))

n3 = 3

edges3 = [[0, 1]]

succProb3 = [0.5]

start3 = 0

end3 = 2

print(maxProbability(n3, edges3, succProb3, start3, end3))

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

Output: 0.00000