1)
```
trie = {}

while True:
    print("1. Insert word")
    print("2. Get autocomplete suggestions")
    print("3. Exit")
    choice = input("Enter your choice: ")

    if choice == '1':
        word = input("Enter the word to insert: ").strip()
        node = trie
        for char in word:
            if char not in node:
                node[char] = {}  # Add a new node for each character
            node = node[char]
        node['#'] = True  # Mark the end of the word
        print(f"Inserted '{word}' into the trie.\n")

    elif choice == '2':
        prefix = input("Enter the prefix to search: ").strip()
        node = trie
        found = True

        # Traverse the trie to find the prefix
        for char in prefix:
            if char not in node:
                found = False
                break
            node = node[char]

        if not found:
            print(f"No suggestions found for '{prefix}'.\n")
        else:
            # Gather all words starting with the prefix
            words = []
            stack = [(node, prefix)]
            while stack:
                current_node, current_prefix = stack.pop()
                if '#' in current_node:  # End of word
                    words.append(current_prefix)
                for char, next_node in current_node.items():
                    if char != '#':
                        stack.append((next_node, current_prefix + char))

            if words:
                print(f"Autocomplete suggestions for '{prefix}': {words}\n")
            else:
                print(f"No suggestions found for '{prefix}'.\n")

    elif choice == '3':
        print("Exiting...")
        break

    else:
        print("Invalid choice. Please choose again.\n")
```

2)
```
city = {
    '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 = 'A'
end = 'D'

unvisited = list(city.keys())
shortest_distance = {}
previous_place = {}

for place in city:
    shortest_distance[place] = float('inf')
shortest_distance[start] = 0

while unvisited:
    current = None
    for place in unvisited:
        if current is None or shortest_distance[place] <
shortest_distance[current]:
            current = place

    for neighbor in city[current]:
        new_distance = shortest_distance[current] + city[current][neighbor]
        if new_distance < shortest_distance[neighbor]:
            shortest_distance[neighbor] = new_distance
            previous_place[neighbor] = current

    unvisited.remove(current)

path = []
current = end
while current != start:
    path.insert(0, current)
    current = previous_place[current]
path.insert(0, start)

print("Shortest path:", path)
print("Total travel time:", shortest_distance[end])
```

3)
```
print("Enter the names of the cities (separated by spaces):")
cities = input().split()

print("Enter the number of connections:")
num_connections = int(input())

connections = []
for _ in range(num_connections):
    print("Enter connection (city1 city2 cost):")
    city1, city2, cost = input().split()
    connections.append((city1, city2, int(cost)))

parent = {city: city for city in cities}

def find(city):
    while parent[city] != city:
        parent[city] = parent[parent[city]]
        city = parent[city]
    return city

mst = []

sorted_connections = sorted(connections, key=lambda x: x[2])

for city1, city2, cost in sorted_connections:
    root1 = find(city1)
    root2 = find(city2)

    if root1 != root2:
        parent[root2] = root1
        mst.append((city1, city2, cost))

print("\nThe roads included in the Minimum Spanning Tree (MST) are:")
for city1, city2, cost in mst:
    print(f"{city1} - {city2}: {cost}")
```

4)

```
board = [
    ['r', 'n', 'b', 'q', 'k', 'b', 'n', 'r'],
    ['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    ['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'],
    ['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
]
white_turn = True

def print_board():
    print('\n'.join(' '.join(row) for row in board))

def get_pawn_moves(row, col):
    moves = []
    direction = -1 if board[row][col].isupper() else 1
    if board[row + direction][col] == ' ':
        moves.append((row + direction, col))
    if row in [1, 6] and board[row + 2 * direction][col] == ' ':
        moves.append((row + 2 * direction, col))
    for dcol in [-1, 1]:
        new_row, new_col = row + direction, col + dcol
        if 0 <= new_col < 8 and board[new_row][new_col] not in [' ', board[row]
[col].upper()]:
            moves.append((new_row, new_col))
    return moves

while True:
    print_board()
    print(f"{'White' if white_turn else 'Black'}'s turn")
    source, dest = input("Enter move (e.g. a2 a4): ").split()
    src_row, src_col = 8 - int(source[1]), ord(source[0]) - ord('a')
    dest_row, dest_col = 8 - int(dest[1]), ord(dest[0]) - ord('a')

    if (dest_row, dest_col) in get_pawn_moves(src_row, src_col):
        board[dest_row][dest_col], board[src_row][src_col] = board[src_row]
[src_col], ' '
        white_turn = not white_turn
    else:
        print("Invalid move.")
```

5)

```
arr = list(map(int, input("Enter the array elements: ").split()))
n = len(arr)
tree = [0] * (2 * n)

# Build the segment tree
tree[n:] = arr  # Copy array to leaves
for i in range(n - 1, 0, -1):  # Build from bottom up
    tree[i] = tree[2 * i] + tree[2 * i + 1]
```

```
print("Segment Tree:", tree)

# --- Range Sum Query ---
l, r = map(int, input("Enter range (start end): ").split())
l += n  # Move to leaf level
r += n
s = 0
while l <= r:
    if l % 2 == 1:
        s += tree[l]
        l += 1
    if r % 2 == 0:
        s += tree[r]
        r -= 1
    l //= 2
    r //= 2
print("Sum in range:", s)

# --- Point Update ---
i, v = map(int, input("Enter index and new value: ").split())
i += n
tree[i] = v
while i > 1:
    tree[i // 2] = tree[i] + tree[i ^ 1]
    i //= 2

print("Updated Segment Tree:", tree)

# --- Another Range Sum Query after update ---
l, r = map(int, input("Enter range (start end): ").split())
l += n
r += n
s = 0
while l <= r:
    if l % 2 == 1:
        s += tree[l]
        l += 1
    if r % 2 == 0:
        s += tree[r]
        r -= 1
    l //= 2
    r //= 2
print("Updated Sum in range:", s)
```

6)

```
def create_node(x1, y1, x2, y2):
    return {"boundary": (x1, y1, x2, y2), "city": None, "children": []}

def in_boundary(b, x, y):
    return b[0] <= x <= b[2] and b[1] <= y <= b[3]

def insert(node, name, x, y):
    if not in_boundary(node["boundary"], x, y):
        return False
    if node["city"] is None and not node["children"]:
        node["city"] = (name, x, y)
        return True
    if not node["children"]:
        x1, y1, x2, y2 = node["boundary"]
        mx = (x1 + x2) / 2
        my = (y1 + y2) / 2
        node["children"] = [
            create_node(mx, my, x2, y2),     # NE
            create_node(x1, my, mx, y2),     # NW
            create_node(x1, y1, mx, my),     # SW
            create_node(mx, y1, x2, my)      # SE
        ]
        if node["city"]:
            cname, cx, cy = node["city"]
            node["city"] = None
            insert(node, cname, cx, cy)
    for child in node["children"]:
        if insert(child, name, x, y):
            return True
    return False

def search(node, x, y):
    if node["city"] and node["city"][1] == x and node["city"][2] == y:
        return node["city"][0]
    for child in node["children"]:
        if in_boundary(child["boundary"], x, y):
            return search(child, x, y)
    return None

# MAIN
root = create_node(0, 0, 100, 100)
n = int(input("Number of cities: "))
for _ in range(n):
    name = input("City name: ")
    x = int(input("x (0â “100): "))
    y = int(input("y (0â “100): "))
    insert(root, name, x, y)
sx = int(input("Search x: "))
sy = int(input("Search y: "))
result = search(root, sx, sy)
print("City found:" if result else "City not found.", result if result else "")
```