Code:

1. Selection Sort

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

arr = [64, 25, 12, 22, 11]
selection_sort(arr)
print("Selection Sort Result:", arr)
```

Output:

```
Selection Sort Result: [11, 12, 22, 25, 64]
```

2. **Prim's Algorithm**

```
import heapq
def prim_mst(graph):
    mst = []
    visited = set()
    start_node = list(graph.keys())[0]
    visited.add(start_node)
    edges = [(cost, start_node, neighbor) for neighbor, cost in graph[start_node]]
    heapq.heapify(edges)
    while edges:
        cost, u, v = heapq.heappop(edges)
        if v not in visited:
            visited.add(v)
            mst.append((u, v, cost))
            for neighbor, n_cost in graph[v]:
                if neighbor not in visited:
                    heapq.heappush(edges, (n_cost, v, neighbor))
    return mst
graph = {
    'A': [('B', 2), ('C', 3)],
    'B': [('A', 2), ('C', 4), ('D', 5)],
    'C': [('A', 3), ('B', 4), ('D', 1)],
    'D': [('B', 5), ('C', 1)],}
mst = prim_mst(graph)
print("Prim's Minimal Spanning Tree:", mst)
```

Output:

```
Prim's Minimal Spanning Tree: [('A', 'B', 2), ('A', 'C', 3), ('C', 'D', 1)]
```

3. **Kruskal's Algorithm:**

```
def kruskal_mst(graph):
    mst = []
    edges = []

    for node in graph:
        for neighbor, cost in graph[node]:
            edges.append((cost, node, neighbor))

    edges.sort()

    parent = {node: node for node in graph}

    def find(node):
        if parent[node] != node:
            parent[node] = find(parent[node])
        return parent[node]

    for cost, u, v in edges:
        if find(u) != find(v):
            mst.append((u, v, cost))
            parent[find(u)] = find(v)
    return mst
graph = {
    'A': [('B', 2), ('C', 3)],
    'B': [('A', 2), ('C', 4), ('D', 5)],
    'C': [('A', 3), ('B', 4), ('D', 1)],
    'D': [('B', 5), ('C', 1)],
}
mst = kruskal_mst(graph)
print("Kruskal's Minimal Spanning Tree:", mst)
```

Output:

```
Kruskal's Minimal Spanning Tree: [('C', 'D', 1), ('A', 'B', 2), ('A', 'C', 3)]
```

4. **Dijkstra's Algorithm:**

```
import heapq

def dijkstra(graph, start):
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    queue = [(0, start)]

    while queue:
        current_dist, node = heapq.heappop(queue)

        if current_dist > distances[node]:
            continue
```

```
        for neighbor, weight in graph[node]:
            distance = current_dist + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(queue, (distance, neighbor))

    return distances
graph = {
    'A': [('B', 2), ('C', 5)],
    'B': [('A', 2), ('C', 1), ('D', 7)],
    'C': [('A', 5), ('B', 1), ('D', 3)],
    'D': [('B', 7), ('C', 3)],
}
start_node = 'A'
shortest_distances = dijkstra(graph, start_node)
print("Dijkstra's Shortest Distances from", start_node + ":", shortest_distances)
```

Output:

```
Dijkstra's Shortest Distances from A: {'A': 0, 'B': 2, 'C': 3, 'D': 6}
```

Time Complexity:

Selection Sort: O(n^2), where n is the number of elements in the array. This is because, in the worst case, it needs to compare and swap elements for every possible pair.

Prim's Minimal Spanning Tree Algorithm: O(E log V), where E is the number of edges and V is the number of vertices. This complexity is for the binary heap-based implementation of Prim's algorithm.

Kruskal's Minimal Spanning Tree Algorithm: O(E log E), where E is the number of edges. Sorting the edges dominates the time complexity.

Dijkstra's Minimal Spanning Tree Algorithm (Single-Source Shortest Path): O(V^2) for the naive implementation with an adjacency matrix or O((V + E) log V) using a binary heap-based priority queue.