```python
#1a
#GOLD MINE PROBLEM or MAX PATH FINDER

directions = [(0,1),(1,1),(-1,1)] # right , right up diagonal , right down diagonal

def isValid(matrix,i,j):
    return 0 <= i <len(matrix) and 0 <= j < len(matrix[0])

def moveNext(index):
    result_dict = {}
    for di,dj in directions:
        ni,nj = di+index[0] , dj+index[1]
        if isValid(matrix,ni,nj):
            result_dict[matrix[ni][nj]] = [ni,nj]

    max_value = max(result_dict)
    return result_dict[max_value]


def find_max_path(matrix):
    path_cost = 0
    max_value = float('-inf')
    index = []
    path_index = []
    for j in range(1):
        for i in range(len(matrix[0])):
           if max_value < matrix[i][j]:
               max_value = matrix[i][j]
               index=[i,j]

    path_cost += max_value
    path_index += [index]
    print(matrix[index[0]][index[1]],end="->")

    for i in range(len(matrix[0])-1):
        index = moveNext(index)
        path_cost += matrix[index[0]][index[1]]
        path_index += [index]
        print(matrix[index[0]][index[1]],end="->")


    return path_cost,path_index


matrix = [
    [2,5,9],
    [4,8,7],
    [3,5,6]
]

# n = int(input("Enter the n : "))
# matrix = [[int(input(f"Enter the element of matrix[{i}][{j}]: ")) for j in range(n)] for i in range(n)]

maxPath ,path_index = find_max_path(matrix)
print()
print(f"Maximum Path = {maxPath}")
print(f"Path Index = {path_index}")



```

```
4->8->9->
Maximum Path = 21
```

```
Path Index = [[1, 0], [1, 1], [0, 2]]
```

```python
#1b
#All path finder from a start node to goal node


def get_graph():
    graph = {}
    n = int(input("Enter the number of Nodes: "))
    for i in range(n):
        node = input("Enter the node name: ")
        neighbors = input(f"Enter the neighbors of {node} (comma-separated) : ").split(',')
        graph[node] = neighbors

    return graph

def find_all_paths(graph,start,goal,path=None):
    if path is None:
        path = []
    path = path + [start]

    if start not in graph:
        return []

    if start == goal:
        return [path]

    paths =[]

    for neighbor in graph[start]:
        if neighbor not in path:
            new_paths = find_all_paths(graph,neighbor,goal,path)
            for p in new_paths:
                paths.append(p)
    return paths


graph = get_graph()
print(graph)

start = input("Enter the Start Node: ")
goal = input("Enter the Goal Node: ")

all_paths = find_all_paths(graph,start,goal)
print(f"All Possible Paths from {start} to {goal} is ")
for path in all_paths:
    print(" -> ".join(path))

```

```
Enter the number of Nodes: 5
Enter the node name: A
Enter the neighbors of A (comma-separated) : B,C
Enter the node name: B
Enter the neighbors of B (comma-separated) : D,A
Enter the node name: C
Enter the neighbors of C (comma-separated) : A,D,E
Enter the node name: D
Enter the neighbors of D (comma-separated) : B,C,E
Enter the node name: E
Enter the neighbors of E (comma-separated) : C,D
{'A': ['B', 'C'], 'B': ['D', 'A'], 'C': ['A', 'D', 'E'], 'D': ['B', 'C', 'E'], 'E': ['C', 'D']}
Enter the Start Node: A
Enter the Goal Node: D
All Possible Paths from A to D is
A -> B -> D
A -> C -> D
A -> C -> E -> D
```

```
#2a
#Magic Sqaure
#MAGIC SQUARE

def magic_square(matrix):
    n = len(matrix)
    magic_sum = n*(n**2+1)//2
    row_sums = [0] * n
    col_sums = [0] * n
    diag_sum = 0
    diag2_sum = 0

    for i in range(n):
        for j in range(n):
            row_sums[i] += matrix[i][j]
            col_sums[j] += matrix[i][j]
            if i == j:
                diag_sum += matrix[i][j]
            if i+j == n-1:
                diag2_sum += matrix[i][j]

    if diag_sum != magic_sum or diag2_sum != magic_sum:
        return False

    for i in range(n):
        if row_sums[i] != magic_sum or col_sums[i] != magic_sum:
            return False
    return True


# n = int(input("Enter the n : "))
# matrix = [[int(input(f"Enter the element of matrix[{i}][{j}]: ")) for j in range(n)] for i in range(n)]


matrix = [[2,7,6],[9,5,1],[4,3,8]]

if magic_square(matrix):
    print("Yes, it is a Magic Square")
else:
    print("No, it is not a Magic Square")
```

```
Yes, it is a Magic Square
```

```
#2b
#DFS TRAVERSAL
#get graph from user

def get_graph():
    graph = {}
    n = int(input("Enter the number of Nodes: "))
    for i in range(n):
        node = input("Enter the node name: ")
        neighbors = input(f"Enter the neighbors of {node} (comma-separated) : ").split(',')
        graph[node] = neighbors

    return graph


graph = get_graph()
print(graph)
```

```
Enter the number of Nodes: 10
Enter the node name: 4
Enter the neighbors of 4 (comma-separated) : 3,2,10
Enter the node name: 3
Enter the neighbors of 3 (comma-separated) : 4,7,6
Enter the node name: 2
Enter the neighbors of 2 (comma-separated) : 4,5,1
Enter the node name: 10
Enter the neighbors of 10 (comma-separated) : 4,8,2
Enter the node name: 7
Enter the neighbors of 7 (comma-separated) : 3
Enter the node name: 6
Enter the neighbors of 6 (comma-separated) : 3
Enter the node name: 5
Enter the neighbors of 5 (comma-separated) : 2
Enter the node name: 1
Enter the neighbors of 1 (comma-separated) : 2
Enter the node name: 8
Enter the neighbors of 8 (comma-separated) : 10
Enter the node name: 2
Enter the neighbors of 2 (comma-separated) : 10
{'4': ['3', '2', '10'], '3': ['4', '7', '6'], '2': ['10'], '10': ['4', '8', '2'], '7': ['3'], '6': ['3'], '5': ['2'], '1': ['2'], '8': [
```

```python

def DFS(graph,start):
    visited = set()
    stack = [start]
    sum = 0
    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            print(f"{node}",end="->")
            if int(node) & 1 :
              sum += 1
            else:
              sum +=2
            stack.extend(reversed(graph.get(node,[])))
    return sum

start_node = input("Enter the starting node: ")
sum = DFS(graph, start_node)
print()
print(f"The summation of the Travel path is {sum}")

```

```
Enter the starting node: 4
4->3->7->6->2->10->8->The summation of the Travel path is 12
```

```python
#3a
def rotate_left(arr, d):
    # The number of rotations should be within the length of the array
    d = d % len(arr)
    # Perform the rotation by slicing the array
    return arr[d:] + arr[:d]

# Test input
arr = [23, 4, 56, 72, 98, 12]
rotated_arr = rotate_left(arr, 2)

print("Original Array:", arr)
print("Array after 2 rotations to the left:", rotated_arr)

```

```python
#3b
def mice_and_holes(mice_positions, hole_positions):
    # Sort both mice and hole positions to minimize time
```

```
    mice_positions.sort()
    hole_positions.sort()

    # Calculate the time for each mouse
    times = []
    for i in range(len(mice_positions)):
        time_taken = abs(mice_positions[i] - hole_positions[i])
        times.append(time_taken)
        print(f"time taken by {i} th mouse is: {time_taken}")

    # Find the maximum time taken
    max_time = max(times)
    print(f"Maximum time taken is: {max_time}")

# Test input for Mice and Holes problem
mice_positions = [-23, -14, 9, -45, -10]
hole_positions = [3, 4, 5, 6, 7]

mice_and_holes(mice_positions, hole_positions)
```

```
#4a
#Kronocker product

rowa = int(input("Enter the row_a: "))
cola = int(input("Enter the column_a"))

matrix1 = [[int(input(f"Enter the element of matrix1[{i}][{j}]: ")) for j in range(cola)] for i in range(r

rowb = int(input("Enter the row_b: "))
colb = int(input("Enter the column_b"))

matrix2 = [[int(input(f"Enter the element of matrix2[{i}][{j}]: ")) for j in range(colb)] for i in range(r

result = [[0 for j in range(cola*colb)] for i in range(rowa*rowb)]

for i in range(rowa):
    for j in range(rowb):
        for k in range(cola):
            for l in range(colb):
                result[i*rowb+j][k*colb+l] = matrix1[i][k] * matrix2[j][l]

print("Resultant Matrix:- ")
for row in result:
    for ele in row:
        print(ele,end=" ")
    print()
```

```
#4b
from collections import deque

def bfs_multiple_goals(graph, start, goals):
    # A queue for BFS, which stores tuples of the current node and the path
    queue = deque([(start, [start])])
    visited = set([start])  # Keep track of visited nodes

    while queue:
        current_node, path = queue.popleft()

        # If the current node is one of the goal nodes, return the path
        if current_node in goals:
            print(f"Goal {current_node} found!")
            return path

```

```
        # Add neighbors to the queue
        for neighbor in graph[current_node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, path + [neighbor]))

    return None  # No path found to any of the goal nodes

# Example graph: adjacency list representation
graph = {
    1: [2, 3],
    2: [1, 4, 5],
    3: [1, 6],
    4: [2],
    5: [2, 7],
    6: [3],
    7: [5]
}

start_node = 1
goal_nodes = {5, 6}  # More than one goal node (5 and 6)

# Run BFS to find a path to any goal node
path = bfs_multiple_goals(graph, start_node, goal_nodes)

if path:
    print(f"Path to a goal node: {path}")
else:
    print("No path found to any goal node.")

```

```
#5a
def swap_diagonal_elements(matrix, n):
    diagonal_sum = 0
    # Loop through the matrix to swap diagonal elements and calculate the sum
    for i in range(n):
        # Calculate the sum of diagonal elements
        diagonal_sum += matrix[i][i]

        # Swap main diagonal with anti-diagonal
        matrix[i][i], matrix[i][n-i-1] = matrix[i][n-i-1], matrix[i][i]

    # Print the modified matrix
    print("Matrix after swapping diagonal elements:")
    for row in matrix:
        print(row)

    # Print the sum of diagonal elements
    print(f"Sum of the diagonal elements: {diagonal_sum}")

# Input for the matrix size
n = int(input("Enter the size of the matrix (n x n): "))

# Input the matrix elements
matrix = []
print(f"Enter the elements of the {n}x{n} matrix row by row:")
for i in range(n):
    row = list(map(int, input().split()))
    matrix.append(row)

# Call the function to swap diagonal elements and print the result
swap_diagonal_elements(matrix, n)

```

```
#5b
import heapq

def ucs(graph, start, goal):
    # Priority queue (min-heap) to store nodes along with their accumulated costs
    frontier = []
    heapq.heappush(frontier, (0, start))  # Push the start node with cost 0

    # A dictionary to store the parent of each node for path reconstruction
    came_from = {start: None}

    # A dictionary to store the cost of reaching each node
    cost_so_far = {start: 0}

    while frontier:
        # Pop the node with the smallest accumulated cost
        current_cost, current_node = heapq.heappop(frontier)

        # If we've reached the goal, reconstruct the path
        if current_node == goal:
            path = []
            while current_node is not None:
                path.append(current_node)
                current_node = came_from[current_node]
            path.reverse()
            return path, cost_so_far[goal]

        # Explore neighbors of the current node
        for neighbor, weight in graph[current_node]:
            new_cost = current_cost + weight
            if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                cost_so_far[neighbor] = new_cost
                heapq.heappush(frontier, (new_cost, neighbor))
                came_from[neighbor] = current_node

    return None, None  # Return None if no path exists

# Example graph (adjacency list representation)
# Format: node -> [(neighbor1, cost1), (neighbor2, cost2), ...]
graph = {
    'A': [('B', 1), ('C', 4)],
    'B': [('A', 1), ('C', 2), ('D', 5)],
    'C': [('A', 4), ('B', 2), ('D', 1)],
    'D': [('B', 5), ('C', 1)],
}

# Start and Goal
start_node = 'A'
goal_node = 'D'

# Run UCS to find the path and the total cost
path, total_cost = ucs(graph, start_node, goal_node)

if path:
    print(f"Solution Path: {' -> '.join(path)}")
    print(f"Total Cost: {total_cost}")
else:
    print("No path found.")
```