DAA – Lab Questions Dr.T.Suresh Balakrishnan

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

CODE:

def findKthPositive(arr, k):

# Initialize the current number to check, and the missing number count

current\_num = 1

missing\_count = 0

# Loop through the array and count missing numbers

for num in arr:

# Check for missing numbers between current\_num and the array's number

while current\_num < num:

missing\_count += 1

if missing\_count == k:

return current\_num

current\_num += 1

# Move to the next number in the sequence

current\_num = num + 1

# If we didn't find k missing numbers in the loop, continue counting after the array's end

while missing\_count < k:

missing\_count += 1

if missing\_count == k:

return current\_num

current\_num += 1

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

k = 5

print(findKthPositive(arr, k)) # Output: 9

OUTPUT: 9

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.

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

CODE:

def findPeakElement(nums):

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

while left < right:

mid = (left + right) // 2

# Compare middle element with its right neighbor

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

# Peak must be in the right half

left = mid + 1

else:

# Peak is in the left half (including mid)

right = mid

# When left == right, we've found a peak

return left

nums = [1, 2, 3, 1]

print(findPeakElement(nums))

OUTPUT: 2

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.

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

CODE:

def strStr(haystack, needle):

# If the needle is an empty string, return 0 as per the problem's assumption

if not needle:

return 0

# Get the lengths of the haystack and needle

len\_haystack = len(haystack)

len\_needle = len(needle)

# Loop through the haystack to find the needle

for i in range(len\_haystack - len\_needle + 1):

# Compare the substring of haystack with the needle

if haystack[i:i+len\_needle] == needle:

return i

# If the needle is not found, return -1

return -1

haystack = "sadbutsad"

needle = "sad"

print(strStr(haystack, needle))

Output: 0

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

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

CODE:

def stringMatching(words):

result = []

# Loop through each word and compare it with other words in the list

for i in range(len(words)):

for j in range(len(words)):

# Ensure we're not comparing the same word with itself

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

result.append(words[i])

break # No need to check other words once we find a match

return result

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

print(stringMatching(words))

Output: ["as","hero"]

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.

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

CODE:

def wordBreak(s, wordDict):

# Convert the word dictionary to a set for faster lookups

word\_set = set(wordDict)

# Create a dp array of size len(s) + 1, initialized to False

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

# The empty string can always be segmented

dp[0] = True

# Check each substring s[0:i]

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

for j in range(i):

# If the substring s[0:j] can be segmented (dp[j] is True) and

# the substring s[j:i] is in the word dictionary, then s[0:i] can be segmented

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

dp[i] = True

break

# Return whether the entire string s can be segmented

return dp[len(s)]

# Example Input:

s = "leetcode"

wordDict = ["leet", "code"]

# Function call to check if the string can be segmented

result = wordBreak(s, wordDict)

print(result)

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

The string can be segmented as "i like".

Input: ilikesamsung

CODE:

def wordBreak(s, wordDict):

# Create a DP array to store whether the string up to index i can be segmented

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

dp[0] = True # An empty string can always be segmented

# Iterate through the string

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

for word in wordDict:

# Check if the current substring matches a word in the dictionary

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

dp[i] = True

break # No need to check other words if we already found a match

return dp[-1] # Return whether the entire string can be segmented

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

s1 = "ilike"

s2 = "ilikesamsung"

print("Yes" if wordBreak(s1, wordDict) else "No")

print("Yes" if wordBreak(s2, wordDict) else "No")

Output: Yes

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

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

CODE:

def fullJustify(words, maxWidth):

result = []

line, line\_length = [], 0

# Iterate through the words and group them into lines

for word in words:

# Check if adding the current word exceeds the max width

if line\_length + len(word) + len(line) > maxWidth:

# Distribute spaces for the current line

for i in range(maxWidth - line\_length):

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

result.append(''.join(line))

line, line\_length = [], 0 # Reset for the next line

# Add the word to the current line

line.append(word)

line\_length += len(word)

# Handle the last line (left-justified)

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

return result

# Example usage

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

maxWidth = 16

justified\_text = fullJustify(words, maxWidth)

# Print the output

for line in justified\_text:

print(f'"{line}"')

Output:

[ "This is an",

"example of text",

"justification. "]

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"]]

CODE:

class WordFilter:

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

self.prefix\_suffix\_map = {}

# Preprocess words by storing all possible prefix and suffix combinations

for index, word in enumerate(words):

n = len(word)

for i in range(n + 1): # for prefix (from empty string to full word)

for j in range(n + 1): # for suffix (from empty string to full word)

prefix\_suffix = (word[:i], word[j:])

# Store the index in the map for the prefix and suffix combination

self.prefix\_suffix\_map[prefix\_suffix] = index

def f(self, pref, suff):

# Look up the dictionary for the largest index with given prefix and suffix

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

wordFilter = WordFilter(["apple"])

print(wordFilter.f("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

CODE:

def uniquePaths(m, n):

# Create a 2D DP table with m rows and n columns, initialized to 1

dp = [[1] \* n for \_ in range(m)]

# Fill the DP table

for i in range(1, m):

for j in range(1, n):

# The number of ways to reach dp[i][j] is the sum of ways

# to reach the cell above (dp[i-1][j]) and the cell to the left (dp[i][j-1])

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

# The bottom-right corner contains the answer

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

# Example usage

m = 3

n = 7

print(uniquePaths(m, n)) # Output: 28

Output: 28

Example 2:

Input: m = 3, n = 2

Output: 3

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]

CODE:

def numIdenticalPairs(nums):

count = 0

freq = {}

# Iterate through the array to count the frequency of each number

for num in nums:

# If the number already exists in the frequency map, it means all the

# previous occurrences of that number can form a good pair with the current one

if num in freq:

count += freq[num]

freq[num] += 1

else:

freq[num] = 1

return count

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

print(numIdenticalPairs(nums))

Output: 4

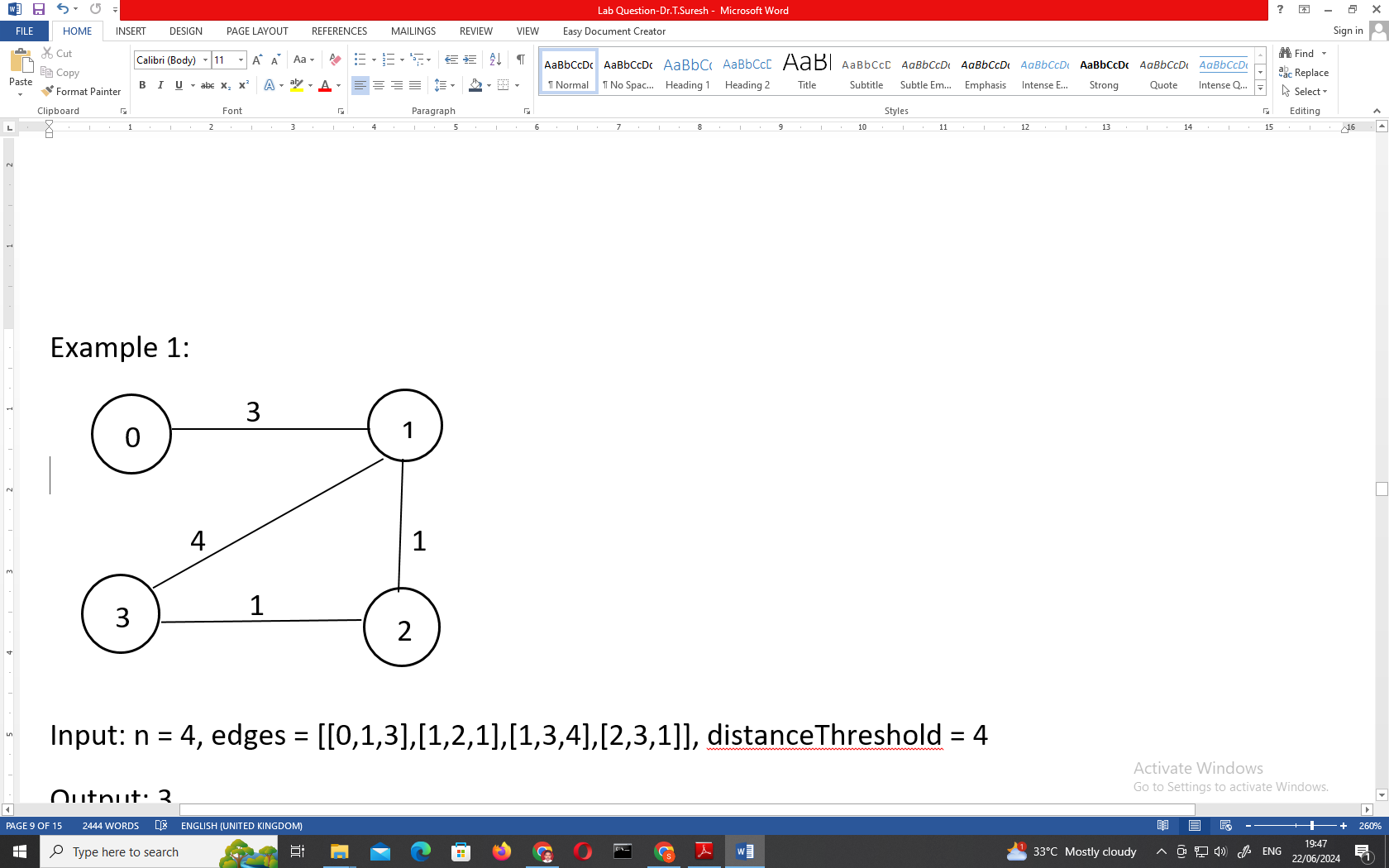
11. Warshall’s& Floyd’s Algorithm

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

CODE:

def findTheCity(n, edges, distanceThreshold):

# Initialize the distance matrix with infinity, except 0 for the diagonal

dist = [[float('inf')] \* n for \_ in range(n)]

# Set the diagonal to 0 since the distance to itself is 0

for i in range(n):

dist[i][i] = 0

# Populate the distance matrix with the given edges

for u, v, w in edges:

dist[u][v] = w

dist[v][u] = w

# Floyd-Warshall algorithm to find shortest paths between all pairs

for k in range(n):

for i in range(n):

for j in range(n):

dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

# Find the city with the smallest number of reachable cities within the distanceThreshold

min\_city = -1

min\_count = n # Initialize with max possible cities count

for i in range(n):

count = 0

for j in range(n):

if dist[i][j] <= distanceThreshold:

count += 1

# Update the city if this one has fewer reachable cities or same count but greater index

if count < min\_count or (count == min\_count and i > min\_city):

min\_city = i

min\_count = count

return min\_city

# Example usage:

n = 4

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

distanceThreshold = 4

print(findTheCity(n, edges, distanceThreshold))

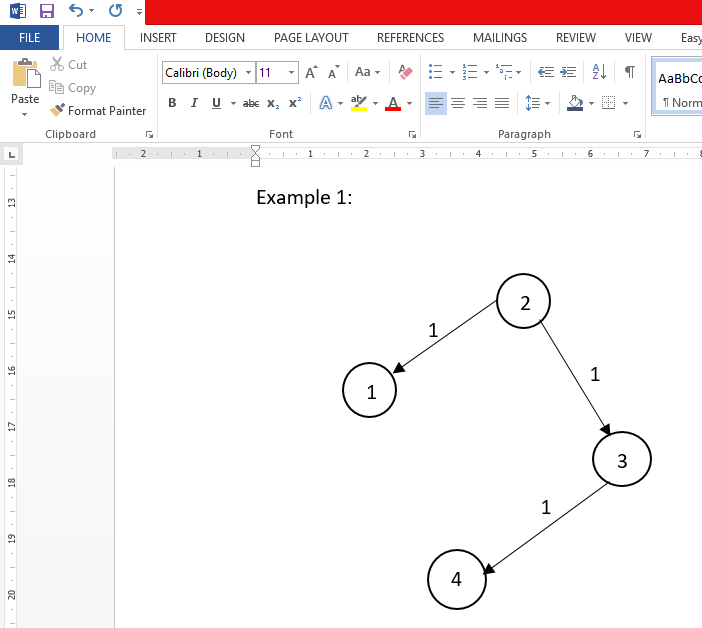
Output: 3

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

CODE:

import heapq

def networkDelayTime(times, n, k):

# Create adjacency list for the graph

graph = {i: [] for i in range(1, n+1)}

for u, v, w in times:

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

# Min heap to get the node with the smallest distance

min\_heap = [(0, k)] # (distance, node)

dist = {i: float('inf') for i in range(1, n+1)}

dist[k] = 0

while min\_heap:

d, node = heapq.heappop(min\_heap)

# If the current distance is larger than the recorded one, continue

if d > dist[node]:

continue

# Explore the neighbors

for neighbor, weight in graph[node]:

new\_dist = d + weight

if new\_dist < dist[neighbor]:

dist[neighbor] = new\_dist

heapq.heappush(min\_heap, (new\_dist, neighbor))

# The result will be the maximum distance in the dist dictionary

max\_dist = max(dist.values())

return max\_dist if max\_dist < float('inf') else -1

# Example usage:

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

n = 4

k = 2

print(networkDelayTime(times, n, k)) # Output: 2

Output: 2

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.

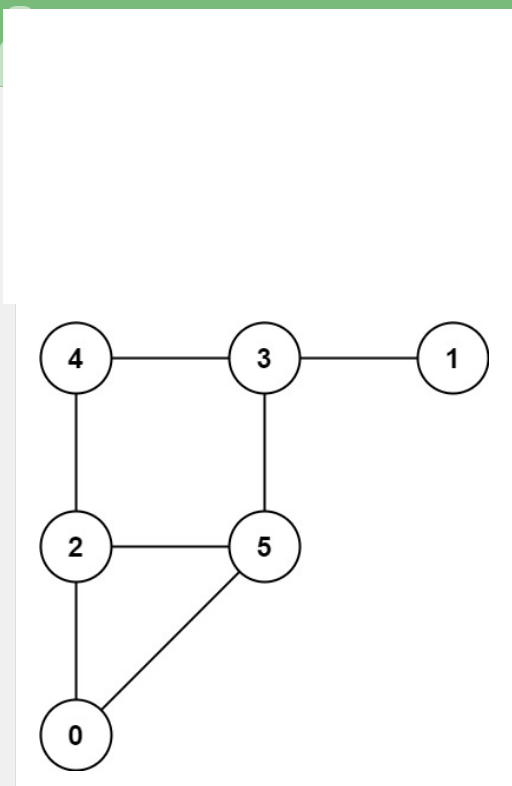
Given a graph, and assuming both players play optimally, return

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]]

CODE:

from collections import deque

def catMouseGame(graph):

n = len(graph)

# DP table to store game results: dp[mouse][cat][turn]

# 0: Draw, 1: Mouse wins, 2: Cat wins

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

# Queue for the BFS

queue = deque()

# Base cases:

for i in range(1, n):

dp[0][i][0] = 1 # Mouse reaches hole, mouse wins

dp[0][i][1] = 1 # Mouse reaches hole, mouse wins

dp[i][i][0] = 2 # Cat catches mouse, cat wins

dp[i][i][1] = 2 # Cat catches mouse, cat wins

queue.append((0, i, 0)) # Mouse wins when it reaches the hole

queue.append((0, i, 1)) # Mouse wins when it reaches the hole

queue.append((i, i, 0)) # Cat wins when they meet

queue.append((i, i, 1)) # Cat wins when they meet

# Helper to determine if the player has no valid moves

def has\_no\_moves(graph, pos, player\_turn):

if player\_turn == 0: # Mouse's turn

return len(graph[pos]) == 0

else: # Cat's turn

return len(graph[pos]) == 1 and 0 in graph[pos]

# Start BFS

while queue:

mouse, cat, turn = queue.popleft()

result = dp[mouse][cat][turn]

if turn == 0: # Mouse's turn, so it's the Cat's previous turn

for prev in graph[cat]:

if prev == 0:

continue # Cat can't go to the hole

if dp[mouse][prev][1] == 0:

if result == 2:

dp[mouse][prev][1] = 2

queue.append((mouse, prev, 1))

elif has\_no\_moves(graph, prev, 1):

dp[mouse][prev][1] = 1

queue.append((mouse, prev, 1))

else: # Cat's turn, so it's the Mouse's previous turn

for prev in graph[mouse]:

if dp[prev][cat][0] == 0:

if result == 1:

dp[prev][cat][0] = 1

queue.append((prev, cat, 0))

elif has\_no\_moves(graph, prev, 0):

dp[prev][cat][0] = 2

queue.append((prev, cat, 0))

return dp[1][2][0]

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

print(catMouseGame(graph))

Output: 0

14. Bellman-Ford Algorithm

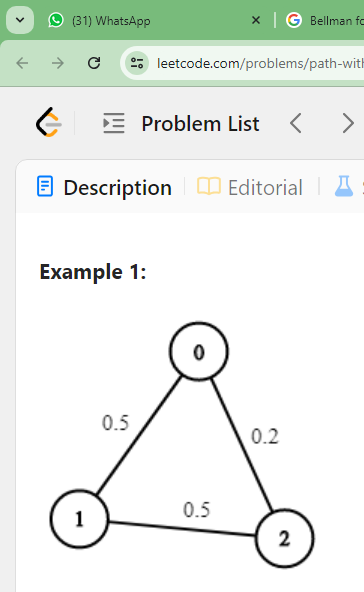
Path with Maximum Probability

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

CODE:

import heapq

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

# Create adjacency list to store graph

graph = {i: [] for i in range(n)}

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

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

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

# Max-heap to store the probabilities in descending order

max\_heap = [(-1, start)] # (-probability, node), we use -1 to signify 100% probability

prob = [0] \* n # Probability of reaching each node, initialized to 0

prob[start] = 1 # Start with probability 1 at the start node

while max\_heap:

current\_prob, node = heapq.heappop(max\_heap)

current\_prob = -current\_prob # Convert back to positive probability

# If we reached the end node, return the probability

if node == end:

return current\_prob

# Visit all neighbors

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

new\_prob = current\_prob \* edge\_prob

if new\_prob > prob[neighbor]:

prob[neighbor] = new\_prob

heapq.heappush(max\_heap, (-new\_prob, neighbor)) # Push negative for max-heap behavior

# If we never reach the end node, return 0

return 0

n = 3

edges = [[0, 1], [1, 2], [0, 2]]

succProb = [0.5, 0.5, 0.2]

start = 0

end = 2

print(maxProbability(n, edges, succProb, start, end)) # Output: 0.25000

Output: 0.25000

15.Write a Program to find both the maximum and minimum values in the array. Implement using any programming language of your choice. Execute your code and provide the maximum and minimum values found.

Input : N= 8, a[] = {5,7,3,4,9,12,6,2}

CODE:

def find\_min\_max(arr):

if len(arr) == 0:

return None, None # If the array is empty, return None for both

# Initialize min and max with the first element of the array

min\_val = arr[0]

max\_val = arr[0]

# Iterate through the array to find the minimum and maximum values

for num in arr:

if num < min\_val:

min\_val = num

if num > max\_val:

max\_val = num

return min\_val, max\_val

# Example Input:

arr = [5, 7, 3, 4, 9, 12, 6, 2]

# Function call to find min and max

min\_val, max\_val = find\_min\_max(arr)

# Output the results

print(f"Min = {min\_val}, Max = {max\_val}")

Output : Min = 2, Max = 12

16.Consider an array of integers sorted in ascending order: 2,4,6,8,10,12,14,18.

Write a Program to find both the maximum and minimum values in the array. Implement using any programming language of your choice. Execute your code and provide the maximum and minimum values found.

Input :N=8, 2,4,6,8,10,12,14,18.

CODE:

def find\_min\_max(arr):

if len(arr) == 0:

return None, None # If the array is empty, return None for both

# Since the array is sorted, min is the first element, and max is the last element

min\_val = arr[0]

max\_val = arr[-1]

return min\_val, max\_val

arr = [2, 4, 6, 8, 10, 12, 14, 18]

# Function call to find min and max

min\_val, max\_val = find\_min\_max(arr)

# Output the results

print(f"Min = {min\_val}, Max = {max\_val}")

Output : Min = 2, Max =18

17.You are given an unsorted array 31,23,35,27,11,21,15,28. Write a program for Merge Sort and implement using any programming language of your choice.

Input :N= 8, a[] = {31,23,35,27,11,21,15,28}

CODE:

# Function to merge two halves

def merge(left, right):

result = []

i = 0

j = 0

# Traverse both lists and append smaller element from either list to the result

while i < len(left) and j < len(right):

if left[i] < right[j]:

result.append(left[i])

i += 1

else:

result.append(right[j])

j += 1

# Collect the remaining elements from both halves

result.extend(left[i:])

result.extend(right[j:])

return result

# Function to perform merge sort

def merge\_sort(arr):

# Base case: if the array is of length 0 or 1, it's already sorted

if len(arr) <= 1:

return arr

# Find the middle point and split the array into two halves

mid = len(arr) // 2

left\_half = merge\_sort(arr[:mid])

right\_half = merge\_sort(arr[mid:])

# Merge the sorted halves

return merge(left\_half, right\_half)

arr = [31, 23, 35, 27, 11, 21, 15, 28]

# Function call to sort the array using merge sort

sorted\_arr = merge\_sort(arr)

print(sorted\_arr)

Output :11,15,21,23,27,28,31,35

18.Implement the Merge Sort algorithm in a programming language of your choice and test it on the array 12,4,78,23,45,67,89,1. Modify your implementation to count the number of comparisons made during the sorting process. Print this count along with the sorted array.

Input : N= 8, a[] = {12,4,78,23,45,67,89,1}

CODE:

# Global variable to keep track of comparisons

comparison\_count = 0

# Function to merge two halves and count comparisons

def merge(left, right):

global comparison\_count

result = []

i = 0

j = 0

# Traverse both lists and append smaller element from either list to the result

while i < len(left) and j < len(right):

comparison\_count += 1 # Each comparison

if left[i] < right[j]:

result.append(left[i])

i += 1

else:

result.append(right[j])

j += 1

# Collect the remaining elements from both halves

result.extend(left[i:])

result.extend(right[j:])

return result

# Function to perform merge sort and count comparisons

def merge\_sort(arr):

# Base case: if the array is of length 0 or 1, it's already sorted

if len(arr) <= 1:

return arr

# Find the middle point and split the array into two halves

mid = len(arr) // 2

left\_half = merge\_sort(arr[:mid])

right\_half = merge\_sort(arr[mid:])

# Merge the sorted halves

return merge(left\_half, right\_half)

# Example Input:

arr = [12, 4, 78, 23, 45, 67, 89, 1]

# Reset comparison count before sorting

comparison\_count = 0

# Function call to sort the array using merge sort

sorted\_arr = merge\_sort(arr)

print(f"Sorted Array: {sorted\_arr}")

print(f"Number of comparisons: {comparison\_count}")

Output :1,4,12,23,45,67,78,89