第一题

首先，我们考虑单循环比赛的情况。

单循环比赛是指每个选手都需要和其他所有选手各比赛一次，且仅比赛一次。这种情况下，我们可以用无向完全图来描述。在无向完全图中，每个节点代表一个选手，每对节点之间都有一条边，代表这两个选手之间的一场比赛。

假设有n个选手，那么他们之间的比赛数量就是图中边的数量。在无向完全图中，边的数量可以用组合数学中的组合公式C(n, 2)来计算，即n个不同的元素中取出2个进行组合的方法数，数学公式为：  
C(n, 2) = n! / (2!(n-2)!) = n(n-1)/2

因此，单循环比赛需要进行n(n-1)/2场比赛。

接下来，我们考虑主客场制的联赛情况。

主客场制联赛是指每对选手之间都需要进行两场比赛，一场在A选手的主场，另一场在B选手的主场。这种情况下，我们可以用有向完全图来描述。在有向完全图中，每个节点代表一个选手，每对节点之间都有两条方向相反的边，代表这两个选手之间的两场比赛。

同样地，假设有n个选手，那么他们之间的比赛数量就是图中边的数量的两倍。由于每条边都代表一场比赛（主场或客场），所以总的比赛数量就是n(n-1)。

总结来说，单循环比赛需要进行n(n-1)/2场比赛，可以用无向完全图来描述；主客场制联赛需要进行n(n-1)场比赛，可以用有向完全图来描述。

第二题

（1）证明在任意一个有向图中，所有顶点的入度之和与出度之和相等：

设一个有向图 *G* 有 *n* 个顶点，记作 *V*={*v*1​,*v*2​,…,*vn*​}。

对于每一个顶点 *vi*​，我们定义其入度为 *ID*(*vi*​)，出度为 *OD*(*vi*​)。

考虑图 *G* 中所有的边，每一条有向边都有一个起始顶点和一个终止顶点。因此，每一条边都会贡献一个出度到一个顶点，并贡献一个入度到另一个顶点。

所以，所有顶点的出度之和为 ∑*i*=1*n*​*OD*(*vi*​)，所有顶点的入度之和为 ∑*i*=1*n*​*ID*(*vi*​)。

由于每一条边都同时贡献一个出度和一个入度，所以 ∑*i*=1*n*​*OD*(*vi*​)=∑*i*=1*n*​*ID*(*vi*​)。

因此，所有顶点的入度之和与出度之和相等。

（2）证明任一无向图中各顶点的度的和一定为偶数：

设一个无向图 *G* 有 *n* 个顶点，记作 *V*={*v*1​,*v*2​,…,*vn*​}。

对于每一个顶点 *vi*​，我们定义其度为 *D*(*vi*​)。

在无向图中，每一条边都连接两个顶点，因此每一条边都会贡献两个度到其连接的两个顶点。

所以，所有顶点的度之和为 ∑*i*=1*n*​*D*(*vi*​)。

由于每一条边都贡献两个度，所以 ∑*i*=1*n*​*D*(*vi*​) 一定是偶数（因为每条边贡献的度数是2的倍数）。

因此，任一无向图中各顶点的度的和一定为偶数。

第三题

在一个强连通图中，各顶点的度（即入度和出度之和）具有以下特点：

1. **度之和的下限**：由于强连通图要求任意两个顶点之间都存在路径（无论是直接的还是通过其他顶点的），这意味着每个顶点至少有一个入边和一个出边（除了可能存在的自环）。因此，每个顶点的度（入度加出度）至少为2（不考虑自环），除非该图是只有一个顶点的图（即平凡图），其度为0。
2. **没有孤立顶点**：在强连通图中，不存在度为0的孤立顶点，因为这样的顶点无法满足与其他所有顶点都存在路径的要求。
3. **度与边的关系**：在具有n个顶点的强连通图中，边的数量（记为m）至少为n（因为每个顶点至少有一条出边），但通常边数会更多。边的数量m与顶点度之和的关系是：所有顶点的度之和等于边数的两倍（因为每条边都会贡献两个度到一个顶点）。
4. **自环不影响强连通性**：自环（即顶点连接自身的边）不会改变图的强连通性，但会增加顶点的度。即使图中存在自环，只要任意两个顶点之间都存在路径，该图仍然是强连通的。
5. **与弱连通图的区别**：在弱连通图中，只需要存在从任意顶点到任意其他顶点的有向路径（不考虑方向）即可。这意味着在弱连通图中，可能存在一些顶点的入度或出度为0，但仍然满足弱连通性的定义。而在强连通图中，由于要求任意两个顶点之间都存在双向路径，因此不存在入度或出度为0的顶点。

总之，在强连通图中，各顶点的度至少为2（不考虑自环和平凡图），并且不存在度为0的孤立顶点。同时，所有顶点的度之和等于边数的两倍。

第四题

为了求解有向图G中顶点vi的入度、出度和度，我们可以使用邻接矩阵。假设邻接矩阵为adjMatrix，其大小为n x n，其中n是图中顶点的数量。

以下是求解顶点vi的入度、出度和度的算法步骤（伪代码）：

|  |  |
| --- | --- |
|  | // 假设 adjMatrix 是有向图的邻接矩阵，n 是图中顶点的数量，vi 是要查询的顶点索引 |
|  | function calculateDegrees(adjMatrix, n, vi): |
|  | // 初始化入度、出度和度 |
|  | inDegree = 0 |
|  | outDegree = 0 |
|  | degree = 0 |
|  |  |
|  | // 计算入度 |
|  | for j in range(n): |
|  | if adjMatrix[j][vi] != 0: // 如果从顶点vj到顶点vi有一条边 |
|  | inDegree += 1 |
|  |  |
|  | // 计算出度 |
|  | for j in range(n): |
|  | if adjMatrix[vi][j] != 0: // 如果从顶点vi到顶点vj有一条边 |
|  | outDegree += 1 |
|  |  |
|  | // 计算度（入度加出度） |
|  | degree = inDegree + outDegree |
|  |  |
|  | // 返回结果 |
|  | return inDegree, outDegree, degree |
|  |  |
|  | // 使用示例 |
|  | n = 5 // 假设有5个顶点 |
|  | adjMatrix = [ |
|  | [0, 1, 0, 0, 0], |
|  | [0, 0, 1, 1, 0], |
|  | [0, 0, 0, 1, 1], |
|  | [1, 0, 0, 0, 1], |
|  | [0, 0, 0, 0, 0] |
|  | ] |
|  |  |
|  | vi = 2 // 假设要查询顶点2的入度、出度和度 |
|  | inDeg, outDeg, deg = calculateDegrees(adjMatrix, n, vi) |
|  | print("顶点", vi, "的入度为:", inDeg) |
|  | print("顶点", vi, "的出度为:", outDeg) |
|  | print("顶点", vi, "的度为:", deg) |

这个算法首先初始化入度、出度和度为0，然后遍历邻接矩阵的行和列来计算入度和出度。如果邻接矩阵的adjMatrix[j][vi]不为0，说明从顶点vj到顶点vi有一条边，因此入度加1。同样地，如果adjMatrix[vi][j]不为0，说明从顶点vi到顶点vj有一条边，因此出度加1。最后，度就是入度和出度的和。

第五题

在邻接矩阵表示的图G中，firstadj(G, v)函数通常用于找到与顶点v相邻的第一个顶点（如果存在的话），而nextadj(G, v, w)函数则用于找到与顶点v相邻的顶点w之后的下一个顶点（如果存在的话）。

但是，邻接矩阵并不直接支持这种顺序遍历的语义，因为它是以矩阵的形式存储图的信息，无法直接给出邻接顶点的顺序。然而，我们可以基于邻接矩阵设计一个算法来模拟这种顺序遍历，通常是通过将邻接矩阵的行或列视为一个数组，并进行遍历。

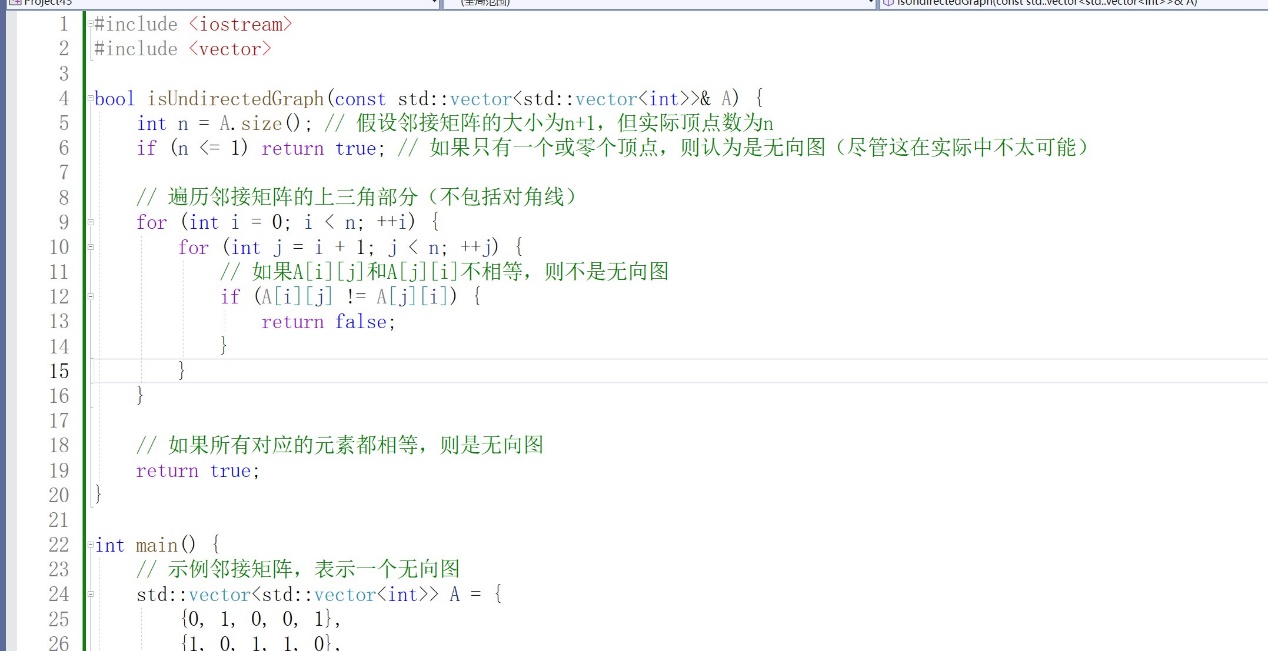
以下是一个简化的实现示例，假设我们按照邻接矩阵的行来模拟顺序遍历：

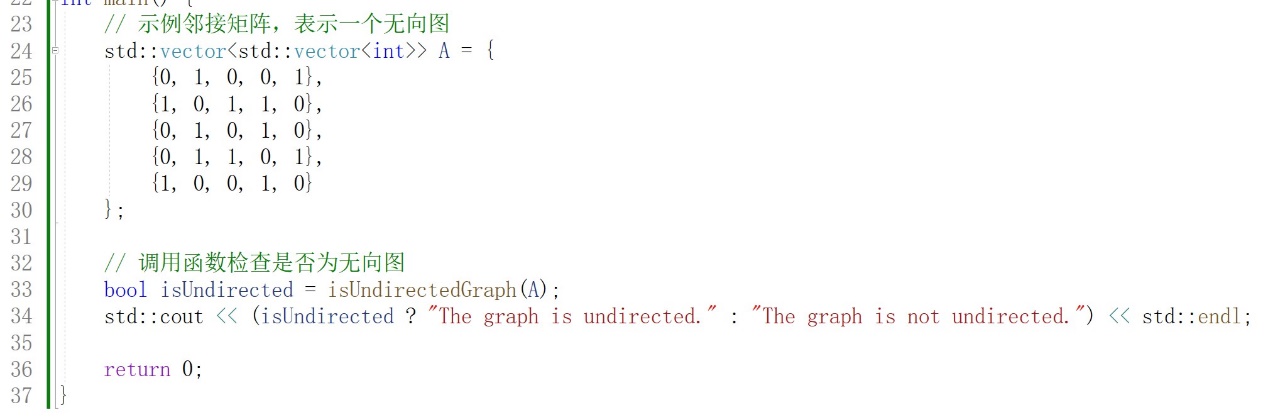
|  |  |
| --- | --- |
|  | def firstadj(G, v): |
|  | """ |
|  | 返回与顶点v相邻的第一个顶点（如果有的话），否则返回None |
|  | """ |
|  | n = len(G) # 假设G是n x n的邻接矩阵 |
|  | for w in range(n): |
|  | if G[v][w] != 0: # 如果v和w之间有边 |
|  | return w # 返回第一个相邻的顶点w |
|  | return None # 如果没有相邻的顶点 |
|  |  |
|  | def nextadj(G, v, w): |
|  | """ |
|  | 返回与顶点v相邻的顶点w之后的下一个顶点（如果有的话），否则返回None |
|  | 注意：这个实现假设顶点w是v的一个相邻顶点，并且按照邻接矩阵的列顺序来寻找下一个相邻顶点 |
|  | """ |
|  | n = len(G) # 假设G是n x n的邻接矩阵 |
|  | # 从w的下一个顶点开始搜索 |
|  | for next\_w in range(w + 1, n): |
|  | if G[v][next\_w] != 0: # 如果v和next\_w之间有边 |
|  | return next\_w # 返回下一个相邻的顶点next\_w |
|  | return None # 如果没有下一个相邻的顶点 |
|  |  |
|  | # 示例 |
|  | G = [ |
|  | [0, 1, 0, 0, 0], |
|  | [1, 0, 1, 1, 0], |
|  | [0, 1, 0, 1, 1], |
|  | [0, 1, 1, 0, 1], |
|  | [0, 0, 1, 1, 0] |
|  | ] |
|  | v = 1 |
|  | w = 2 # 假设w是v的一个相邻顶点 |
|  |  |
|  | print(firstadj(G, v)) # 应该输出与顶点1相邻的第一个顶点（如果存在） |
|  | print(nextadj(G, v, w)) # 应该输出与顶点1相邻的顶点2之后的下一个顶点（如果存在） |

请注意，这个实现假设你正在按照邻接矩阵的列顺序来寻找相邻顶点。但是，由于邻接矩阵通常不保证顶点之间的顺序（即，它只是一个布尔矩阵，表示边是否存在），因此这种顺序可能是任意的，并且可能不适用于所有情况。

如果你需要更复杂的顺序遍历（例如深度优先搜索或广度优先搜索），那么你可能需要使用不同的数据结构（如邻接表）来表示图，或者使用基于邻接矩阵的更复杂算法来模拟顺序遍历。

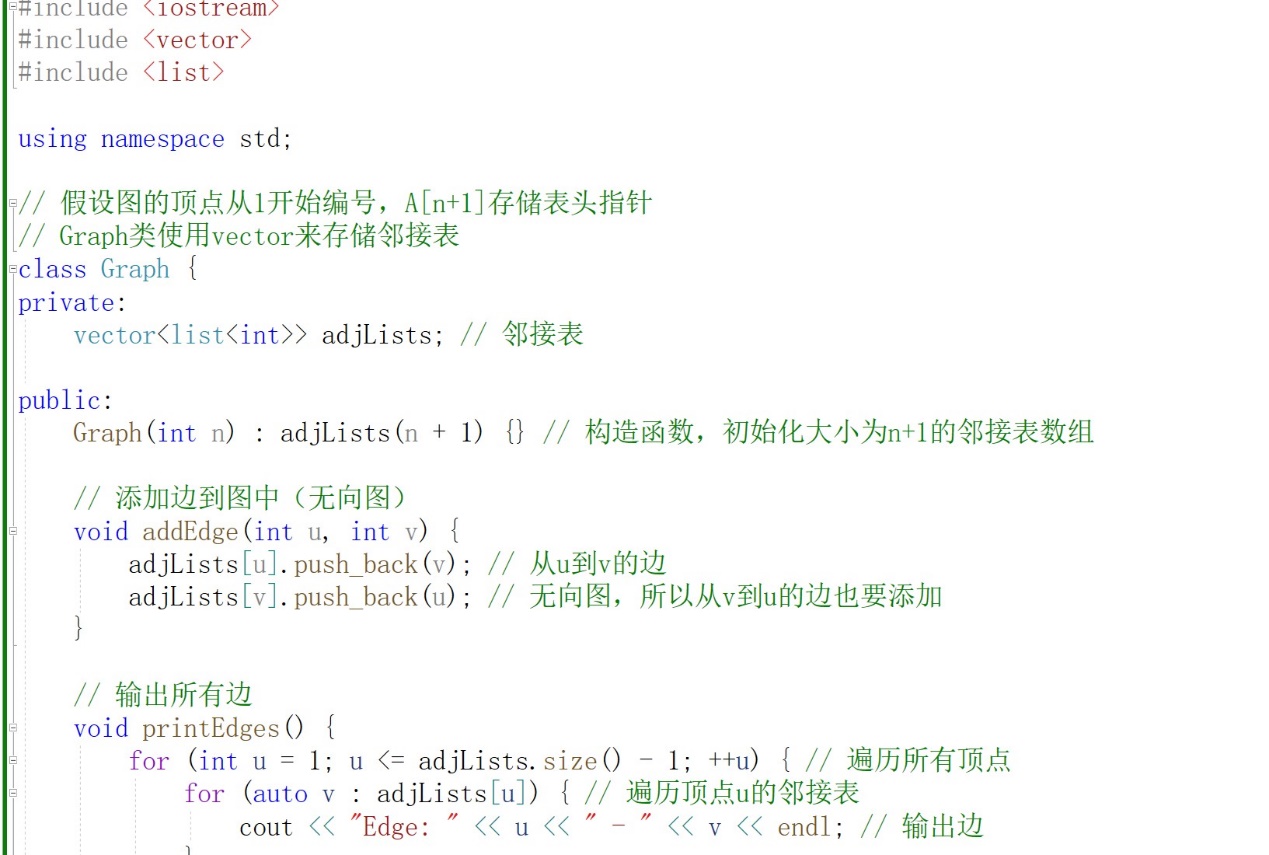
第六题



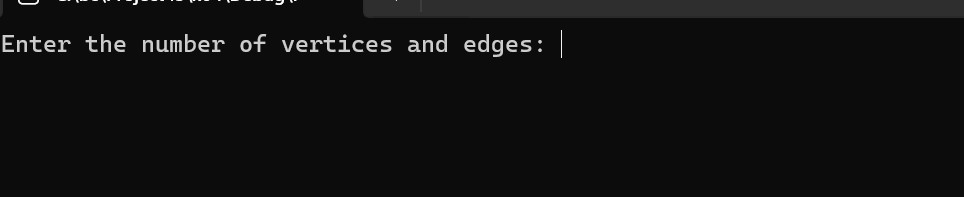




第七题



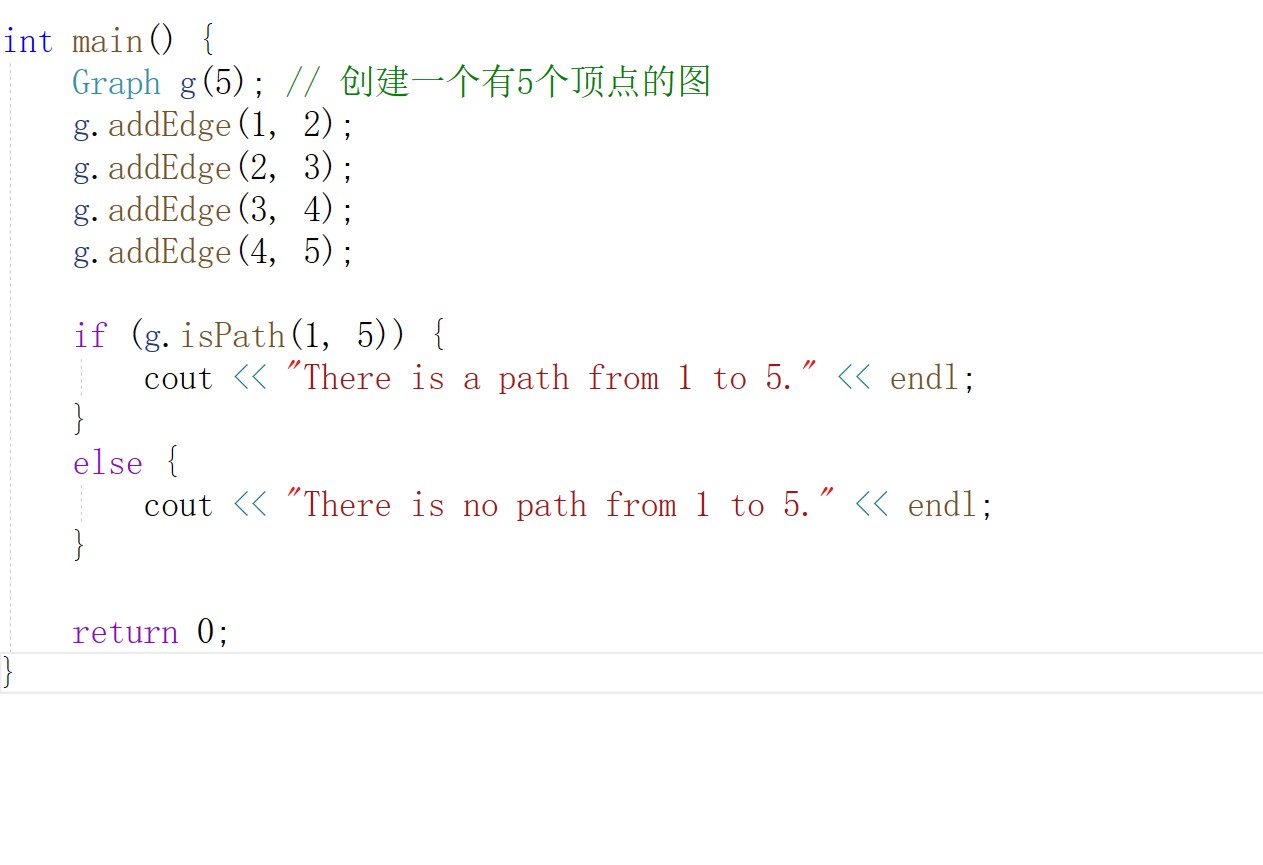


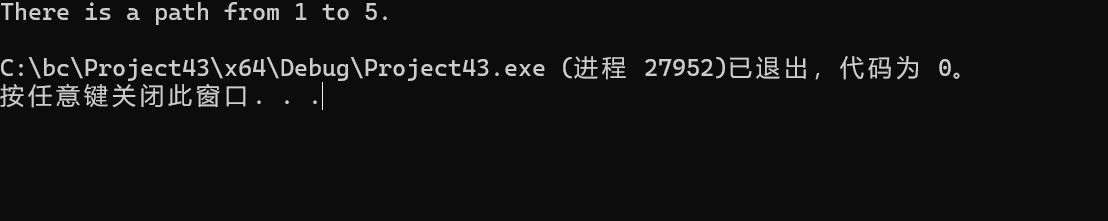


第九题



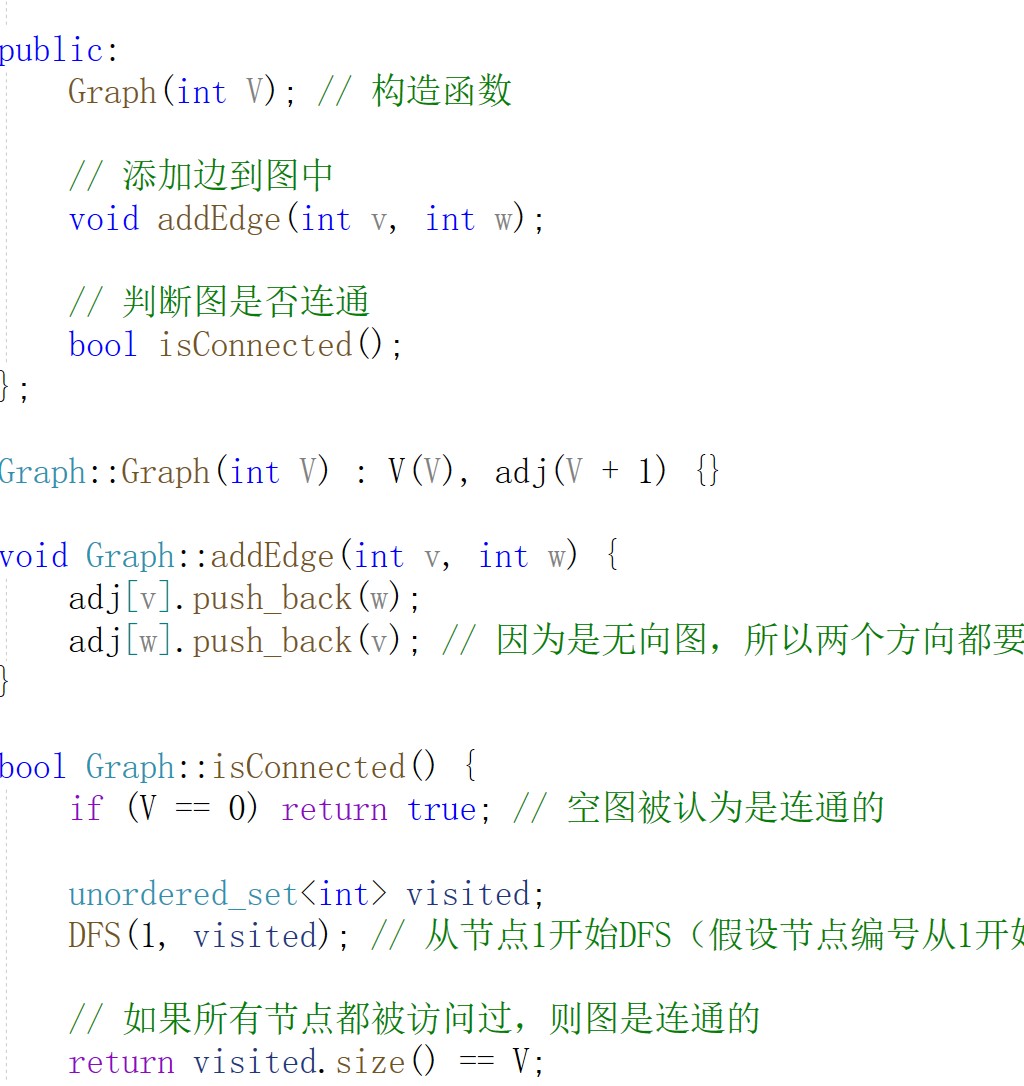


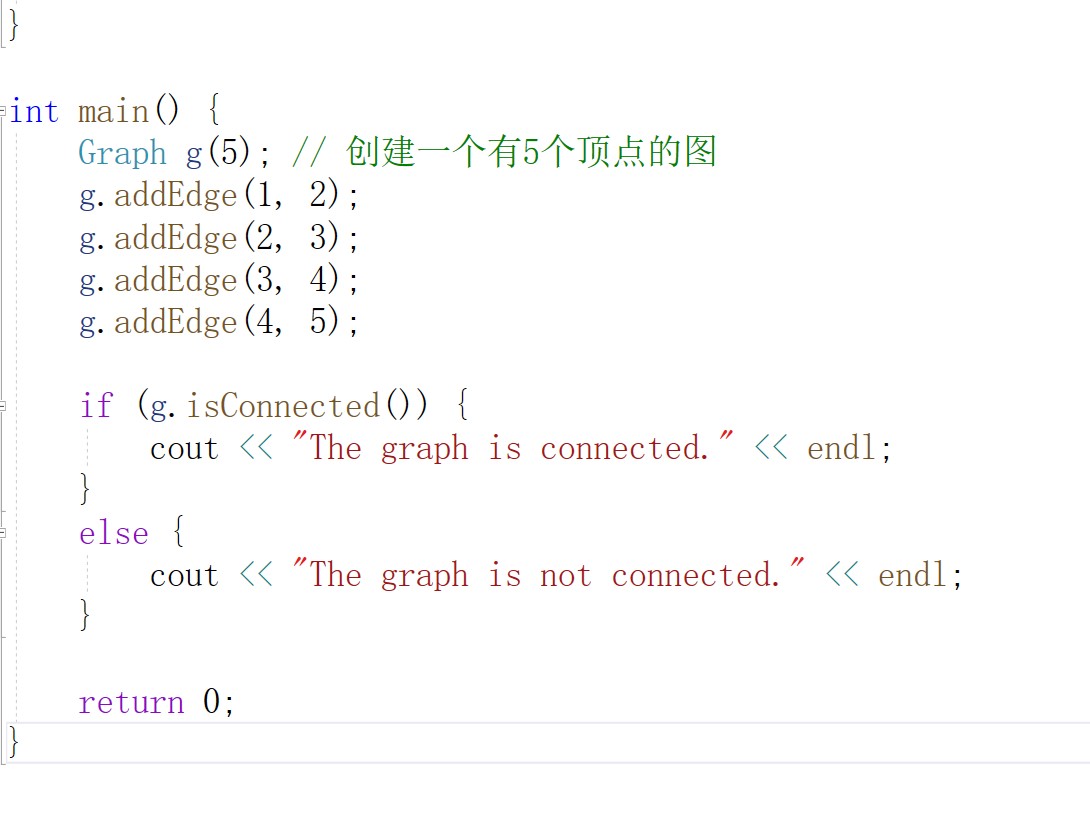




第十题

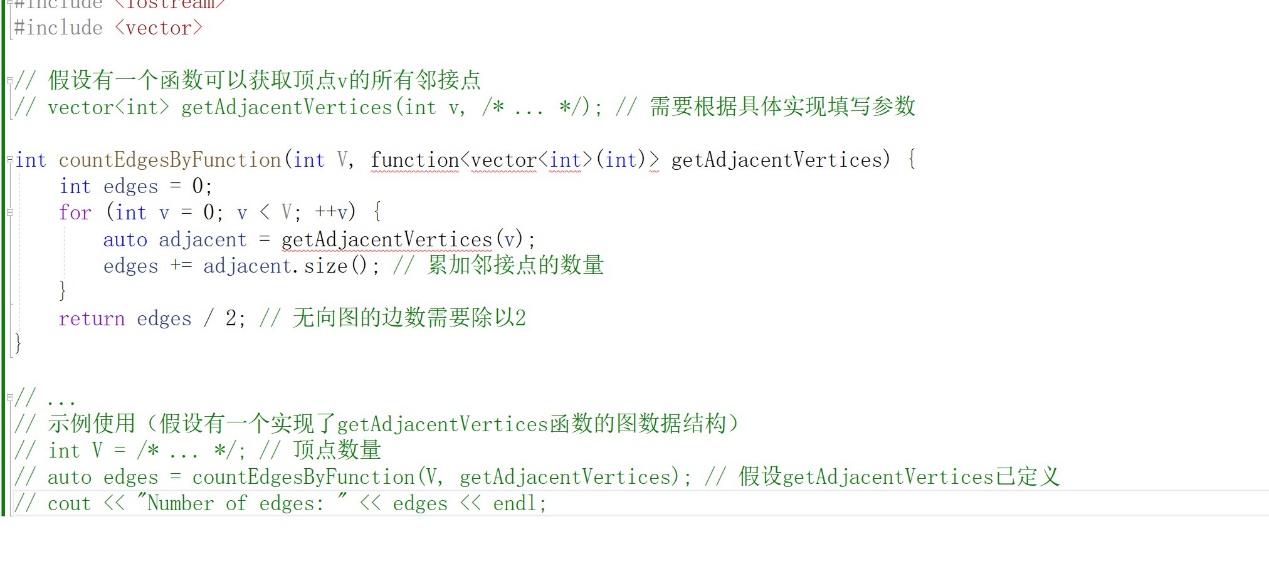




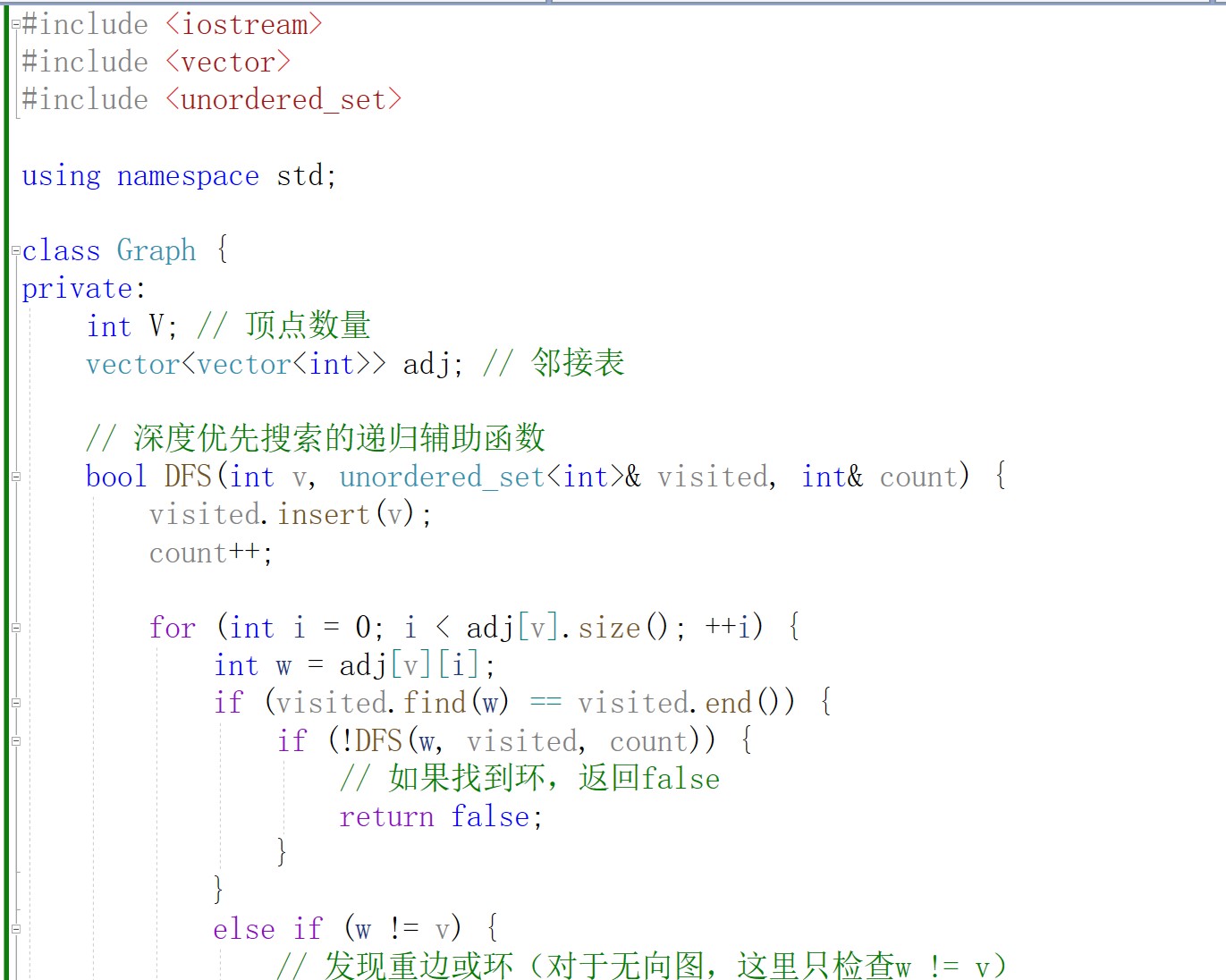


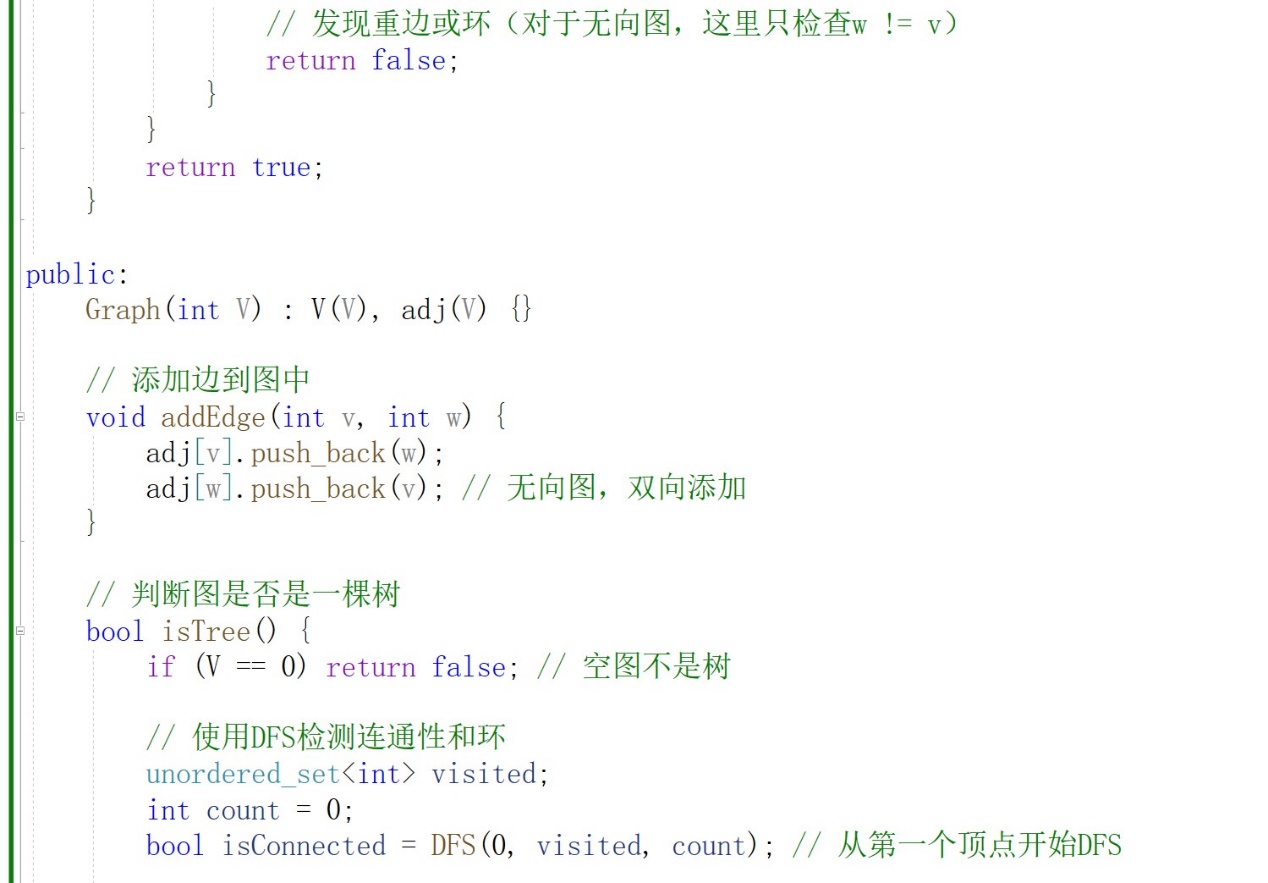


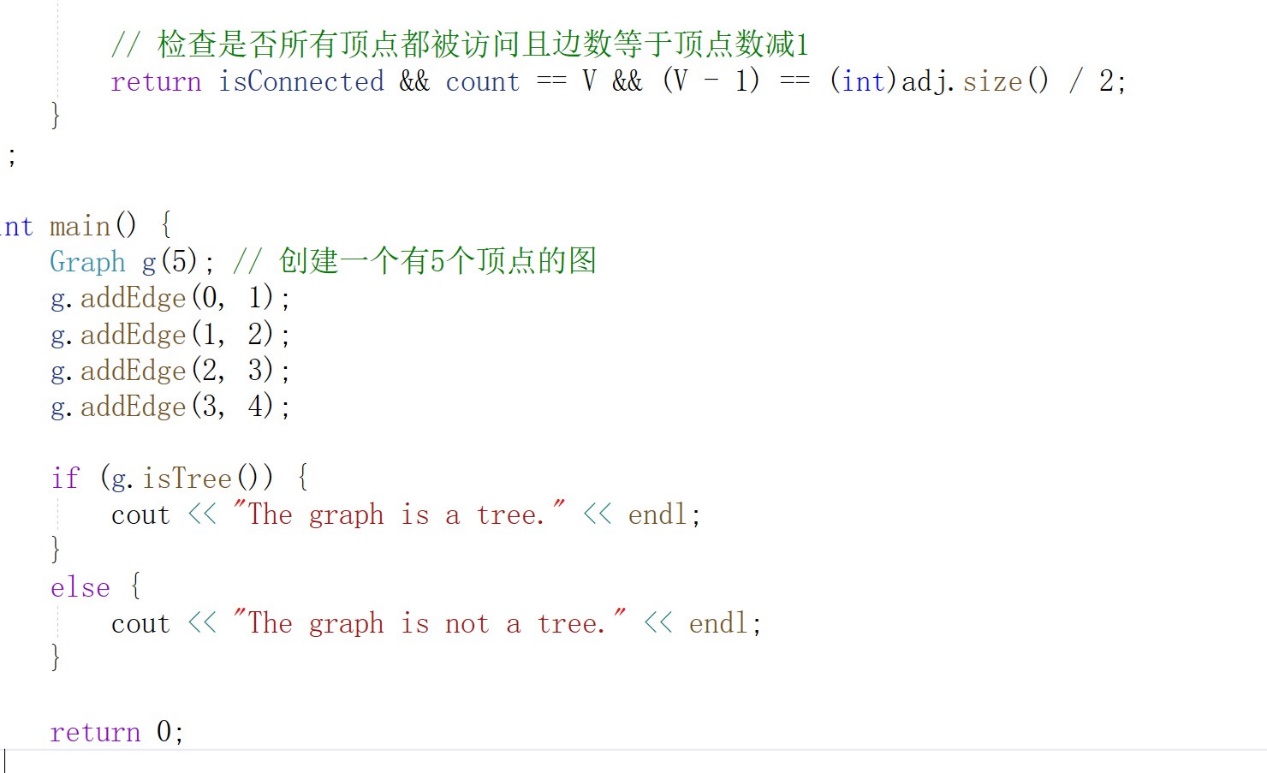
第十一题



第十二题



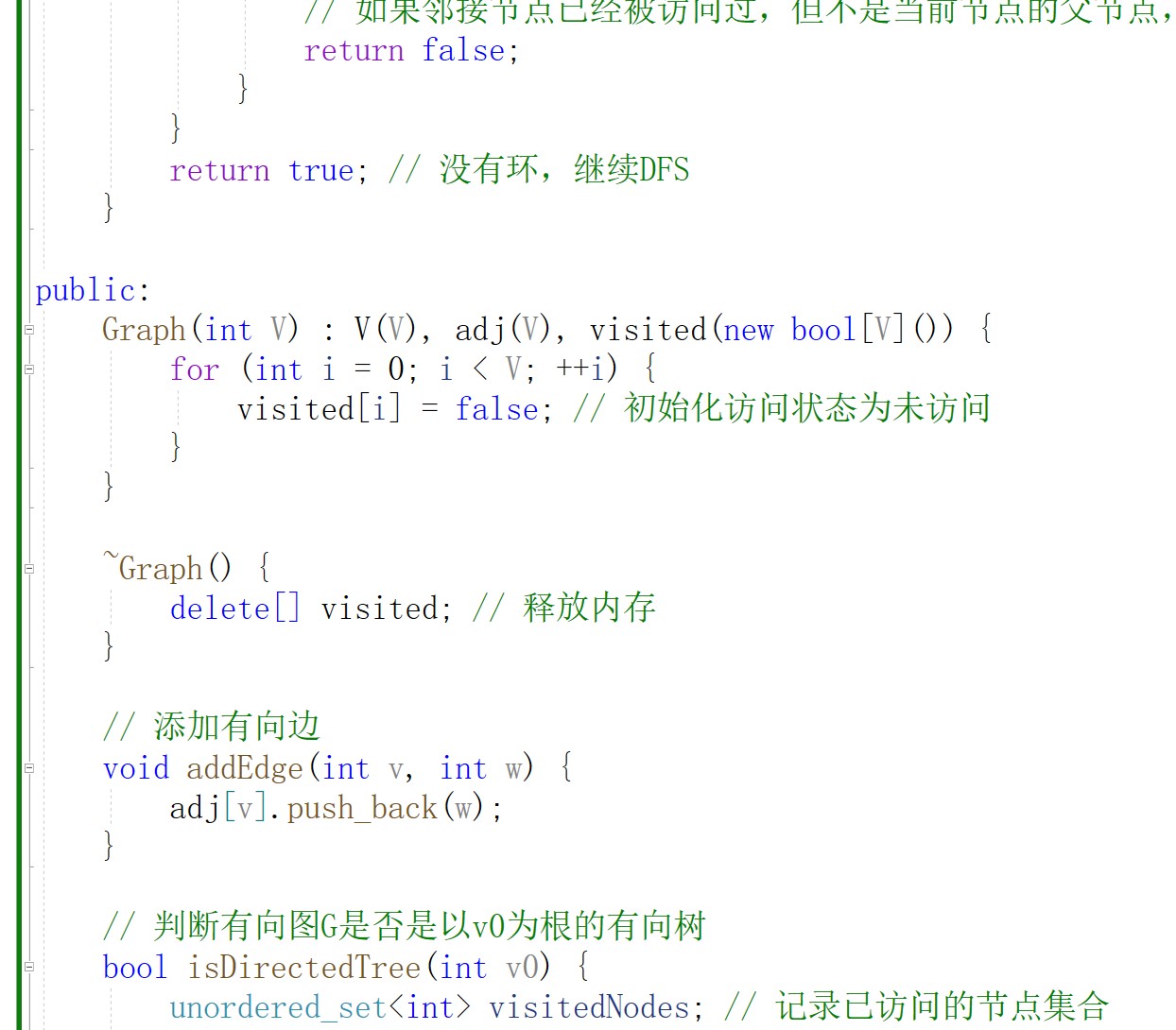


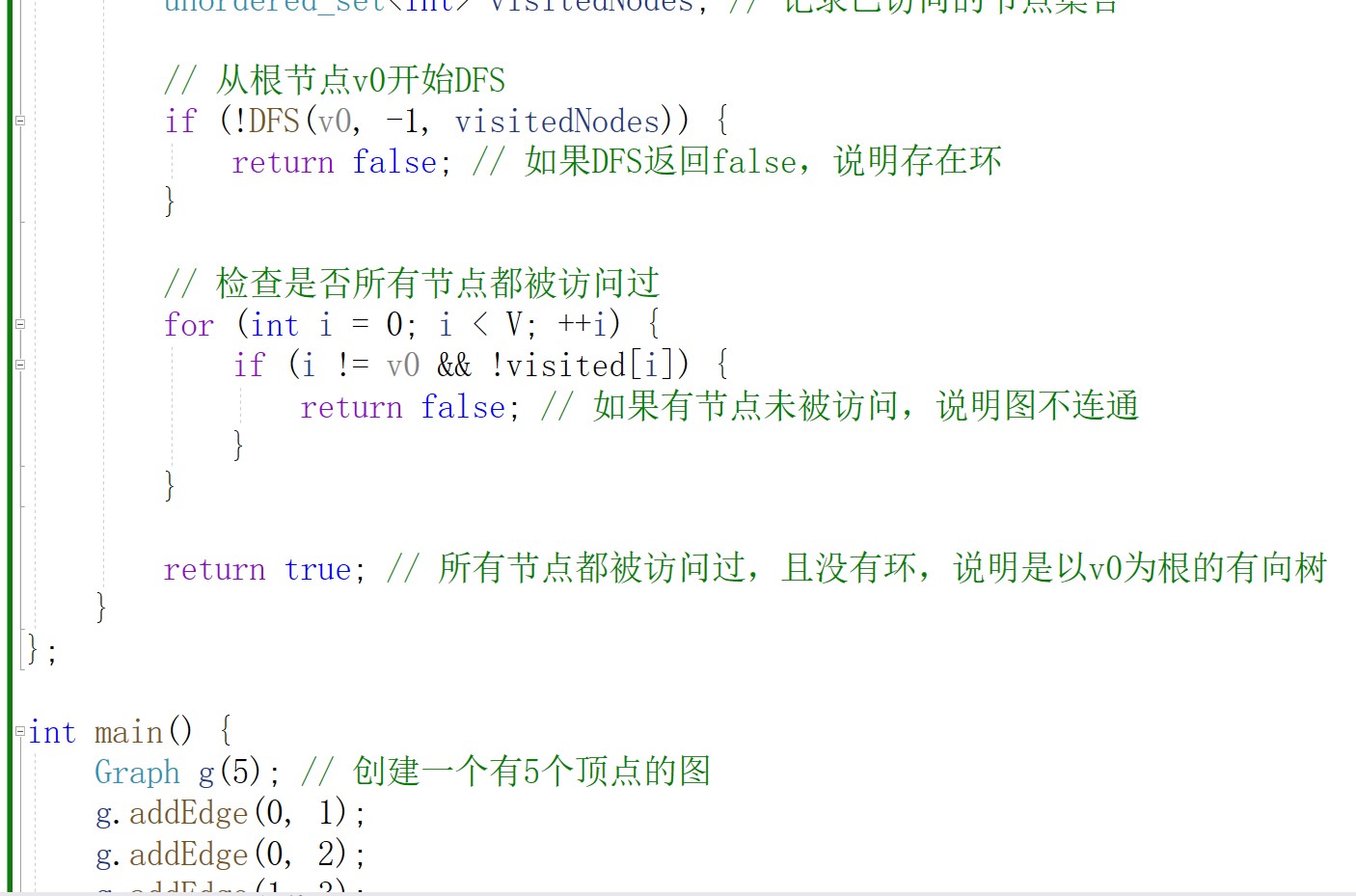


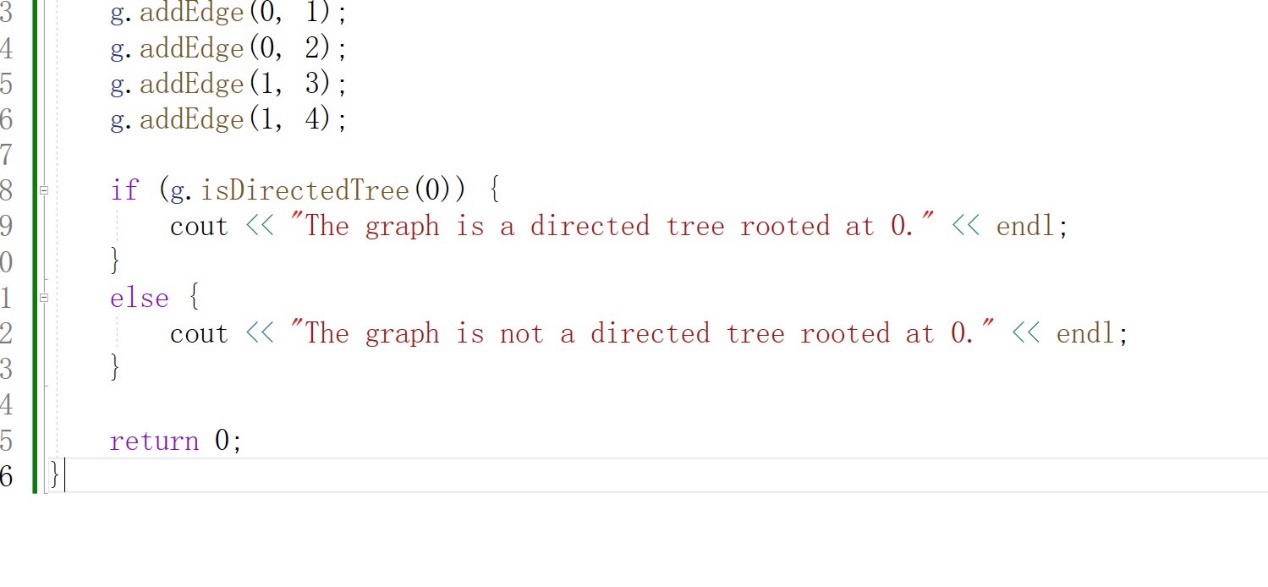


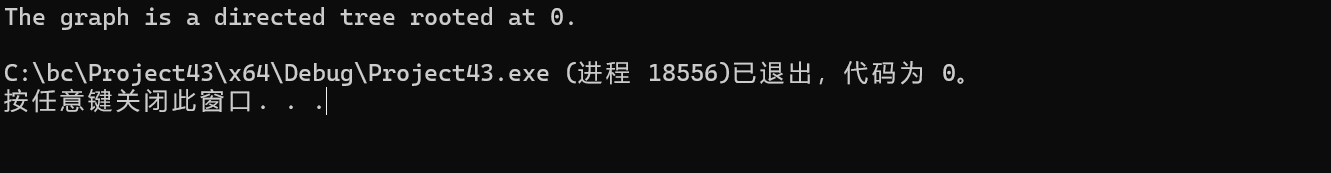
第十三题











第十四题

在图中，深度优先遍历（DFS）算法的时间复杂度主要取决于图的存储方式以及图的特性（特别是边的数量）。这里，我们将分析在邻接矩阵和邻接表存储方式下，深度优先遍历算法的时间复杂度。

**邻接矩阵**

在邻接矩阵中，图G用一个二维数组adjMatrix[V][V]来表示，其中V是顶点的数量。如果adjMatrix[i][j]为1（或true），则表示存在一条从顶点i到顶点j的边；如果为0（或false），则表示不存在这样的边。

**时间复杂度分析**：

1. **初始化**：创建一个大小为V的访问标记数组（或集合），并将所有元素初始化为未访问状态。这一步的时间复杂度是O(V)。
2. **DFS递归**：对于每个未访问的顶点，启动DFS递归过程。在递归过程中，我们需要检查当前顶点的所有邻接顶点。由于邻接矩阵提供了直接访问任意两个顶点之间是否存在边的信息，所以这一步对于每个顶点都需要O(V)的时间（检查所有可能的邻接顶点）。
3. **遍历所有顶点**：在最坏的情况下，我们需要遍历图中的每个顶点一次。因此，这一步的时间复杂度是O(V)。

**总时间复杂度**：  
由于我们需要遍历所有顶点，并且对于每个顶点，我们都要检查所有其他顶点（在邻接矩阵中），所以总的时间复杂度是O(V^2)。但是，如果图是稀疏的（即边的数量远小于V^2），那么实际上会有很多不必要的检查（因为大部分adjMatrix[i][j]都是0）。然而，从理论上看，即使图是稀疏的，时间复杂度仍然是O(V^2)，因为算法没有利用图的稀疏性。

**邻接表**

在邻接表中，图G用一个数组adjList[V]来表示，其中adjList[i]是一个列表，包含所有与顶点i相邻的顶点的索引。

**时间复杂度分析**：

1. **初始化**：与邻接矩阵相同，创建一个大小为V的访问标记数组（或集合），并将所有元素初始化为未访问状态。这一步的时间复杂度是O(V)。
2. **DFS递归**：对于每个未访问的顶点，启动DFS递归过程。在递归过程中，我们只需要检查当前顶点的邻接表中的所有顶点。由于邻接表只存储了与当前顶点相邻的顶点的信息，所以这一步的时间复杂度是O(E)，其中E是边的数量。
3. **遍历所有顶点**：同样地，在最坏的情况下，我们需要遍历图中的每个顶点一次。这一步的时间复杂度是O(V)。

**总时间复杂度**：  
由于我们只需要遍历所有顶点，并且对于每个顶点，我们只需要检查其邻接表中的顶点（在邻接表中），所以总的时间复杂度是O(V+E)。这个复杂度反映了图的稀疏性：如果图是稀疏的（即E远小于V^2），那么算法将运行得更快。

**结论**：

* 在邻接矩阵中，深度优先遍历的时间复杂度是O(V^2)，不依赖于图的稀疏性。
* 在邻接表中，深度优先遍历的时间复杂度是O(V+E)，这反映了图的稀疏性，并且在实际应用中通常更高效。

第十五题





