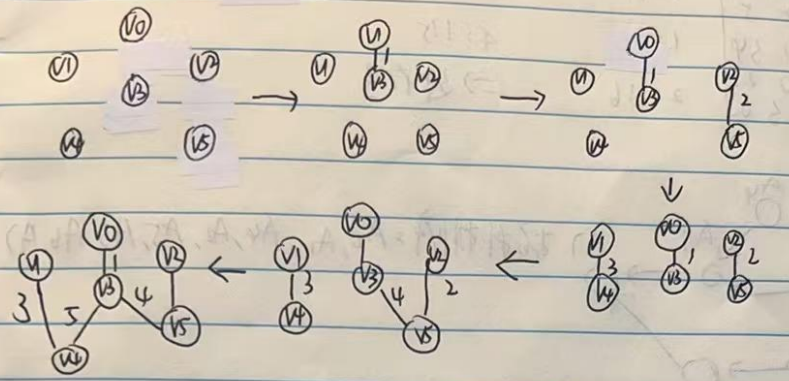
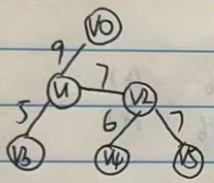
四.7 Kruskal



四.8 Prim



$$\begin{bmatrix} 0 & 9 & \infty & \infty & \infty & \infty \\ 9 & 0 & 7 & 5 & \infty & \infty \\ \infty & 7 & 0 & \infty & 6 & 7 \\ \infty & 5 & \infty & 0 & \infty & \infty \\ \infty & \infty & 6 & \infty & 0 & \infty \\ \infty & \infty & 7 & \infty & \infty & 0 \end{bmatrix}$$

加入顺序 ① V0

② V1 (V0, V1)

③ V3 (V1, V3)

④ V2 (V1, V2)

⑤ V4 (V2, V4)

⑥ V5 (V2, V5)

四.10 邻接矩阵

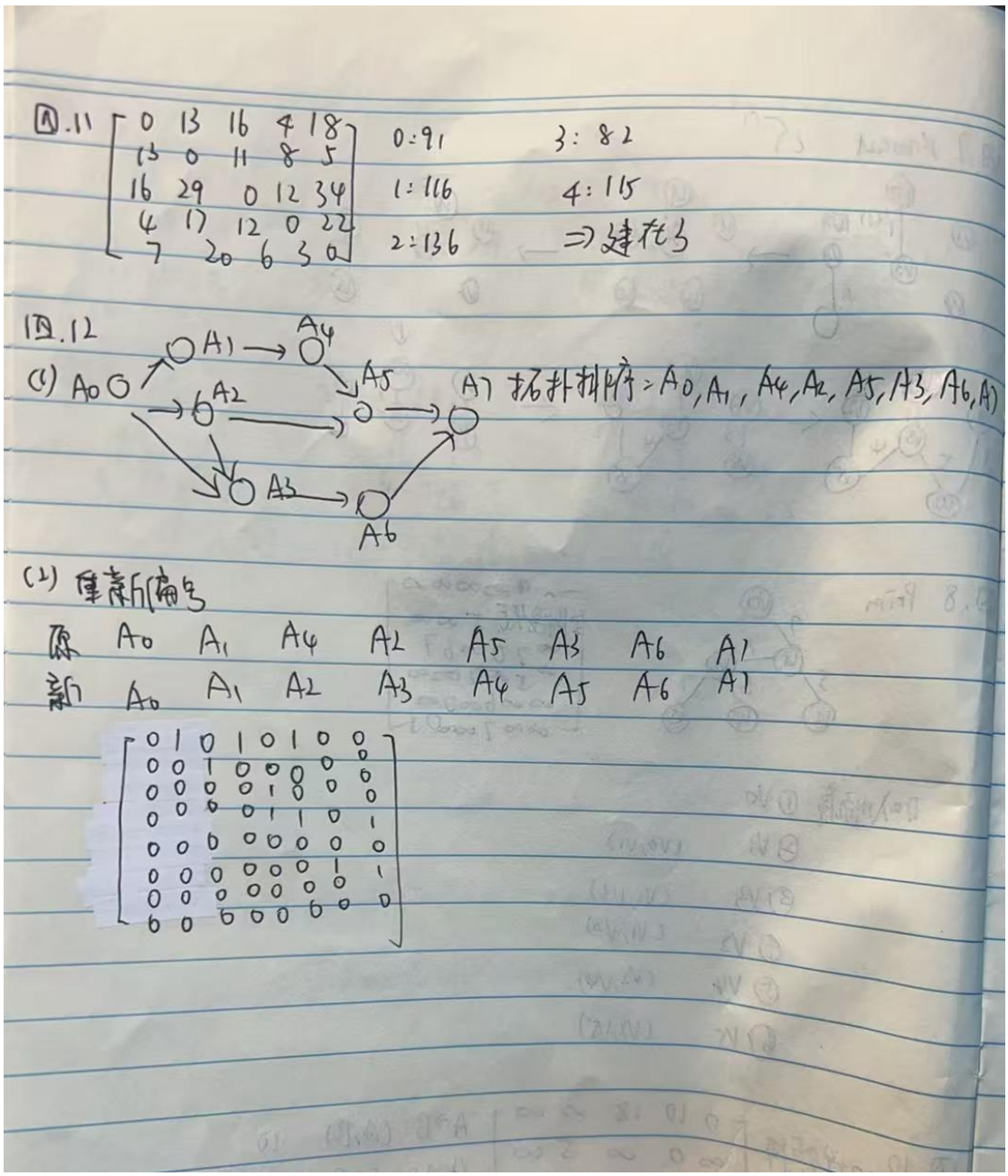
$$\begin{bmatrix} 0 & 10 & 18 & \infty & \infty \\ \infty & 0 & \infty & 5 & \infty \\ \infty & 5 & 0 & \infty & \infty \\ \infty & \infty & 2 & 0 & 3 \\ \infty & \infty & 2 & \infty & 0 \end{bmatrix}$$

A→B: (A,B) 10

A→C: (A,B,D,C) 17

A→D: (A,B,D) 15

A→E: (A,B,D,E) 18



5.5

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;
class Graph {
private:
    int V; // 顶点数
    vector<int>* adj; // 邻接表

public:
```

```
// 构造函数
Graph(int V) {
    this->V = V;
    adj = new vector<int>[V];
}

// 添加边
void addEdge(int v, int w) {
    adj[v].push_back(w);
    adj[w].push_back(v); // 因为是无向图，所以需要添加两条边
}

// 非递归 DFS
void DFSUtil(int v, vector<bool>& visited) {
    stack<int> stack;

    // 将起始顶点压入栈
    stack.push(v);

    while (!stack.empty()) {
        // 弹出栈顶元素
        v = stack.top();
        stack.pop();

        // 如果该顶点未被访问，则访问之
        if (!visited[v]) {
            cout << v << " ";
            visited[v] = true;
        }

        // 遍历所有邻接顶点
        for (int i = 0; i < adj[v].size(); i++) {
            // 如果邻接顶点未被访问，则将其压入栈
            if (!visited[adj[v][i]]) {
                stack.push(adj[v][i]);
            }
        }
    }
}

// DFS 函数，用于调用 DFSUtil
void DFS(int v) {
    vector<bool> visited(V, false);
    DFSUtil(v, visited);
```

```
    }
};

int main() {
    // 创建一个有 4 个顶点的图
    Graph g(4);

    // 添加边
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 3);

    cout << "深度优先搜索（从顶点 2 开始）: ";
    g.DFS(2); // 从顶点 2 开始深度优先搜索

    return 0;
}
```

5.7

```
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

class Graph {
private:
    int V; // 顶点数
    vector<vector<int>> adj; // 邻接表

public:
    // 构造函数
    Graph(int V) {
        this->V = V;
        adj.resize(V);
    }

    // 添加边
    void addEdge(int v, int w) {
        adj[v].push_back(w);
    }

    // DFS 函数，用于寻找从 u 到 v 的路径
```

```
    void DFS(int v, int u, int dest, vector<int>& path, vector<bool>& visited) {
        // 标记当前顶点为已访问
        visited[v] = true;
        // 将当前顶点添加到路径中
        path.push_back(v);

        // 如果当前顶点是目标顶点，输出路径并返回
        if (v == dest) {
            for (int i = path.size() - 1; i >= 0; i--) {
                cout << path[i] << " ";
            }
            cout << endl;
            return;
        }

        // 遍历所有邻接顶点
        for (int i = 0; i < adj[v].size(); i++) {
            int next = adj[v][i];
            // 如果邻接顶点未被访问，则继续 DFS
            if (!visited[next]) {
                DFS(next, u, dest, path, visited);
            }
        }

        // 如果通过当前路径没有找到目标，则回溯
        path.pop_back();
    }

    // 寻找从 u 到 v 的路径
    void findPath(int u, int v) {
        vector<int> path; // 存储路径
        vector<bool> visited(V, false); // 访问标记

        DFS(u, u, v, path, visited);
    }
};

int main() {
    // 创建一个有 5 个顶点的图
    Graph g(5);

    // 添加边
    g.addEdge(0, 1);
    g.addEdge(0, 4);
```

```
    g.addEdge(1, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
    g.addEdge(2, 3);
    g.addEdge(3, 4);

    int u = 0, v = 3;
    cout << "从顶点 " << u << " 到顶点 " << v << " 的路径: ";
    g.findPath(u, v); // 寻找从顶点 u 到顶点 v 的路径

    return 0;
}
```

5.12

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// 定义边的类
class Edge {
public:
    int u, v, weight;
    Edge(int u, int v, int weight) : u(u), v(v), weight(weight) {}
};

// 定义图的类
class Graph {
    int V; // 顶点数量
    vector<Edge> edges; // 存储所有边的列表

public:
    // 构造函数
    Graph(int V) : V(V) {}

    // 添加边和它们的权值
    void addEdge(int u, int v, int weight) {
        edges.push_back(Edge(u, v, weight));
    }

    // 深度优先搜索（DFS）用于检查连通性
    void DFS(int v, vector<bool>& visited) {
        visited[v] = true; // 标记当前顶点为已访问
        for (auto& edge : edges) {
```

```
        // 遍历与顶点 v 相连的边
        if (edge.u == v && !visited[edge.v]) {
            DFS(edge.v, visited);
        }
        else if (edge.v == v && !visited[edge.u]) {
            DFS(edge.u, visited);
        }
    }
}

// 检查图是否连通
bool isConnected() {
    vector<bool> visited(V, false); // 初始化访问标记数组
    DFS(0, visited); // 从顶点 0 开始 DFS
    for (bool v : visited) {
        if (!v) return false; // 如果有未访问的顶点，返回 false
    }
    return true; // 所有顶点都访问过，返回 true
}

// 实现破圈法来构造最小生成树
void breakCycle() {
    // 按权值降序排序边
    sort(edges.begin(), edges.end(), [](const Edge& a, const Edge& b) {
        return a.weight < b.weight;
        });

    vector<Edge> mst; // 用于存储最小生成树的边
    for (int i = 0; i < edges.size() && mst.size() < V - 1; ++i) {
        Edge e = edges[i];
        vector<Edge> tempEdges = mst;
        tempEdges.push_back(e);
        if (isConnected(tempEdges)) {
            mst.push_back(e);
        }
    }
    edges = mst; // 更新图的边
}

// 打印图的函数，用于展示结果
void printGraph() {
    for (Edge& e : edges) {
        cout << e.u << " -- " << e.v << " [权重: " << e.weight << "]" << endl;
    }
```

```
    }
};

int main() {
    Graph g(4); // 创建一个包含 4 个顶点的图
    // 添加边
    g.addEdge(0, 1, 10);
    g.addEdge(1, 2, 15);
    g.addEdge(2, 3, 4);
    g.addEdge(3, 1, 6);
    g.addEdge(0, 2, 5);
    cout << "原始图:" << endl;
    g.printGraph(); // 打印原始图
    g.breakCycle(); // 应用破圈法
    cout << "最小生成树:" << endl;
    g.printGraph(); // 打印应用破圈法后的图
    return 0;
}
```

5.14

```
#include <iostream>
#include <vector>
#include <limits.h>
using namespace std;

void shortestpath(vector<vector<int>> a, int v, int g[], int dist[], vector<vector<int>>&
path) {
    int n = a.size();
    vector<bool> s(n, false); // 用来标记节点是否已访问
    int i, j, k, w, min, u;

    // 初始化 dist 数组和 path 数组
    for (i = 0; i < n; i++) {
        dist[i] = a[v][i];
        s[i] = false;
        if (i != v && dist[i] < 100) path[i].push_back(v); // 100=maxweight
        else path[i].push_back(-1); // 未访问的节点
    }

    s[v] = true;
    dist[v] = 0;

    for (i = 0; i < n - 1; i++) {
        min = 100;
```

```
        u = -1;

        // 找到未访问的最小 dist 值对应的节点 u
        for (j = 0; j < n; j++) {
            if (!s[j] && dist[j] < min) {
                u = j;
                min = dist[j];
            }
        }

        s[u] = true;

        // 更新所有邻接点的 dist 值和 path
        for (k = 0; k < n; k++) {
            w = a[u][k];
            if (!s[k] && w < 100 && dist[u] + w < dist[k]) {
                dist[k] = dist[u] + w;
                g[k] = g[u] + 1;
                path[k].clear(); // 清空之前的路径
                path[k].push_back(u); // 更新路径
            }
            else if (!s[k] && w < 100 && dist[u] + w == dist[k]&&g[k]>g[u]+1) {
                g[k] = g[u] + 1;
                path[k].clear(); // 清空之前的路径
                path[k].push_back(u); // 更新路径
            }
        }
    }
}

// 打印从源点到目标点的路径
void printPath(const vector<vector<int>>& path, int target) {
    if (path[target][0] == -1) {
        cout << "No path available.";
        return;
    }

    vector<int> finalPath;
        for (int v = target; v != -1; v = path[v][0]) {
            finalPath.push_back(v);
        }
    // 输出路径
    for (int i = finalPath.size() - 1; i >= 0; i--) {
        cout << finalPath[i] << " ";
```

```
    }
    cout << endl;
}

int main() {
    int n;
    cin >> n;
    vector<vector<int>> a(n, vector<int>(n));

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cin >> a[i][j];
        }
    }

    int dist[100];
    int g[100];
    vector<vector<int>> path(n); // 初始化 path 为 n 个空向量
    shortestpath(a, 0, g, dist, path);
    printPath(path, 3); // 输出从源点(0)到目标点(1)的路径
    return 0;
}
```

5.16

```
#include <iostream>
#include <list>
#include <stack>
#include <vector>

using namespace std;

// 函数用于执行深度优先搜索
void dfs(int v, vector<bool>& visited, vector<list<int>>& adj, stack<int>& Stack) {
    visited[v] = true;

    // 遍历所有邻接顶点
    for (int i : adj[v]) {
        if (!visited[i]) {
            dfs(i, visited, adj, Stack);
        }
    }

    // 将顶点压入栈中
    Stack.push(v);
```

```
}

// 函数用于执行拓扑排序
void topologicalSort(int V, vector<list<int>>& adj) {
    // 创建一个栈来存储拓扑排序结果
    stack<int> Stack;

    // 创建一个布尔向量来跟踪访问过的顶点
    vector<bool> visited(V, false);

    // 对每个顶点执行 DFS
    for (int i = 0; i < V; i++) {
        if (!visited[i]) {
            dfs(i, visited, adj, Stack);
        }
    }

    // 打印拓扑排序结果
    while (!Stack.empty()) {
        cout << Stack.top() << " ";
        Stack.pop();
    }
    cout << endl;
}

// 主函数
int main() {
    // 图中的顶点数
    int V = 6;

    // 邻接表表示的图
    vector<list<int>> adj(V);

    // 添加边
    adj[5].push_back(2);
    adj[5].push_back(0);
    adj[4].push_back(0);
    adj[4].push_back(1);
    adj[2].push_back(3);
    adj[3].push_back(1);

    // 执行拓扑排序
    topologicalSort(V, adj);
```

```
    return 0;
}
```