实验二循环赛日程表算法的设计与分析

班级：2022211304

姓名：赵宇鹏

学号：2022211119

一、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。

二、实验内容及要求

1. 算法的设计与实现

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2^k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。

3. 实验要求

设计测试数据集，编写测试程序，用于测试：

a) 正确性：所实现算法的正确性；

b) 算法复杂性：分析评价各个算法在算法复杂性上的表现；（最差情

况、平均情况）

三、问题分析

本次实验要求基于递归的分治算法对n个运动员进行比赛日程表的设计，在课程上，老师已经对特殊情况$n = 2^n$的解题思路进行了说明：

按分治策略，将所有的选手分为两半，n个选手的比赛日程表就可以通过为n/2个选手设计的比赛日程表来决定。递归地用对选手进行分割，直到只剩下2个选手时，比赛日程表的制定就变得很简单。这时只要让这2个选手进行比赛就可以了。

1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

由这张表我们可以看出当遇到$n = 2^n$时，日程表的左上部分与右下部分是完全相同的，左下部分与右上部分也是相同的，而左下部分又等于左上部分向下平移并加上 n / 2，那么对于$n! = 2^n$的情况，我们也可以利用这个思路去进一步求解：

在这里，我们可以先以3人循环赛为例：

1	2	3	0
2	1	0	3
3	0	1	2

这是三人循环赛的赛程表，第1列表示选手编号，后面每一列都表示选手在当天的对手，可以看到对于奇数个人进行循环赛，必定会有人轮空，那么就可以进行拓展，假设一个虚拟人物`4`，轮空既是与`4`对战，那么循环赛日程表如下：

1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

由此，可以看到4人循环赛其实就是3人循环赛的拓展，而对偶数人的循环赛则以6人循环赛为例：

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	5	6	1	3
6	3	4	5	2	1

可以看到对偶数人次的循环赛日程表，原来$n = 2^n$时的思路可以完全沿用，由此，我们可以对问题进行推广：当我们需要安排 n 人循环赛时，若 n 为奇数，我们可以安排 n + 1 人的循环赛，日程表中若有人与第 n + 1 号人进行比赛，其实并非真的比赛而表示轮空。若 n 为偶数，则可以将所有选手分为两半，n 个选手的比赛日程表可以通过设计 n / 2 人的循环赛日程表决定，当设计完 n / 2 人的循环赛日程表后：

- 若 n / 2 为偶数，则左上角保持不变，左下角则按照原来思路将左上角元素加上 n / 2 得到左下角部分，右上角则为左下角的复制，右下角为左上角的复制，总体思路和$n = 2^n$时相同。
- 当 n / 2 为奇数时，情况较为复杂，不能就单纯进行复制了，因为在 n / 2 为奇数时日程表中会有轮空情况，对轮空的地方，我们也需要安排选手进行对决，这时后面新加进来的 n / 2 个选手还没有进行排表，所以我们可以直接将后面新加进来的选手直接放到轮空位即可，我们将原有选手号加上 n / 2 即可得到不重复的轮空填充，这样我们就完成了对初始 n/2 表的更新，接下来继续更新左下角，若已分配对手（之前轮空匹配）则不变，否则按照原来的规则在原左上角的基础上加上 n / 2 ，这样就完成了左下角的分配，然后由于 n / 2 是奇数，所以表合并前有 n + 1 列（包括第一列选手列），那么对从第 n / 2 + 2 列开始是没有进行排表的，对右上角我们只需排第 n / 2 + 2 列到第 n 列，而由于这一行其他位置都已填好，所以我们只需要对剩余的 n / 2 - 1 个未匹配选手循环匹配对手即可，右下角则可以根据前三部分的匹配填充好了，通过3人制和6人制日程表的比对，我们可以发现完全符合这个流程。

通过上述步骤，我们就可以实现对任意 n 人制网球循环赛日程表的规划了：若 n 为奇数，则为其添加一个虚拟选手求 n + 1 人制的日程表，然后再求 (n + 1) / 2 人制的日程表，以此递归直到 n 为 1，然后再进行复制合并，最后输出日程表即可。

四、程序设计说明

1. 全局变量

```
int a[MAX][MAX]; //a[i][j]表示第i个人第j天比赛的对手
int b[MAX]; //b[i]表示第i个人的轮空对手
```

2. 打印输出模块

代码如下：

```
void print(int n) {
    //偶数时直接输出
    if(n % 2 == 0) {
        for(int i = 1; i <= n; i++) {
            for(int j = 1; j <= n; j++)
                cout << a[i][j] << "  ";
            cout << endl;
        }
        return;
    } else { //奇数时必有人轮空
        for(int i = 1; i <= n; i++) {
            for(int j = 1; j <= n + 1; j++){
                if(a[i][j] >= n + 1)    cout<< "空" << " ";
                else    cout << a[i][j] << "  ";
            }
            cout << endl;
        }
```

```
    }
}
```

在程序中我设置的日程表数组大小是大于实际需求的，所以输出需要对数组范围进行限制，人数为偶数时，输出 n * n 矩阵，人数为奇数时，输出 n * n + 1 矩阵即可。

3. 日程表合并模块

代码如下：

```
//偶数时直接复制
void even(int n) {
    int m = n / 2;
    for(int i = 1; i <= m; i++) {
        for(int j = 1; j <= m; j++) {
            a[i + m][j + m] = a[i][j];//右下角
            a[i + m][j] = a[i][j] + m;//左下角
            a[i][j + m] = a[i + m][j];//右上角
        }
    }
}
//奇数时把轮空的和后面的进行匹配
void odd(int n) {
    int m = n / 2;

    for(int i = 1; i <= m; i++) {
        b[i] = i + m;
        b[m + i] = i + m;//实现循环
    }
    for(int i = 1; i <= m; i++) {
        for(int j = 1; j <= m + 1; j++) {
            if(a[i][j] > m) {
                a[i][j] = b[i];
                a[m + i][j] = i;
            }
            else
            a[m + i][j] = a[i][j] + m;//左下角
        }
        for(int j = 1; j <= m - 1; j++) {
            a[i][m + 1 + j] = b[i + j];//右上角
            a[b[i + j]][m + j + 1] = i;//右下角
        }
    }
}
```

具体原理已在问题分析中说明，不再做过多描述。

4. 分治模块

```
void decide(int n) {
    if(n / 2 > 1 && ((n / 2) % 2 == 1)) {
        odd(n);
    } else  even(n);
}
```

```
6
7  void merge(int n) {
8      if(n == 1) {
9          a[1][1] = 1;
10         return;
11     }
12     if(n % 2 == 1) {
13         merge(n + 1);
14         return;
15     }
16     merge(n / 2);
17
18     decide(n);
19 }
```

通过对人数的不同执行相应的奇偶数分治策略并最后归并起来。

五、 基于其他算法的探索实现

1. 旋转多边形法

算法思路

网球循环赛是两两一组互为对手且对手不能重复的，所以旋转多边形法就把这些选手看作一个多边形，每个选手为多边形的一个顶点，若 n 为偶数，则每天选择相对的两组顶点进行匹配，若 n 为奇数，则加上一个虚拟选手，然后执行偶数策略，每天对多边形进行旋转来匹配新的对手，在我的代码中，则是选择将选手设为一个循环数组，然后首尾两两配对，配对完成后则对数组进行移动，在此基础上，由于选手的号码其实就是一个顺序数组，所以移动可以优化为对数字进行加减操作后进行模运算，从而避免每天对整个选手数组的移动，程序比较简单，因此不对其进行说明，代码如下：

```
1  #include <stdio.h>
2  #include <iostream>
3  using namespace std;
4
5  int schedule[20000][20000];
6
7  void generateScheduleMatrix(int n) {
8      int isOdd = n % 2;
9      int m = isOdd ? n + 1 : n;// 若为奇数，则相当于多一个虚拟选手
10
11     for (int day = 0; day < m - 1; ++day) {
12         for (int i = 0; i < m / 2; ++i) {
13             int p1 = (day + i) % (m - 1);
14             int p2 = (m - 1 - i + day) % (m - 1);
15             if (i == 0) p2 = m - 1;
16
17             if (p1 < n && p2 < n) {
18                 schedule[p1][day] = p2 + 1;
19                 schedule[p2][day] = p1 + 1;
20             }
21         }
22     }
23
24 }
```

```
25
26  void print(int n) {
27      int isOdd = n % 2;
28      int m = isOdd ? n + 1 : n;
29      for (int i = 0; i < n; ++i) {
30          printf("%3d ", i + 1);
31          for (int j = 0; j < m - 1; ++j) {
32              if(schedule[i][j] == 0) {
33                  printf("%3d ", 0);
34              } else {
35                  printf("%3d ", schedule[i][j]);
36              }
37          }
38          printf("\n");
39      }
40  }
41
42  int main() {
43      int n;
44      scanf("%d", &n);
45      generateScheduleMatrix(n);
46      print(n);
47      return 0;
48  }
49
```

2. 回溯法

算法思路

回溯法定义了一个二维数组`schedule[i][j]`表示 i 号选手在第 j 天的对手，再用一个二维布尔数组`used[i][j]`表示选手 i 和选手 j 是否已经比赛过，通过对每个选手的每天比赛日程进行遍历，如果没有对手则按照顺序给其分配对手，有的话就跳过，直到分配完毕，若安排的比赛导致后续安排失败，则进行回溯，撤销这一次安排，尝试其他可能性，具体代码如下：

```
1   #include <iostream>
2   #include <vector>
3   #include <iomanip>
4   using namespace std;
5
6   vector<vector<int>> schedule; // 比赛日程表
7   vector<vector<bool>> used;    // 记录两人是否已经比赛过
8   int n, totalDays;
9
10  // 检查某天两个选手是否可以比赛
11  bool canPlay(int p1, int p2, int day) {
12      return !used[p1][p2] && !used[p2][p1] && schedule[p1][day] == 0 &&
    schedule[p2][day] == 0;
13  }
14
15  bool assignMatch(int day, int pairIndex) {
16  if (day >= totalDays) return true;
17      if (pairIndex >= n / 2) {
```

```
        return assignMatch(day + 1, 0);
    }

    // 尝试为当前天分配比赛对
    for (int p1 = 1; p1 <= n; p1++) {
        if (schedule[p1][day] != 0) continue;

        for (int p2 = p1 + 1; p2 <= n; p2++) {
            if (!canPlay(p1, p2, day)) continue;

            // 分配比赛
            schedule[p1][day] = p2;
            schedule[p2][day] = p1;
            used[p1][p2] = used[p2][p1] = true;

            // 递归分配下一对
            if (assignMatch(day, pairIndex + 1)) return true;

            // 回溯
            schedule[p1][day] = 0;
            schedule[p2][day] = 0;
            used[p1][p2] = used[p2][p1] = false;
        }
    }
    return false;
}

void printSchedule(bool isOdd) {
    if(isOdd) {
        for (int i = 1; i <= n - 1; i++) {
            cout << setw(3) << i << "  ";
            for (int j = 0; j < totalDays; j++) {
                if(schedule[i][j] == n) {
                    cout << setw(3) << "0" << "  ";
                } else {
                    cout << setw(3) << schedule[i][j] << "  ";
                }
            }
            cout << endl;
        }
    }
    else {
        for (int i = 1; i <= n; i++) {
            cout << setw(3) << i << "  ";
            for (int j = 0; j < totalDays; j++) {
                cout << setw(3) << schedule[i][j] << "  ";
            }
            cout << endl;
        }
    }
}

int main() {
    cin >> n;
    bool isOdd = (n % 2 == 1);
```

```
    if (isOdd) n++; // 引入虚拟选手
    totalDays = n - 1;

    // 初始化数据结构
    schedule = vector<vector<int>>(n + 1, vector<int>(totalDays, 0));
    used = vector<vector<bool>>(n + 1, vector<bool>(n + 1, false));

    if (assignMatch(0, 0)) {
        printSchedule(isOdd);
    } else {
        cout << "No solution" << endl;
    }

    return 0;
}
```

六、算法复杂度分析

1. 基于递归的分治算法

在分治算法中，`merge`函数是递归实现的核心，其作用是若输入为奇数则为其加一，若输入为偶数则进行分治二分，因此递归的深度约为$log_2 n$，而对`even`函数和`odd`函数，里面都存在二层循环，循环范围都大约为`1-m`，所以复杂度大约为$O(m^2)$，其中$m = n/2$，则复杂度为$O(\frac{n^2}{4})$，由于`merge`的递归深度为$log_2 n$，每一层递归调用`decide`处理的复杂度为 $O(\frac{n^2}{4})$，打印输出函数的时间复杂度为$O(n^2)$，由于基于递归的分治算法的计算是步骤是相对固定的，所以最差情况和最好情况时间复杂度几乎相同，所以总的时间复杂度为：

$$T(n) = O(n^2) + O(\frac{n^2}{4}) + O(\frac{n^2}{8}) + \ldots + O(1)$$

所以$T(n) = O(n^2)$

2. 旋转多边形法

对旋转多边形法，由于只有`generateScheduleMatrix`函数和`print`函数，显然，`generateScheduleMatrix`函数的时间复杂度为$O(\frac{n^2}{4})$，而`print`函数的时间复杂度为$O(n^2)$

所以旋转多边形法的时间复杂度为：$T(n) = O(n^2) + O(\frac{n^2}{4}) = O(n^2)$

3. 回溯法

`assignMatch(day, pairIndex)` 的递归每次尝试为某一天的某个选手对分配比赛。对于每一天，`pairIndex` 表示已经分配的对数。

递归的终止条件是 `day >= totalDays`，表示所有比赛日的对阵已经完成。总共需要处理 n/2 对选手，一共有 n 天（偶数为 n - 1，奇数为 n,此处近似为 n），因此递归深度为：

$$\text{递归深度} = O(n * \frac{n}{2}) = O(\frac{n^2}{2}) \approx O(n^2)$$

对内部循环有

```
for (int p1 = 1; p1 <= n; p1++) {
    for (int p2 = p1 + 1; p2 <= n; p2++) {
        ...
    }
}

```

外层循环运行 $O(n)$次，内层循环运行 $O(n)$次，因此两层循环的总复杂度为：$O(n^2)$，对于最坏情况，`assignMatch` 函数会尝试所有可能的比赛组合。对于每一天，它会尝试所有可能的选手对组合，这意味着时间复杂度是 $O(n^2)$。由于总共有 totalDays 天，因此最坏情况下的时间复杂度是 $O((n^2)^{totalDays})$，所以其时间复杂度为：$T(n) = O((n^2)^{totalDays})$，若遇到2的幂次数的输入，则时间复杂度为$T(n) = O(n^2) * O(n^2) = O(n^4)$

七、测试程序及算法正确性检验

测试程序会分别调用三个网球循环赛算法文件并进行编译，输入相同的输入并检查其输出，对输出，首先会检查每一行及每一列选手的唯一性，即对每一行和每一列，都不存在相同的选手，否则代表有选手和其他选手比了两次比赛或在同一天和两个人比赛了，说明错误，然后对每一行，检查选手之间的对手是否对应，不会出现第一天1号选手的对手是2号选手，2号选手的对手却是三号选手的情况，核心代码如下：

```
# 验证输出
def validate_output(output, n):
    lines = output.split('\n')
    schedule = []
    for line in lines:
        if line.strip():
            schedule.append([int(x) if x.isdigit() else 0 for x in
line.split()])

    # 检查行数是否正确
    if len(schedule) != n:
        print(f"Invalid number of rows: expected {n}, got {len(schedule)}")
        return False

    # 检查每一行的长度和唯一性
    expected_columns = n if n % 2 == 0 else n + 1
    for row in schedule:
        if len(row) != expected_columns:
            print(f"Invalid row length: expected {expected_columns}, got
{len(row)}")
            return False
        seen = set()
        for num in row:
            if num != 0:  # 忽略 "空" 的位置
                if num in seen or num < 1 or num > n:
                    print(f"Invalid number in row: {num}")
                    return False
                seen.add(num)

    # 检查每一列的唯一性
    for col in range(expected_columns):
        seen = set()
```

```
        for row in schedule:
            num = row[col]
            if num != 0:  # 忽略 "空" 的位置
                if num in seen or num < 1 or num > n:
                    print(f"Invalid number in column: {num}")
                    return False
                seen.add(num)

    # 检查对称性
    for i in range(n):
        for j in range(expected_columns):
            if schedule[i][j] != 0:
                opponent = schedule[i][j]
                if schedule[opponent - 1][j] != i + 1:
                    print(f"Invalid match: schedule[{i + 1}][{j}] =
{opponent}, but schedule[{opponent}][{j}] = {schedule[opponent - 1][j]}")
                    return False

    return True
```

八、测试结果输出及说明

注：在报告中均省略了算法输出的日程表，详细测试输出情况文件中的output内容

测试一

由于回溯算法的时间复杂度过高，因此对其测试小数据量的输入，在本次输入中我们输入19进行测试

输出结果为

```
Program 1 produced valid output.
Program 2 produced valid output.
Program 3 produced valid output.
Program 1 took 0.043362 seconds.
Program 2 took 0.034448 seconds.
Program 3 took 4.017570 seconds.
```

测试二

输入特殊情况2的幂次数来对算法进行检验，输入256

输出为

```
Program 1 produced valid output.
Program 2 produced valid output.
Program 3 produced valid output.
Program 1 took 0.038720 seconds.
Program 2 took 0.033776 seconds.
Program 3 took 0.182902 seconds.
```

可以看到虽然数据量更大了，但是各算法的运行效率都有了提升，消耗时间更短了，回溯法的时间也显著减少。

测试三（从此测试开始不对回溯法进行测试）

输入50进行测试

输出：

```
Program 1 produced valid output.
Program 2 produced valid output.
Program 1 took 0.035819 seconds.
Program 2 took 0.037942 seconds.
```

可以看到两个算法的执行时间相近，符合计算的时间复杂度，算法正确性得到检验。

测试四

本次测试为最后一个测试，在前面的三个测试中已经检验了算法的正确性，本次测试则意在测试算法的效率，输入为`1023,1024,1025,8192,8193,8194`，覆盖了奇数，普通偶数以及2的幂次数，由于输出过多，我关闭了矩阵输出，所以算法无法检验其正确性，仅能检测执行时间，以下是执行结果：

```
Testing for n = 1023:
------------------------------
Output from Program 1:


Invalid number of rows: expected 1023, got 0
Program 1 produced invalid output.
Output from Program 2:


Invalid number of rows: expected 1023, got 0
Program 2 produced invalid output.
Program 1 took 0.048346 seconds.
Program 2 took 0.046507 seconds.

Testing for n = 1024:
------------------------------
Output from Program 1:


Invalid number of rows: expected 1024, got 0
Program 1 produced invalid output.
Output from Program 2:


Invalid number of rows: expected 1024, got 0
Program 2 produced invalid output.
Program 1 took 0.038804 seconds.
Program 2 took 0.037741 seconds.

Testing for n = 1025:
------------------------------
Output from Program 1:


Invalid number of rows: expected 1025, got 0
Program 1 produced invalid output.
Output from Program 2:


```

```
Invalid number of rows: expected 1025, got 0
Program 2 produced invalid output.
Program 1 took 0.038853 seconds.
Program 2 took 0.042320 seconds.

Testing for n = 8192:
------------------------------
Output from Program 1:


Invalid number of rows: expected 8192, got 0
Program 1 produced invalid output.
Output from Program 2:


Invalid number of rows: expected 8192, got 0
Program 2 produced invalid output.
Program 1 took 0.619529 seconds.
Program 2 took 1.009715 seconds.

Testing for n = 8193:
------------------------------
Output from Program 1:


Invalid number of rows: expected 8193, got 0
Program 1 produced invalid output.
Output from Program 2:


Invalid number of rows: expected 8193, got 0
Program 2 produced invalid output.
Program 1 took 0.738181 seconds.
Program 2 took 1.051771 seconds.

Testing for n = 8194:
------------------------------
Output from Program 1:


Invalid number of rows: expected 8194, got 0
Program 1 produced invalid output.
Output from Program 2:


Invalid number of rows: expected 8194, got 0
Program 2 produced invalid output.
Program 1 took 0.728577 seconds.
Program 2 took 1.030646 seconds.
```

结果作为表格呈现为：

数据规模/时间	1023	1024	1025	8192	8193	8194

数据规模/时间	1023	1024	1025	8192	8193	8194
分治	0.048346	0.038804	0.038853	0.619529	0.738181	0.728577
旋转	0.046507	0.037741	0.042320	1.009715	1.051771	1.030646

可以看到，当数据规模变大后，分治的效率会逐渐大于旋转多边形法。

九、实验总结

本次实验我实现了使用基于递归的分治算法，旋转多边形法以及回溯法来解决网球循环赛问题，在老师课上提醒的方法下苦思冥想才把奇偶情况抽象出来进行讨论，通过本次实验，我已基本学会了递归分治算法的使用，收获颇丰。