实验一

1. 实验题目

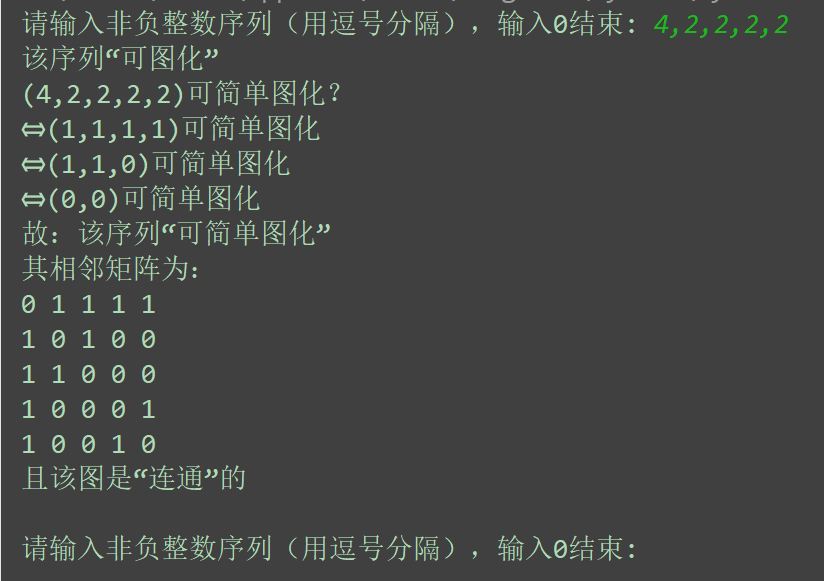
可图化、可简单图化、连通性的判别及图的矩阵表达

1. 实验目的
2. 掌握可图化的定义及判断方法；
3. 掌握可简单图化的定义及判断方法；
4. 掌握连通图的判断方法；
5. 掌握图的矩阵表达。
6. 实验要求
7. 给定非负整数序列（例如：(4,2,2,2,2)）。
8. 判断此非负整数序列是否是可图化的。
9. 请利用Havel定理判断此非负整数序列是否是可简单图化的，要求输出判断过程与结果。
10. 如果是可简单图化的，请输出该序列对应一个简单图的相邻矩阵，并判断该图是否连通。
11. 实验内容和实验步骤
12. 需求分析
13. 输入的形式和输入值的范围：

输入形式：例如4,2,2,2,2（可给定任意数量的序列）

输入值的范围：非负整数

1. 输出的形式：



如图所示，输出时，按照实验要求的顺序依次展开输出

─┬─不可图化

└─可图化─┬─不可简单图化

└─可简单图化─┬─不连通

**(相邻矩阵)** └─连通

1. 程序所能实现的功能：

* 能通过输入任意个数的非负整数序列判断是否可图化，是否可简单图化，并输出相邻矩阵和判断是否连通
* 可循环，反复输入，方便测试多组数据

1. 概要设计

我选择的是Python进行编程，

下面是

数据结构定义、主程序的流程及各模块之间的调用关系：

1. # 可图化
2. **def** Graphitization():
3. # 通过定理d1+d2+…+dn=0(mod 2)判断是否可图化
5. # 可简单图化
6. **def** Graphitization\_simple():
7. # 先通过n-1≥d1筛选判断
8. # 再通过Havel定理判断是否可简单图化
10. #相邻矩阵
11. **def** Adjacency\_matrix():
12. # 通过优化后的Havel-Hakimi 算法将序列转换成
13. # 相邻矩阵并输出
15. #连通
16. **def** Connected():
17. # 通过DFS深度优先搜索判断是否连通
19. #主函数
20. While True:
21. # 判断是否可图化
22. **if** Graphitization():
23. # “可图化”
24. # 判断是否可简单图化
25. **if** Graphitization\_simple():
26. # “可简单图化”
27. # 判断图是否连通
28. **if** Connected():
29. # “可连通”
30. **else**:
31. # “不可连通”
32. **else**:
33. # “不可简单图化”
34. **else**:
35. # “不可图化”
36. 详细设计

源代码：

1. # 可图化
2. **def** Graphitization(array):
3. sum = 0
4. **for** i **in** range(len(array)):
5. sum += array[i]
6. **if** sum % 2 == 0 **and** sum != 0:
7. **return** True
8. **return** False
10. # 可简单图化
11. **def** Graphitization\_simple(array):
12. a = sorted(array, reverse=True)     #降序排序
13. **if** a[0] >= len(array):
14. **print**("但不满足n-1>=d1")
15. **return** False
16. string = '('
17. **for** i **in** range(len(a)):
18. **if** i != len(a)-1:
19. string += str(a[i]) + ','
20. **else**:
21. string += str(a[i])
22. **print**(string+')可简单图化？')
23. **while** True:
24. head = a.pop(0)
25. **for** i **in** range(head):
26. a[i] -= 1
27. a = sorted(a, reverse=True)  # 降序排序
28. string = '⇔('
29. **for** i **in** range(len(a)):
30. **if** i != len(a) - 1:
31. string += str(a[i]) + ','
32. **else**:
33. string += str(a[i])
34. **print**(string + ')可简单图化')
35. **if** a.count(0) == len(a):
36. **return** True
37. **if** a.count(1) == 1 **and** a.count(0) == len(a)-1:
38. **return** False
39. **for** i **in** range(len(a)):
40. **if** a[i] < 0:
41. **return** False
43. # 二维列表排序
44. **def** sort\_two(a):
45. **for** i **in** range(len(a[0])):
46. **for** j **in** range(i, len(a[0])):
47. **if** a[0][i] < a[0][j]:
48. a[0][i], a[0][j] = a[0][j], a[0][i]
49. a[1][i], a[1][j] = a[1][j], a[1][i]
51. # 相邻矩阵
52. **def** Adjacency\_matrix(array, m):
53. a = [[0] \* len(array) **for** \_ **in** range(2)]
54. a[0] = sorted(array, reverse=True)     #降序排序
55. **for** i **in** range(len(array)):
56. a[1][i] = i
57. **for** i **in** range(len(array)):
58. **for** j **in** range(len(array)):
59. **for** k **in** range(len(array)):
60. **if** a[1][k] ==  i **and** a[0][k] > 0 **and** a[0][j] > 0 **and** i < a[1][j]:
61. m[i][ a[1][j] ] = 1
62. m[ a[1][j] ][i] = 1
63. a[0][j] -= 1
64. a[0][k] -= 1
65. sort\_two(a)
66. **print**("其相邻矩阵为：")
67. **for** i **in** range(len(array)):
68. **for** j **in** range(len(array)):
69. **print**(m[i][j], end=' ')
70. **print**()
72. # 连通
73. **def** Connected(m):
74. **def** dfs(node):
75. visited[node] = True
76. **for** neighbor, has\_edge **in** enumerate(m[node]):
77. **if** has\_edge **and** **not** visited[neighbor]:
78. dfs(neighbor)
79. n = len(m)
80. visited = [False] \* n
81. # 从第一个节点开始深度优先搜索
82. dfs(0)
83. # 如果所有节点都被访问到，则图是连通的
84. **return** all(visited)
86. # 主函数
87. **while** True:
88. # 从用户输入获取非负整数序列
89. cin = input("请输入非负整数序列（用逗号分隔），输入0结束: ")
91. **if** cin == '0':
92. **break**
94. **try**:
95. # 将输入的字符串转换为非负整数序列
96. array = list(map(int, cin.split(',')))
97. m = [ [0 **for** \_ **in** range(len(array))] **for** \_ **in** range(len(array)) ]
99. # 判断是否可图化
100. **if** Graphitization(array):
101. **print**("该序列“可图化”")
103. # 判断是否可简单图化
104. **if** Graphitization\_simple(array):
105. **print**("故：该序列“可简单图化”")
106. Adjacency\_matrix(array, m)
107. # 判断图是否连通
108. **if** Connected(m):
109. **print**("且该图是“连通”的")
110. **else**:
111. **print**("且该图是“不连通”的")
112. **else**:
113. **print**("故：该序列“不可简单图化”")
115. **else**:
116. **print**("该序列“不可图化”")
118. **print**()
120. **except** ValueError:
121. **print**("输入错误！请确保输入为非负整数序列，并使用逗号分隔。")
122. 调试分析
123. 调试过程中所遇到的问题及解决方法：

* 判断可图化时未考虑到全0的情况导致出错

解决方法：将sum!=0加入if判断

* 判断可简单图化时未考虑到n-1≥d1的情况

解决方法：已添加

* 判断连通时未熟练掌握DFS

解决方法：通过网上的素材借鉴理解学习

1. 算法的时空分析：

1. Graphitization(array)

时间复杂性：

* + **算法中的循环次数与输入数组的长度成正比，因此时间复杂性为 O(n)。**

空间复杂性：

* + **仅使用常量级别的额外空间，因此空间复杂性为 O(1)。**

2. Graphitization\_simple(array)

时间复杂性：

* + **算法中的排序操作为 O(n log n)，其中 n 为数组的长度。**
  + **主循环的时间复杂性主要取决于排序操作，因此总体时间复杂性为 O(n log n)。**

空间复杂性：

* + **除了输入数组外，仅使用了常量级别的额外空间，因此空间复杂性为 O(1)。**

3. sort\_two(a)

时间复杂性：

* + **该算法是选择排序，时间复杂性为 O(n^2)，其中 n 为二维列表中的列数。**

空间复杂性：

* + **仅使用常量级别的额外空间，因此空间复杂性为 O(1)。**

4. Adjacency\_matrix(array, m)

时间复杂性：

* + **该算法中的主循环中包含排序和嵌套循环，因此时间复杂性可能高达 O(n^3)，其中 n 为数组的长度。**

空间复杂性：

* + **除了输入数组和邻接矩阵外，仅使用了常量级别的额外空间，因此空间复杂性为 O(n^2)。**

5. Connected(m)

时间复杂性：

* + **深度优先搜索的时间复杂性为 O(V + E)，其中 V 为节点数，E 为边数。在这里，E 的最坏情况可能是 O(n^2)，因此总体时间复杂性为 O(n^2)。**

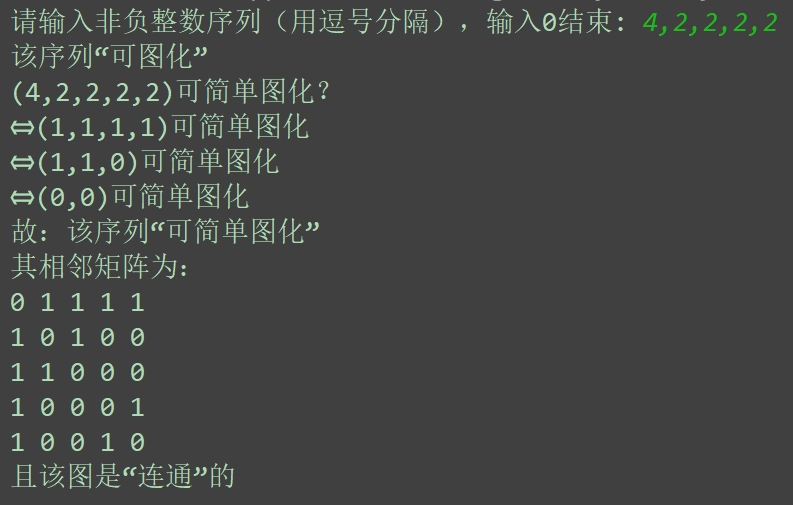
空间复杂性：

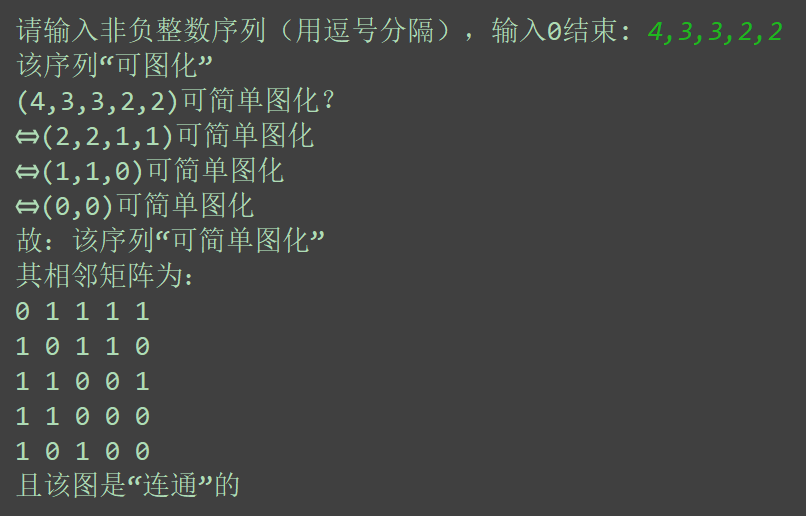
* + **除了输入的邻接矩阵和递归调用栈，仅使用了常量级别的额外空间，因此空间复杂性为 O(n)。**

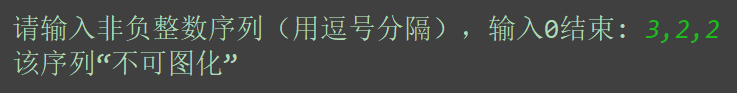
6. 总体时空复杂性：

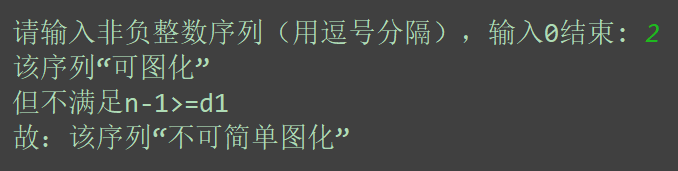
* + **主程序的循环次数取决于用户输入，可以忽略。**
* 这里的分析是基于每个算法的最坏情况。
* 实际性能可能会受到不同输入情况的影响。

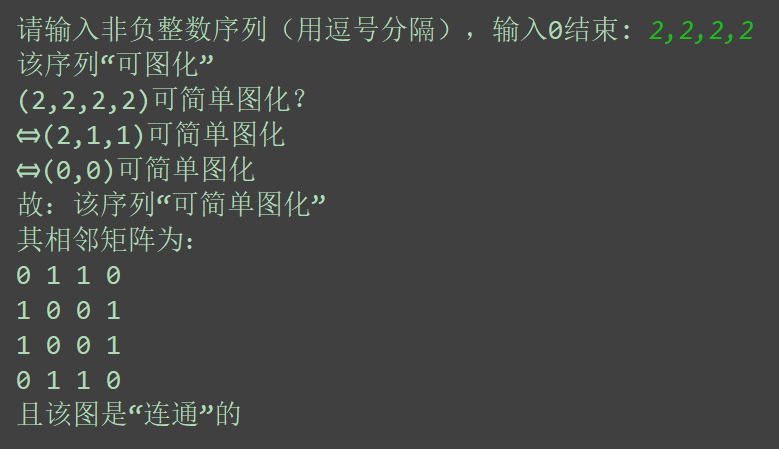
1. 实验结果

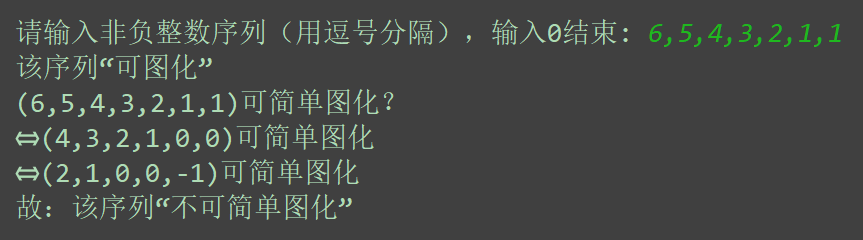


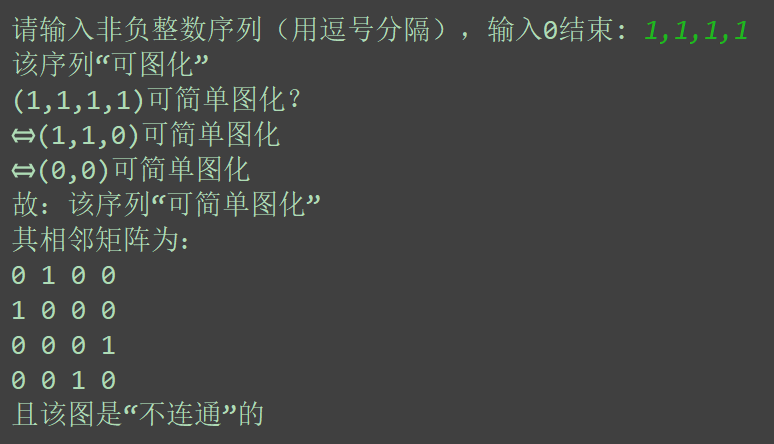












1. 实验总结

* 实验过程中的感悟和体会：

1. 图论知识的应用：

通过编写判断序列是否可图化、可简单图化的程序，深入理解了图论中的一些概念，例如图的连通性和邻接矩阵的应用。

1. 深度优先搜索的重要性：

实验中通过深度优先搜索判断图的连通性，深刻体会到深度优先搜索在图算法中的重要性。这种算法可以用来遍历图，查找路径，判断连通性等。

1. 函数模块化的优势：

将任务分解成不同的函数，每个函数负责一个明确的功能，提高了代码的可读性和可维护性。每个函数成为一个模块，更易于理解和测试。

1. 异常处理的重要性：

在用户输入时，通过异常处理机制可以有效地捕获错误，向用户提供友好的提示信息。这对于用户友好性和程序的稳定性都是至关重要的。

* 经验和教训：

1. 合理利用函数：

函数的使用使代码更为模块化和结构化，有助于提高代码的可读性。在实验中，函数的设计有助于更好地组织和管理代码。

1. 注重异常处理：

在用户输入部分，通过对异常的捕获和处理，提高了程序的健壮性。对于用户可能输入错误的情况，提供清晰的错误提示是良好实践。

1. 代码注释和输出信息：

充分利用注释和输出信息，有助于自己和其他人更好地理解代码的逻辑和执行流程。在输出信息中，提供详细和清晰的结果有助于用户理解程序运行的结果。

1. 持续学习和改进：

在实验中可能遇到一些挑战和问题，但通过不断学习和改进，能够更好地理解问题的本质，并提高解决问题的能力。

* 总结：

通过这个实验，不仅巩固了图论的相关知识，还提高了编写图算法的能力。在实践中，注重代码的可读性、异常处理和输出信息，是编写高质量代码的关键。在今后的学习和工作中，将继续保持学习的态度，不断提升编程技能。