1.Prim's Algorithm

```
void prims(int cost[10][10], int n) {
    int i, j;
    int u, v;
    int sum, k;
    int t[10][2];
    int p[10], d[10], s[10];
    int min = 999, source = 0;

    // Find the initial source vertex
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            if (cost[i][j] != 0 && cost[i][j] < min) {
                min = cost[i][j];
                source = i;
            }
        }
    }

    // Initialize arrays
    for (i = 0; i < n; i++) {
        d[i] = cost[source][i];
        s[i] = 0;
        p[i] = source;
    }
    s[source] = 1;
```

```
sum = 0;
k = 0;

// Prim's algorithm
for (i = 1; i < n; i++) {
    min = 999;
    u = -1;

    for (j = 0; j < n; j++) {
        if (s[j] == 0 && d[j] < min) {
            min = d[j];
            u = j;
        }
    }

    t[k][0] = u;
    t[k][1] = p[u];
    k++;
    sum += cost[u][p[u]];
    s[u] = 1;

    for (v = 0; v < n; v++) {
        if (s[v] == 0 && cost[u][v] < d[v]) {
            d[v] = cost[u][v];
            p[v] = u;
        }
```

```
        }
    }

    printf("\nWeighted minimum spanning tree\n");
    for (i = 1; i < n; i++) {  // start from 1 since the root doesn't have a parent
        printf("(%d,%d) -> weight: %d\n", p[i], i, cost[p[i]][i]);
    }
    printf("\nSum of minimum spanning tree: %d\n", sum);
}
```

Output:

```
Enter the number of vertices: 5
Enter the adjacency matrix:
0 2 0 6 0
2 0 3 8 5
0 3 0 0 7
6 8 0 0 9
0 5 7 9 0
Weighted minimum spanning tree
(0,1) -> weight: 2
(1,2) -> weight: 3
(0,3) -> weight: 6
(1,4) -> weight: 5

Sum of minimum spanning tree: 16
```

2.Kruskal's Algorithm

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 10

// Structure to represent an edge
typedef struct {
    int src, dest, weight;
} Edge;

// Structure to represent a subset for union-find
typedef struct {
    int parent, rank;
} Subset;

// Function to find the subset of an element
int find(Subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

// Function to perform union of two subsets
void Union(Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
```

```
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Function to compare two edges (used in qsort)
int compare(const void *a, const void *b) {
    Edge *edge1 = (Edge *)a;
    Edge *edge2 = (Edge *)b;
    return edge1->weight - edge2->weight;
}

// Function to implement Kruskal's algorithm
void KruskalMST(Edge edges[], int V, int E) {
    Edge result[MAX];
    int e = 0; // Count of edges in MST
    int i = 0; // Initial index of sorted edges

    // Sort edges in non-decreasing order of their weight
```

```
qsort(edges, E, sizeof(edges[0]), compare);

// Allocate memory for creating V subsets
Subset *subsets = (Subset *)malloc(V * sizeof(Subset));

// Create V subsets with single elements
for (int v = 0; v < V; ++v) {
    subsets[v].parent = v;
    subsets[v].rank = 0;
}

// Process each edge in sorted order
while (e < V - 1 && i < E) {
    Edge next_edge = edges[i++];

    int x = find(subsets, next_edge.src);
    int y = find(subsets, next_edge.dest);

    // If including this edge does not cause a cycle
    if (x != y) {
        result[e++] = next_edge;
        Union(subsets, x, y);
    }
}

// Print the MST
```

```
    printf("Following are the edges in the constructed MST\n");
    for (i = 0; i < e; ++i)
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);

    free(subsets);
}

int main() {
    int V, E;
    Edge edges[MAX];

    printf("Enter the number of vertices: ");
    scanf("%d", &V);
    printf("Enter the number of edges: ");
    scanf("%d", &E);

    printf("Enter the edges (src dest weight):\n");
    for (int i = 0; i < E; ++i) {
        scanf("%d %d %d", &edges[i].src, &edges[i].dest, &edges[i].weight);
    }

    KruskalMST(edges, V, E);

    return 0;
}
```

Output:

Enter the number of vertices: 4

Enter the number of edges: 5

Enter the edges (src dest weight):

0 1 10

0 2 6

0 3 5

1 3 15

2 3 4

Following are the edges in the constructed MST

2 -- 3 == 4

0 -- 3 == 5

0 -- 2 == 6

3.Dijkstra’s Algorithm

```
#include <stdio.h>
#include <limits.h>

#define MAX 10

int minDistance(int dist[], int sptSet[], int n) {
    int min = INT_MAX, min_index;

    for (int v = 0; v < n; v++)
        if (sptSet[v] == 0 && dist[v] < min)
            min = dist[v], min_index = v;
```

```
    return min_index;
}

void dijkstra(int graph[MAX][MAX], int src, int n) {
    int dist[MAX];
    int sptSet[MAX]; // Shortest Path Tree Set

    // Initialize distances and sptSet
    for (int i = 0; i < n; i++) {
        dist[i] = INT_MAX;
        sptSet[i] = 0;
    }

    // Distance from the source to itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < n - 1; count++) {
        int u = minDistance(dist, sptSet, n);

        sptSet[u] = 1;

        for (int v = 0; v < n; v++)
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
```

```
    }

    // Print the constructed distance array
    printf("Vertex \t Distance from Source\n");
    for (int i = 0; i < n; i++)
        printf("%d \t %d\n", i, dist[i]);
}

int main() {
    int n;
    int graph[MAX][MAX];

    printf("Enter the number of vertices: ");
    scanf("%d", &n);

    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    int src;
    printf("Enter the source vertex: ");
    scanf("%d", &src);
```

```
    dijkstra(graph, src, n);


    return 0;
}
```

Output:

```
Enter the number of vertices: 5
Enter the adjacency matrix:
0 10 0 30 100
10 0 50 0 0
0 50 0 20 10
30 0 20 0 60
100 0 10 60 0
Enter the source vertex: 0
Vertex  Distance from Source
0       0
1       10
2       60
3       30
4       70
```

4.Fractional Knapsack

```
#include <stdio.h>

void sortItemsByProfit(int weights[], int profits[], float ratios[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - 1 - i; j++) {
            if (profits[j] < profits[j + 1]) {
                int tempWeight = weights[j];
                int tempProfit = profits[j];
                float tempRatio = ratios[j];

                weights[j] = weights[j + 1];
                profits[j] = profits[j + 1];
                ratios[j] = ratios[j + 1];

                weights[j + 1] = tempWeight;
                profits[j + 1] = tempProfit;
                ratios[j + 1] = tempRatio;
            }
        }
    }
}

void sortItemsByWeight(int weights[], int profits[], float ratios[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - 1 - i; j++) {
```

```
            if (weights[j] > weights[j + 1]) {
                int tempWeight = weights[j];
                int tempProfit = profits[j];
                float tempRatio = ratios[j];

                weights[j] = weights[j + 1];
                profits[j] = profits[j + 1];
                ratios[j] = ratios[j + 1];

                weights[j + 1] = tempWeight;
                profits[j + 1] = tempProfit;
                ratios[j + 1] = tempRatio;
            }
        }
    }
}

void sortItemsByRatio(int weights[], int profits[], float ratios[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - 1 - i; j++) {
            if (ratios[j] < ratios[j + 1]) {
                int tempWeight = weights[j];
                int tempProfit = profits[j];
                float tempRatio = ratios[j];

                weights[j] = weights[j + 1];
```

```
            profits[j] = profits[j + 1];
            ratios[j] = ratios[j + 1];

            weights[j + 1] = tempWeight;
            profits[j + 1] = tempProfit;
            ratios[j + 1] = tempRatio;
        }
    }
  }
}

void fractionalKnapsack(int weights[], int profits[], int n, int capacity, int
criterion) {
  float ratios[n];
  for (int i = 0; i < n; i++) {
    ratios[i] = (float)profits[i] / weights[i];
  }

  switch (criterion) {
    case 1:
      sortItemsByProfit(weights, profits, ratios, n);
      break;
    case 2:
      sortItemsByWeight(weights, profits, ratios, n);
      break;
    case 3:
```

```
        sortItemsByRatio(weights, profits, ratios, n);
        break;
    default:
        printf("Invalid criterion\n");
        return;
}

int curWeight = 0;
float totalProfit = 0.0;

for (int i = 0; i < n; i++) {
    if (curWeight + weights[i] <= capacity) {
        curWeight += weights[i];
        totalProfit += profits[i];
    } else {
        int remaining = capacity - curWeight;
        totalProfit += profits[i] * ((float)remaining / weights[i]);
        break;
    }
}

switch (criterion) {
    case 1:
        printf("Maximum Profit: %.2f\n", totalProfit);
        break;
    case 2:
```

```
            printf("Maximum Profit with Minimum Weight: %.2f\n", totalProfit);
            break;
        case 3:
            printf("Maximum Profit with Maximum Ratio: %.2f\n", totalProfit);
            break;
    }
}

int main() {
    int n, capacity, criterion;
    printf("Enter the number of items: ");
    scanf("%d", &n);
    printf("Enter the capacity of the knapsack: ");
    scanf("%d", &capacity);

    int weights[n], profits[n];
    for (int i = 0; i < n; i++) {
        printf("Enter weight and profit for item %d: ", i + 1);
        scanf("%d %d", &weights[i], &profits[i]);
    }
    printf("Choose criterion (1: Max Profit, 2: Min Weight, 3: Max Profit/Weight
Ratio): ");
    scanf("%d", &criterion);
    fractionalKnapsack(weights, profits, n, capacity, criterion);
    return 0;
}
```