1. （1）=>（2）：如果G是一棵树，那么它必定是连通图。因为在树中，任意两个顶点之间都存在一条路径，并且移除任何一条边后都会破坏连通性。

（2）=>（3）：如果G是连通图，那么对于任意两个顶点vu，在G中一定存在一条路径连接它们。由于G是无向图，路径是简单路径，即不存在重复经过的边或顶点。

（3）=>（4）：如果对任意两个顶点vu，只有一条简单路径，那么图G中不存在环路。假设存在一条环路，那么取环路上的两个顶点，它们之间至少存在两条简单路径，与题设矛盾。另外，由于G有n个顶点，所以最多只有n-1条边。

（4）=>（1）：如果G没有环且包含n-1条边，那么G必定是连通图。因为如果不是连通图，那么至少存在两个连通分量，而每个连通分量中至少有一个顶点，所以总共顶点数目大于n，与题设矛盾。

四条命题彼此蕴含，因此是等价的。

2. 关于DFS和BFS两个图的遍历算法，回答下列问题:

(1)两个算法之间的联系与区别?

两个算法的联系在于他们都可以用来找到图中的连通分量，遍历图中的节点和寻找最短路径等，区别在于DFS是深度优先算法，沿着每条路径深入直到不能再继续搜索，然后返回上一级继续搜索其他路径，用递归实现；BFS是广度优先算法，从起点开始先访问相邻节点，再访问相邻节点的子节点，使用队列实现。

(2)DFS对内存的要求比BFS要少的多，为什么?

DFS在搜索时只保存当前路径上的节点，不需要保存所有已访问节点，而BFS需要一个队列的内存保存需要访问的节点。

(3)哪个算法更快?说明你的理由

BFS和DFS的时间复杂度都是O（n+e），如果需要查找的节点位置比较深，DFS略快。

(4)哪个算法更好? 说明你的理由

寻找最短路径或者确定两个节点间最短距离时BFS更好，找到某个节点时DFS更好，因为BFS会先访问距离起始节点近的节点，可以快速找到最短路径。而且BFS空间复杂度较高，如果需要更少的空间也可以用DFS。

3.

算法思想：

首先，初始化两个数组prev[]和dist[]，分别用于存储最短路径的前驱节点和起始节点到各节点的距离，将它们都初始化为无穷大（INF），表示起始节点到它们之间没有直接连接的边。将起始节点加入队列，并将其距离设为0。然后，对于队列中的每个节点，遍历其所有邻居节点，如果邻居节点没有被访问过，就将它加入队列中，并更新它的距离和前驱节点。重复上述步骤，直到队列为空或找到目标节点。如果找到了目标节点，可以利用 prev[] 数组来获取从起始节点到目标节点的最短路径，使用一个二维数组 graph[][] 来表示图，其中 graph[i][j].val 表示从节点 i 到节点 j 的边的目标节点，如果不存在这条边则为 -1。graph[i][j].weight 表示边的权重，即从节点 i 到节点 j 的距离。

代码实现：

const int MAX\_SIZE = 100; //假设数组的最大长度

const int INF = INT\_MAX;

struct Node {

int val;//节点值

int weight;//边的权重

Node(int v, int w) : val(v), weight(w) {}

};

void bfs(Node graph[MAX\_SIZE][MAX\_SIZE], int start, int end, int prev[MAX\_SIZE], int dist[MAX\_SIZE]) {广度优先搜索算法，找到从起始节点到目标节点的最短路径

int n = MAX\_SIZE;

//初始化prev和dist数组

for (int i = 0; i < n; ++i) {

prev[i] = -1;

dist[i] = INF;

}

queue<int> q;

q.push(start);

dist[start] = 0;

while (!q.empty()) {

int curr = q.front();

q.pop();

for (int i = 0; i < n; ++i) {

if (graph[curr][i].val != -1) {

int neighbor = graph[curr][i].val;

if (dist[curr] + graph[curr][i].weight < dist[neighbor]) {

dist[neighbor] = dist[curr] + graph[curr][i].weight;

prev[neighbor] = curr;

q.push(neighbor);

}

}

}

}

}

void getPath(int prev[MAX\_SIZE], int start, int end, int path[MAX\_SIZE]) {

//从起始节点到目标节点的最短路径中获取具体的路径节点序列

int index = 0;

for (int curr = end; curr != -1; curr = prev[curr]) {

path[index++] = curr;}

int pathLength = index;

for (int i = 0; i < pathLength / 2; ++i) {

int temp = path[i];

path[i] = path[pathLength - 1 - i];

path[pathLength - 1 - i] = temp;

}

if (path[0] != start) {

//清空path数组

for (int i = 0; i < MAX\_SIZE; ++i) {

path[i] = -1;

}

}

}

void bfsShortestPath(Node graph[MAX\_SIZE][MAX\_SIZE], int start, int end, int path[MAX\_SIZE]) {//计算最短路径并将结果存储

int n = MAX\_SIZE;

int prev[MAX\_SIZE];

int dist[MAX\_SIZE];

bfs(graph, start, end, prev, dist);

getPath(prev, start, end, path);

}

时间复杂度：O(n^2)

空间复杂度：O(n)