



**实验报告： 循环赛日程表算法设计分析**

**学院：计算机学院（国家示范性软件学院）**

**专业： 计算机科学与技术**

**班级： 2022211305**

**学号： 2022211683**

**姓名： 张晨阳**

**2024年11月22号**

**目录**

[1. 实验概述 1](#_Toc183202405)

[1.1. 实验目的 1](#_Toc183202406)

[1.2. 实验内容及要求 1](#_Toc183202407)

[1.3. 实验环境 1](#_Toc183202408)

[2. 算法设计与实现 2](#_Toc183202409)

[2.1. 分治法问题分析 2](#_Toc183202410)

[2.1.1. 实例分析 2](#_Toc183202411)

[2.1.2. 普遍情况分析 3](#_Toc183202412)

[2.1.3. 分治算法逻辑推理 4](#_Toc183202413)

[2.2. 分治法算法思路 6](#_Toc183202414)

[2.3. 分治法算法实现 7](#_Toc183202415)

[3. 测试设计与结果 9](#_Toc183202416)

[3.1. 测试数据设计 9](#_Toc183202417)

[3.2. 测试结果展示 10](#_Toc183202418)

[4. 算法效率探索对比 13](#_Toc183202419)

[4.1. 分治法效率分析 13](#_Toc183202420)

[4.2. 回溯法效率对比 14](#_Toc183202421)

[4.2.1. 回溯法核心逻辑 14](#_Toc183202422)

[4.2.2. 效率分析 14](#_Toc183202423)

[4.2.3. 优缺点对比 15](#_Toc183202424)

[4.3. 多边形法效率对比 16](#_Toc183202425)

[4.3.1. 多边形法核心逻辑 16](#_Toc183202426)

[4.3.2. 效率分析 17](#_Toc183202427)

[4.3.3. 优缺点对比 17](#_Toc183202428)

[5. 总结心得 18](#_Toc183202429)

# 实验概述

## 实验目的

* 理解分治法的策略，掌握基于递归的分治算法的实现方法；
* 掌握基于数学模型建立算法模型的建模方法；
* 理解并掌握在渐进意义下的算法复杂性的评价方法。

## 实验内容及要求

1. **算法的设计与实现**

**问题描述：**

设有 n 个运动员要进行网球循环赛，设计一个满足下列条件的比赛日程表：

1）每个选手必须与其他 n - 1 个选手各赛一次；

2）每个选手一天只能赛一次

3）当 n 是偶数时，循环赛只能进行 n - 1 天

4）当 n 是奇数时，循环赛只能进行 n 天

**2. 实验内容**

依据数学方法，解决选手人数不等于 2k时，在偶数和奇数情况下，依题目条件建立算法模型。

① 数据生成： 不同规模的数据集，用于测试算法的正确性及效率。

② 算法实现：实现能够满足题目要求的循环赛日程表算法及程序。

## 实验环境

* Visual Studio Code
* C++ 17

# 算法设计与实现

## 分治法问题分析

### 实例分析

首先从具体的例子开始分析。我们首先列出9位运动员的循环赛表，并在此基础上开始分析。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **人/天** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| **1** | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| **2** | 1 | 5 | 3 | 7 | 4 | 8 | 9 | 10 | 6 |
| **3** | 8 | 1 | 2 | 4 | 5 | 9 | 10 | 6 | 7 |
| **4** | 5 | 9 | 1 | 3 | 2 | 10 | 6 | 7 | 8 |
| **5** | 4 | 2 | 10 | 1 | 3 | 6 | 7 | 8 | 9 |
| **6** | 7 | 8 | 9 | 10 | 1 | 5 | 4 | 3 | 2 |
| **7** | 6 | 10 | 8 | 2 | 9 | 1 | 5 | 4 | 3 |
| **8** | 3 | 6 | 7 | 9 | 10 | 2 | 1 | 5 | 4 |
| **9** | 10 | 4 | 6 | 8 | 7 | 3 | 2 | 1 | 5 |

上表中，每一行表示一名选手的比赛安排；每一列表示一天内的比赛对手。其中，选手10指的是该天轮空，没有比赛对手。

接下来，让我们从该例推广到普适的情况。

由上表知，其实9名选手安排的是10人的循环赛，不过不考虑第10人的比赛安排。那么，对于n人的循环赛：

当n为奇数时，我们安排n+1人的循环赛，其中与第n+1位选手比赛的表示轮空；

当n为偶数时，则直接使用分治法，考虑 位选手的赛程表。

### 普遍情况分析

根据上述思路，我们具体分析如何继续分治下去：

在完成 位选手的日程表后，我们的n位选手表此时已经完成了左上角和左下角的两个部分。其中左上角为选手1~ ，左下角为选手 的前 天的赛程安排。

并且左上角选手1~ 的赛程表加上 就得到的了左下角选手 的赛程表。

那么，我们将左上角的安排复制到对应的右下角，将左下角的安排复制到右上角，即可得到满足要求的n名选手的循环赛安排表。

所以可以得到：为设计9人循环赛，可以先设计10人循环赛。为设计10人循环赛，可以先设计5人循环赛。为设计5人循环赛，可以先设计6人循环赛。以此类推，设计3人、4人、2人循环赛。

### 分治算法逻辑推理

基于上述结论，我们先给出2人循环赛的安排表：

|  |  |
| --- | --- |
| **人/天** | **1** |
| **1** | 2 |
| **2** | 1 |

然后按照上面的思路（左下角为左上角，剩下两个角交叉复制），得到4人循环赛的安排：（此处用同样的颜色标记复制的内容）

|  |  |  |  |
| --- | --- | --- | --- |
| **人/天** | **1** | **2** | **3** |
| 1 | 2 | 3 | 4 |
| 2 | 1 | 4 | 3 |
| 3 | 4 | 1 | 2 |
| 4 | 3 | 2 | 1 |

在这张表的基础上，我们将选手4视为轮空，即可得到3位选手的循环赛安排表：

|  |  |  |  |
| --- | --- | --- | --- |
| **人/天** | **1** | **2** | **3** |
| **1** | 2 | 3 | 4 |
| **2** | 1 | 4 | 3 |
| **3** | 4 | 1 | 2 |

继续推理，下一个为6人的循环赛。

此时可以发现，奇数选手的安排表的复制方法与偶数略有不同：

对于左下部分，在把左上位置复制到左下并按相对位置加上 后，还需要处理奇数时的“轮空”选手。“轮空”选手在 后会超出选手的范围，需要单独处理。

我们这样考虑：原本轮空的比赛，应该选择一个后 的选手进行比赛，因为前 已经匹配完成。那么选手1匹配选手 1 ，以此类推。得到选手1对选手4，选手2对选手5，选手3对选手6.而对于超出的问题，加上即可。

即可完成**左上角和左下角**。我们继续考虑右侧的情况：

对于**右上部分**：前 个选手在前 天与后 名选手未进行比赛（除去第1 位选手。那么，我们给出这样的对应关系：

第i位选手在第 天（第列，因为第1列其实可以算作第0天，虽有点违背逻辑，但为了对应数组下标从0开始，不多赘述）对应第 位选手。举例来说，第1名选手在第4天（第5列）与第名选手比赛。

对于**右下部分**：因为右上部分的安排是前 对战后 选手，所以这里只需要将对应关系反过来就行。

至此，我们即可构建出6人循环赛表，也可以不断推出任意n选手的循环赛表。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **人/天** | **1** | **2** | **3** | **4** | **5** |
| **1** | 2 | 3 | 4 | 5 | 6 |
| **2** | 1 | 5 | 3 | 6 | 4 |
| **3** | 6 | 1 | 2 | 4 | 5 |
| **4** | 5 | 6 | 1 | 3 | 2 |
| **5** | 4 | 2 | 6 | 1 | 3 |
| **6** | 3 | 4 | 5 | 2 | 1 |

## 分治法算法思路

由上述分析可知，对于偶数和奇数的拓展（复制×2）的方式不同，所以需要实现两个函数分别处理。

对于奇数的拓展，由于不断需要前 和后 选手的对应，我们在引入一个辅助数组用于实现上述对应。

下面描述完整的算法思路：

若且为奇数，则先安排人循环赛。

若且为偶数，则先递归地安排 人循环赛。

在完成 人循环赛后，若为偶数，则执行偶数版本的函数将 人循环赛安排表拓展成n人循环赛。

若 为奇数，则执行奇数版本的函数将 人循环赛安排表拓展。

若*n*为2，则直接得到2人循环赛，递归结束。

## 分治法算法实现

**算法步骤**

1. **递归分解：**

* 如果选手人数为偶数，递归分解为一半人数的赛程；
* 如果选手人数为奇数，则增加一个虚拟选手，使人数变为偶数再递归处理。

1. **基础情况：**

* 当人数为 1 时，直接返回赛程表 schedule[1][1] = 1。

1. **合并子问题：**

* 当人数为偶数时，按照四象限复制法构造完整赛程：
  + 左上角直接继承子问题的赛程；
  + 右上角在左上角基础上加上 n/2；
  + 左下角复制右上角；
  + 右下角复制左上角。
* 当人数为奇数时，使用辅助数组记录虚拟选手轮空情况，并在赛程表中相应调整。

**核心代码实现**：以下是实现算法的核心代码部分，展示了递归构造赛程表和分治复制的逻辑：

1. // 递归生成赛程

2. void generateSchedule(int num) {

3. if (num == 1) { // 基础情况

4. schedule[1][1] = 1;

5. return;

6. }

7. if (num % 2 == 1) {

8. generateSchedule(num + 1); // 奇数人，补充一个轮空

9. return;

10. }

11. generateSchedule(num / 2); // 递归生成一半人数的赛程

12. copySchedule(num); // 根据一半赛程复制生成完整赛程

13. }

14.

15. // 偶数选手的复制逻辑

16. void handleEven(int num) {

17. int half = num / 2;

18. for (int i = 1; i <= half; i++) {

19. for (int j = 1; j <= half; j++) {

20. schedule[i][j + half] = schedule[i][j] + half; // 右上角

21. schedule[i + half][j] = schedule[i][j + half]; // 左下角

22. schedule[i + half][j + half] = schedule[i][j]; // 右下角

23. }

24. }

25. }

26.

27. // 奇数选手的复制逻辑

28. void handleOdd(int num) {

29. int half = num / 2;

30. // 计算轮空选手对应情况

31. for (int i = 1; i <= half; i++) {

32. byePlayers[i] = half + i;

33. byePlayers[half + i] = byePlayers[i];

34. }

35. for (int i = 1; i <= half; i++) {

36. for (int j = 1; j <= half + 1; j++) {

37. if (schedule[i][j] > half) { // 当前选手轮空

38. schedule[i][j] = byePlayers[i];

39. schedule[half + i][j] = (byePlayers[i] + half) % num;

40. } else { // 非轮空情况

41. schedule[half + i][j] = schedule[i][j] + half;

42. }

43. }

44. for (int j = 2; j <= half; j++) {

45. schedule[i][half + j] = byePlayers[i + j - 1];

46. schedule[byePlayers[i + j - 1]][half + j] = i;

47. }

48. }

49. }

50.

51. // 复制赛程

52. void copySchedule(int num) {

53. if (num / 2 > 1 && (num / 2) % 2 == 1)

54. handleOdd(num);

55. else

56. handleEven(num);

57. }

# 测试设计与结果

## 测试数据设计

为了验证分治法生成循环赛程表的算法正确性和性能，我设计了以下测试数据，覆盖基础功能、边界条件、大规模输入以及特殊输入场景。

测试数据详细如下：

**基础测试**

* 输入 10：测试偶数人数，验证简单赛程表的生成。
* 输入 5：测试奇数人数，验证轮空选手的处理逻辑是否正确。

**边界测试**

* 输入 1：测试单人赛程生成，验证最小输入的正确性。
* 输入 2：测试最小偶数人数，验证简单赛程表的生成。
* 输入 3：测试最小奇数人数，验证轮空选手的处理逻辑是否正确。
* 输入 30：验证在打印阈值时的赛程表生成是否正确。

**大规模测试**

* 输入 1000：测试中等规模参赛人数的性能。
* 输入 10000：测试大规模参赛人数的性能。

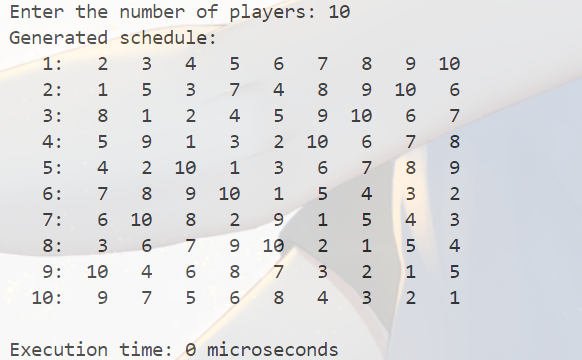
**特殊测试**

* 输入 16：测试为的参赛人数，验证分治法的深度递归能力。

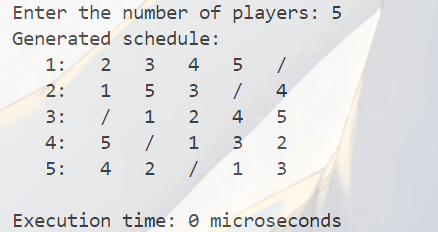
## 测试结果展示

**1. 基础测试**

* **输入 10**  
  程序生成完整的偶数人数赛程表，验证正确性。输出如下：

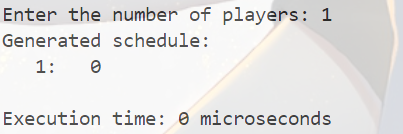


* **输入 5**  
  程序生成奇数人数赛程表，验证轮空选手的正确性。输出如下：

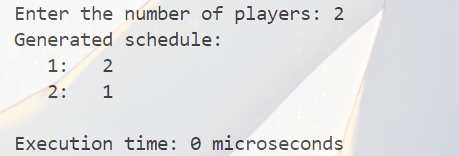


**2. 边界测试**

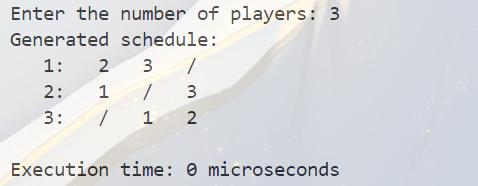
* **输入 1**  
  程序生成单人赛程表，结果为：



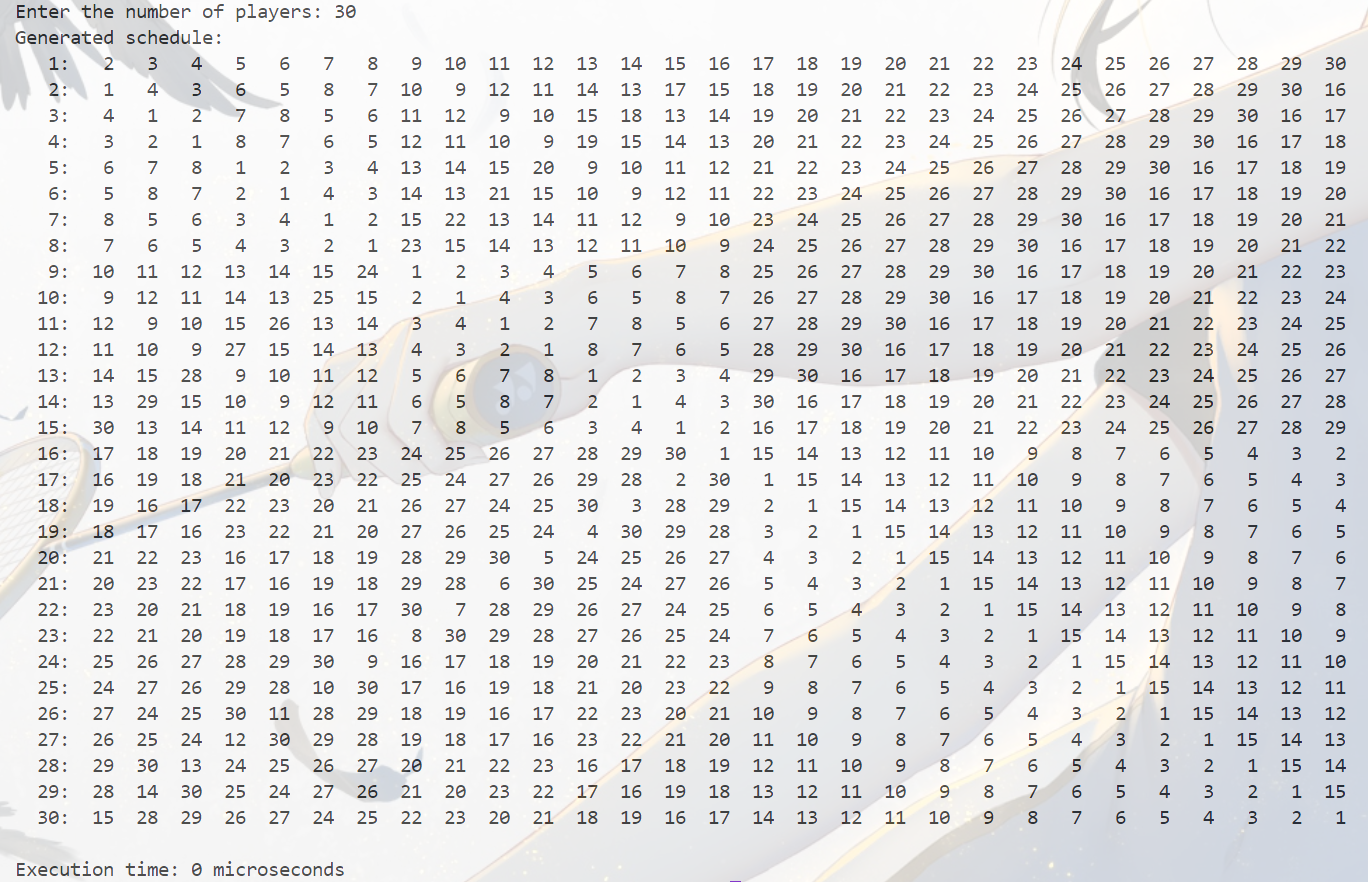
* **输入 2**  
  程序生成最小偶数人数赛程表，结果为：



* **输入 3**  
  程序生成最小奇数人数赛程表，结果为：



* **输入 30**  
  程序完整打印赛程表：



**3. 大规模测试**

* **输入 1000**  
  程序未打印赛程表，仅输出执行时间：

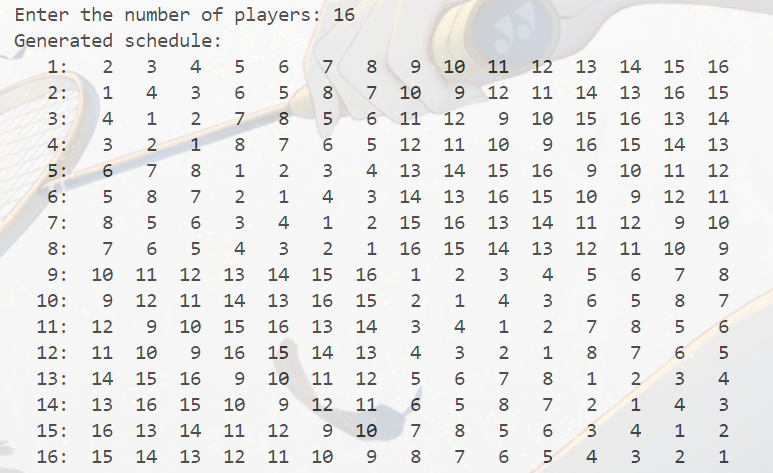


* **输入 10000**  
  程序未打印赛程表，仅输出执行时间：



**4. 特殊测试**

* **输入 16**



# 算法效率探索对比

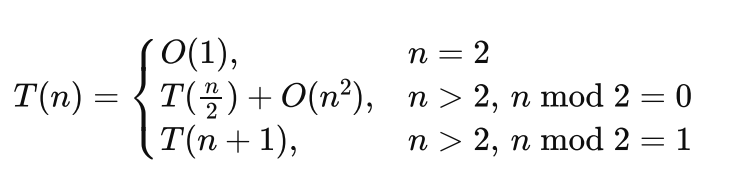
## 分治法效率分析

**时间复杂度分析**

分治法的核心步骤包括：

1. **递归拆分问题**：将 n 个选手划分为两组，每组大小为，递归处理子问题。
2. **合并结果**：通过复制逻辑扩展子问题的解，生成完整的赛程表。

假设生成 n 人的赛程表所需时间为 ，则递归关系可以表示为：



解此递归方程可得.即时间复杂度为

**空间复杂度分析**

分治法生成赛程表的空间开销主要由以下两部分组成：

1. **存储赛程表**：赛程表是一个 的二维数组，所需空间为 。
2. **递归调用栈**：分治法的递归深度为 ，每次递归的额外空间为常数级，因此递归栈空间为 。

综合来看，分治法的空间复杂度为 。

## 回溯法效率对比

我们将从两种方法的时间复杂度、空间复杂度进行对比分析。

### 回溯法核心逻辑

回溯法采用逐步尝试的方式生成赛程表，通过递归尝试安排每名选手的对手，当某个安排不满足条件时，回溯到上一步重新尝试。

**回溯法伪代码**：

1. function backtrack(round, player):

2. if round > totalRounds:

3. return true # 所有轮次完成

4.

5. if player > totalPlayers:

6. return backtrack(round + 1, 1) # 下一轮比赛

7.

8. for opponent in range(1, totalPlayers + 1):

9. if isValid(round, player, opponent): # 检查比赛安排是否有效

10. schedule[round][player] = opponent

11. schedule[round][opponent] = player

12. markAsUsed(player, opponent)

13.

14. if backtrack(round, player + 1):

15. return true

16.

17. undoMark(player, opponent) # 回溯

18.

19. return false

### 效率分析

**时间复杂度**

**1. 回溯法**

* 每轮比赛中，每名选手需要尝试与其他 名选手比赛，递归深度为总轮次 。
* 在最坏情况下，可能需要遍历所有可能的安排组合，总复杂度为 。
* 实际上，由于加入了剪枝策略，时间复杂度通常低于理论上的 ，但仍然随着输入规模迅速增长。

**2. 分治法**

* 分治法将问题拆分为两个子问题，并在每个子问题解决后合并结果。
* 时间复杂度为

**空间复杂度**

1. **回溯法**

* 需要维护赛程表和递归调用栈：
  + 赛程表：二维数组，空间复杂度为 。
  + 递归调用栈：深度为 ，每层需要存储比赛安排，空间复杂度为 。
  + 总空间复杂度为 。

1. **分治法**

* 需要存储赛程表（）和递归调用栈（深度为 ，每层为常数开销），总空间复杂度为 。

### 优缺点对比

|  |  |  |
| --- | --- | --- |
| **比较维度** | **分治法** | **回溯法** |
| **时间复杂度** |  |  |
| **空间复杂度** |  |  |
| **可扩展性** | 适合大规模输入，性能稳定 | 随人数增加，计算量迅速增长，不适合大规模 |
| **实现复杂性** | 需要递归分解和合并子问题 | 实现相对简单，但需要考虑剪枝优化 |
| **性能表现** | 中等规模和大规模下表现优异 | 小规模问题表现尚可，但扩展性较差 |
| **适用场景** | 适合大规模赛程表生成，计算效率高 | 适合用于验证小规模问题的解法正确性 |

## 多边形法效率对比

多边形法是一种基于规则和规律的直接生成循环赛程表的算法，通过固定一个选手，然后轮转其余选手位置来完成赛程安排。

### 多边形法核心逻辑

1. **初始化选手位置**：将参赛选手依次排列成多边形顶点。
2. **轮转生成对战**：

* 每轮比赛中，第一个选手固定；
* 其余选手顺时针或逆时针轮转，形成新一轮的对战组合。

1. **虚拟选手处理**：如果参赛人数为奇数，则添加一个虚拟选手，用于标记轮空。

代码实现如下：

1. void makeTable(int num) {

2.     int days = num - (num + 1) % 2;  // 比赛的天数

3.     cout << "\nTournament Schedule:\n";

4.     for (int i = 0; i < days; i++) {

5.         cout << "Day " << (i + 1) << ": ";

6.         int len = num - 1;

7.         for (int j = 0; j < num / 2; j++) {

8.             cout << ((i + j) % num + 1) << " vs "

9.                  << ((i + j + len) % num + 1);

10.             if (j < num / 2 - 1)

11.                 cout << ", ";  // 添加逗号分隔

12.             len -= 2;

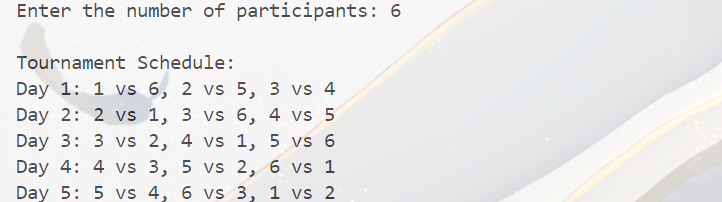
13.         }

14.         cout << endl;  // 换行，进入下一天

15.     }

16. }

运行截图如下：



### 效率分析

**时间复杂度**

1. **多边形法**

* 每一轮比赛中，需要安排所有选手的对手，复杂度为 ；
* 总轮次为 （或 个有效比赛天数）；
* 因此总时间复杂度为 。

**2. 分治法**

* 时间复杂度为

**空间复杂度**

1. **多边形法**

* 需要存储完整的赛程表，空间复杂度为 ；
* 辅助变量（如当前轮次选手位置等）为常数级开销，整体空间复杂度为 。

**2. 分治法**

* 需要存储赛程表（）和递归调用栈（深度为 ，每层为常数开销），总空间复杂度为 。

### 优缺点对比

|  |  |  |
| --- | --- | --- |
| **维度** | **多边形法** | **分治法** |
| **时间复杂度** |  |  |
| **空间复杂度** |  |  |
| **实现复杂性** | 简单直接，逻辑清晰，容易实现 | 需要递归分解和合并，逻辑稍复杂 |
| **性能表现** | 小规模和中等规模下表现优异 | 小规模表现相当，大规模下更稳定 |
| **适用场景** | 适合规则简单的循环赛程表生成 | 适合更复杂的赛程规则和大规模生成 |

# 总结心得

在本次研究与实现循环赛程表生成算法的过程中，我探索了多种算法，包括分治法、回溯法和多边形法，并深入对比了它们在效率、适用场景和扩展性方面的优劣。

这不仅帮助我更全面地理解了循环赛程表生成问题的本质，也强化了我对算法设计和优化的能力。

分治法在处理大规模参赛人数时，性能优势尤为明显。通过将问题逐步拆分为更小的子问题，分治法有效降低了计算复杂度，使得即使在上千名选手的场景下，也能保持较好的运行时间和稳定性。

相比之下，多边形法的实现更加直观和简单。它利用几何规律直接生成赛程表，不需要复杂的递归操作，在小到中等规模的输入场景中表现非常优异。然而，多边形法由于其固定的规则和逻辑，缺乏灵活性，不适合处理更复杂的需求，也难以在超大规模的输入下保持性能的稳定。

通过此次研究，我不仅掌握了三种算法的实现方式，还锻炼了分析算法复杂度的能力。我认识到，选择合适的算法需要综合考虑问题规模、规则复杂性和执行效率等多个维度，而算法设计的核心在于权衡简单性与高效性的最佳结合。

总之，本次项目不仅加深了对算法设计的理解，也提升了我从问题分析到实现优化的全流程能力。