Uninformed bfs

```
from collections import deque
# Function to perform Breadth First Search on a graph
# represented using adjacency list
def bfs(adjList, startNode, visited):
        # Create a queue for BFS
        q = deque()
        # Mark the current node as visited and enqueue it
        visited[startNode] = True
    q.append(startNode)
        # Iterate over the queue
        while q:
        # Dequeue a vertex from queue and print it
        currentNode = q.popleft()
       print(currentNode, end=" ")
        # Get all adjacent vertices of the dequeued vertex
        # If an adjacent has not been visited, then mark it visited and enqueue it
        for neighbor in adjList[currentNode]:
        if not visited[neighbor]:
            visited[neighbor] = True
            q.append(neighbor)
# Function to add an edge to the graph
```

```
def addEdge(adjList, u, v):
    adjList[u].append(v)
def main():
        # Number of vertices in the graph
        vertices = 5
        # Adjacency list representation of the graph
        adjList = [[] for _ in range(vertices)]
        # Add edges to the graph
        addEdge(adjList, 0, 1)
        addEdge(adjList, 0, 2)
        addEdge(adjList, 1, 3)
        addEdge(adjList, 1, 4)
        addEdge(adjList, 2, 4)
        # Mark all the vertices as not visited
        visited = [False] * vertices
        # Perform BFS traversal starting from vertex 0
    print("Breadth First Traversal starting from vertex 0:", end=" ")
        bfs(adjList, 0, visited)
        print('\n')
if __name__ == "__main__":
        main()
```

Informed best first search

```
from queue import PriorityQueue
v = 14
graph = [[] for i in range(v)]
# Function For Implementing Best First Search
# Gives output path having lowest cost
def best_first_search(actual_Src, target, n):
	visited = [False] * n
	pq = PriorityQueue()
	pq.put((0, actual_Src))
	visited[actual_Src] = True
	while pq.empty() == False:
		u = pq.get()[1]
		# Displaying the path having lowest cost
		print(u, end=" ")
		if u == target:
			break
		for v, c in graph[u]:
			if visited[v] == False:
				visited[v] = True
				pq.put((c, v))
	print()
# Function for adding edges to graph
```

```
def addedge(x, y, cost):
        graph[x].append((y, cost))
        graph[y].append((x, cost))



# The nodes shown in above example(by alphabets) are
# implemented using integers addedge(x,y,cost);
addedge(0, 1, 3)
addedge(0, 2, 6)
addedge(0, 3, 5)
addedge(1, 4, 9)
addedge(1, 5, 8)
addedge(2, 6, 12)
addedge(2, 7, 14)
addedge(3, 8, 7)
addedge(8, 9, 5)
addedge(8, 10, 6)
addedge(9, 11, 1)
addedge(9, 12, 10)
addedge(9, 13, 2)
source = 0
target = 9
best_first_search(source, target, v)
```

Local search hill climbing algo

```
def findLocalMaxima(n, arr):
        mx = []

        # Check the first element
        if arr[0] > arr[1]:
        mx.append(0)

        # Check the elements in the middle
        for i in range(1, n - 1):
        if arr[i - 1] < arr[i] > arr[i + 1]:
        mx.append(i)

        # Check the last element
        if arr[-1] > arr[-2]:
        mx.append(n - 1)

        # Print the results
        if len(mx) > 0:
        print("Points of Local maxima are: ", end="")
        print(*mx)
        else:
        print("There are no points of Local maxima.")
```

```
# Main function

if __name__ == "__main__":

        n = 9

        arr = [10, 10, 15, 14, 13, 25, 5, 4, 3]

        findLocalMaxima(n, arr)
```

Aim :- Identify suitable Agent Architecture for the problem
Problem Statement :- Identify Agent architecture for Virtual assistants.

Pseudo Code :-

```
Initialize VirtualAssistant:

    Load profiles, NLP engine, task system



    Function HandleUserRequest(user_id,
       request): profile =
       GetUserProfile(user_id)
    goal = IdentifyGoal(ParseRequest(request))


    If IsKnownTask(goal):

       response = ExecuteTask(goal,
    profile) Else:
       response = LearnAndHandle(goal)


    UpdateUserProfile(user_id, request,
    response) return response


    Function
       ParseRequest(request
       ): return
       NLPParser(request)
```

```
    Function
        IsKnownTask(goal):
        return
        CheckTaskDatabase(goal
        )


Function ExecuteTask(goal, profile):

    Return TaskExecutor(goal, profile) # Handles specific tasks like setting reminders



    Function
        LearnAndHandle(goal):
        return
        LearnFromInteraction(goal
        )


    Function UpdateUserProfile(user_id, request,
        response): ModifyUserProfile(user_id,
        request, response)


# Main loop
For each user_id in ActiveUsers:
    response = HandleUserRequest(user_id,
    GetUserRequest(user_id)) SendResponseToUser(user_id,
    response)
```