相关术语

图由顶点和边组成。

如果边都是无向边，则图为无向图，否则为有向图。

顶点的邻居的数量称为顶点的度。离开顶点的边条数称为顶点的出度，进入顶点的边条数称为顶点的入度。

边的长度称为权。

连通：两个顶点之间可以通过若干条边相互到达，若所有顶点都是连通的，则为连通图。

图的代码实现

邻接矩阵

```
int arr[4][4];
arr[0][1] = 1; // A->B之间的边权值为1
arr[1][0] = 2; // B->A之间的边权值为2
```

劣势：空间复杂度O(V^2)

邻接表法

```
struct Edge {
    int adjNode; // 邻接顶点
    int weight; // 边的权值
    Edge(int _adjNode, int _weight) {
        adjNode = _adjNode;
        weight = _weight;
    }
};
vector<vector<Edge>> graph(4);
Edge e(1, 1);
graph[0].push_back(e); // A->B之间的边权值为1
```

并查集

并查集通常用于检查图的连通性。

`find()`：根据元素查找对应的集合

`union()`：两个集合合并

```
int father[1000]; // 数组下标是集合数据编号father[i]集合数据的父亲的编号，根的父亲编号和根编号相同

void initUFSet(int n) {
    // 0 ~ n - 1
    for(int i = 0; i < n; i++) {
        father[i] = i;
    }
}

int find(int u) {
    while(father[u] != u) {
        u = father[u];
    }
    return u;
}

void union(int u, int v) {
    int uRoot = find(u);
    int vRoot = find(v);
    father[vRoot] = uRoot;
}
```

并查集优化

```
/*路径压缩*/
int find(int u) {
    while(father[u] != u) {
        father[u] = father[father[u]];
        u = father[u];
    }
    return u;
}
```

最小生成树

找到一个子图，需要连通所有点，并且边的权值之和要最小（树）。掌握Kruskal算法即可。

1. 将所有边按权值排序。
2. 按权值从小到大加入子图（可以使用堆实现，也可以先排序再遍历）。如果边的两点已经连通（用并查集实现），则无需加入。
3. 边数=顶点数-1。

单源最短路径

Dijkstra算法用于求一个顶点到所有其他顶点的最短路径长度。

可以使用小根堆，找到离起点最近且未被访问过的结点。

```
struct PQueueNode {
    int u;
    int distance;
    PQueueNode(_u, _distance) {
        u = _u;
        distance = _distance;
    }
};

bool operator < (PQueueNode lhs, PQueueNode rhs) {
    return lhs.distance > rhs.distance;
}

int dijkstra(int s, int t) {
    priority_queue<PQueueNode> pqueue;
    int distance[300];
    bool isVisited[300];
    for(int i = 0; i < 300; i++) {
        distance[i] = -1; // 表示无穷远
        isVisited[i] = false;
    }
    distance[s] = 0;
    PQueueNode qnode(s, 0);
    pqueue.push(qnode);
    while(!pqueue.empty()) {
```

```
        int u = pqueue.top().u;
        pqueue.pop();
        if(isVisited[u] == true) {
            continue;
        } else {
            isVisited[u] = true;
        }
        for(int i = 0; i < graph[u].size(); i++) {
            int v = graph[u][i].v;
            int weight = graph[u][i].weight;
            if(distance[v] == -1 || distance[v] > distance[u] + weight) {
                distance[v] = distance[u] + weight;
                PQueueNode next(v, distance[v]);
                pqueue.push(next);
            }
        }
    }

    return distance[t];
}
```