题目详解

循环赛问题

题目分析

这个问题的目的是设计一个围棋比赛的循环赛程表，使得每位选手都能与其他所有选手比赛一次，且每天只能比赛一次。这个问题的解决方案采用了分治策略，即将问题分解为更小的子问题，然后将子问题的解决方案组合起来形成原问题的解决方案。

在这个问题中，我们首先为第一位选手安排比赛日程，然后将剩余的选手分为多个块，每个块的大小是2的幂。对于每个块，我们复制前一行的数据，但是会根据块的ID进行偏移。这样，我们就可以确保每位选手都能与其他所有选手比赛一次，且每天只能比赛一次。

这个问题的意义在于，它展示了如何使用分治策略来解决复杂的组合设计问题。通过将问题分解为更小的子问题，我们可以更容易地找到解决方案。此外，这个问题也展示了如何使用二维数组来存储和操作数据。

设计流程

这段代码的设计流程如下：

1. 首先，定义一个函数 `problem1`，接受一个默认参数 `n`，默认值为8，表示参赛者的数量。
2. 计算比赛日程表的大小，即 `2^n`。
3. 初始化一个二维数组 `arr`，用于存储比赛日程表。数组的每一行代表一个参赛者，每一列代表一天。
4. 填充数组的第一行，即第一位选手的比赛日程。
5. 使用嵌套循环来填充数组的其余部分。在这个过程中，使用了一种称为"分块"的策略，即将数组分为多个块，并在每个块中复制前一行的数据，但是会根据块的ID进行偏移。
6. 打印出比赛日程表。每一行的数据代表一个选手的比赛日程，即他们每天将与哪位选手比赛。

代码分析

```
void problem1(int n = 8) {
    // 这一行是用来计算比赛的天数的，比如如果有8个人参加比赛，那么就需要2的8次方，也就是256
天来完成比赛
    int sche = static_cast<int>(std::pow(2.0, n));

    // 这一行是创建一个二维数组，用来存储每个人每天的比赛安排
    std::vector<std::vector<int>> arr(sche, std::vector<int>(sche));
```

```
    int bw = 1; // 这个是用来表示每个小块的宽度的，一开始是1
    int bid; // 这个是用来表示每个小块的编号的
    int c_offset, r_offset; // 这两个是用来表示列偏移和行偏移的

    // 这个循环是用来填充第一行的，也就是第一个人的比赛安排
    for (int i = 0; i < sche; i++) {
        arr[0][i] = i + 1;
    }

    // 这个循环是用来填充剩下的行的，也就是其他人的比赛安排
    for (int j = 0; j < n; j++) {
        for (int r = 0; r < bw; r++) {
            for (int c = 0; c < sche; c++) {
                // 这一行是计算每个小块的编号
                bid = (c + bw) / bw;
                // 这两行是计算列偏移和行偏移
                c_offset = static_cast<int>(std::pow(-1.0, bid + 1)) * bw;
                r_offset = bw;
                // 这一行是复制上一行的数据，但是会根据小块的编号进行偏移
                arr[r + r_offset][c + c_offset] = arr[r][c];
            }
        }
        // 这一行是将每个小块的宽度翻倍
        bw = bw * 2;
    }

    // 这一部分是用来打印比赛安排的
    std::cout << "N\\D\t";
    for (int i = 0; i < sche - 1; i++) {
        std::cout << i + 1 << "\t";
    }
    std::cout << "\n";
    for (const auto &row: arr) {
        for (const auto &elem: row) {
            std::cout << elem << "\t";
        }
        std::cout << "\n";
    }
}
```

加工规划问题

题目分析

这个问题的目的是找到一种分配10个零件到两台机器上的方式，使得所有零件都被处理完的总时间最短。这是一个经典的调度问题，可以通过动态规划来解决。

动态规划是一种用于解决最优化问题的数学方法。它的基本思想是将一个复杂的问题分解成一系列简单的子问题，并将子问题的解存储起来，以便在解决更大的问题时可以重复使用。

1. **定义问题**：我们有两台机器和一组任务，每个任务在每台机器上的处理时间是已知的。我们的目标是找到一种任务分配策略，使得所有任务都被处理完的总时间最短。
2. **定义状态**：我们定义一个二维数组 `p`，其中 `p[i][k]` 表示第一台机器处理前 `k` 个任务并且总时间为 `i` 的情况下，第二台机器需要的最短时间。
3. **初始化状态**：我们初始化 `p[i][0]` 为0，表示当没有任务时，两台机器都不需要任何时间。
4. **状态转移**：对于每个任务 `k`，我们有两种选择：将它分配给第一台机器，或者将它分配给第二台机器。如果我们将它分配给第一台机器，那么 `p[i][k]` 就等于 `p[i - a[k]][k - 1]`，其中 `a[k]` 是任务 `k` 在第一台机器上的处理时间。如果我们将它分配给第二台机器，那么 `p[i][k]` 就等于 `p[i][k - 1] + b[k]`，其中 `b[k]` 是任务 `k` 在第二台机器上的处理时间。我们选择这两种情况中的最小值作为 `p[i][k]` 的值。
5. **找到最优解**：我们遍历所有可能的 `i`，找出使得两台机器的最大工作时间最小的 `i`。这个 `i` 就是我们要找的最优解。
6. **回溯找出具体任务**：我们从最优解开始，回溯找出每台机器处理的具体任务。如果 `p[i][k]` 等于 `p[i - a[k]][k - 1]`，那么我们知道任务 `k` 是分配给第一台机器的，否则它是分配给第二台机器的。我们将这个过程重复，直到找出所有任务的分配情况。

这个解决思路的关键在于使用动态规划来遍历所有可能的任务分配方式，并通过状态转移来找出最优解。这种方法可以有效地解决这种类型的调度问题。

设计流程

这段代码的设计流程主要分为以下几个步骤：

1. **定义命名空间和常量**：首先，代码定义了一个命名空间 `scheduling`，以及一个常量 `doubleMax`，这个常量表示的是 `double` 类型能表示的最大值。
2. **定义函数 `getMinTimeWithTaskRecord`**：这个函数的目的是找到最优的分配策略。它接受四个参数：两个表示每个零件在每台机器上的处理时间的向量 `a` 和 `b`，以及两个用于存储每台机器处理的零件的向量 `A` 和 `B`。
3. **计算总时间**：函数首先计算出所有零件在第一台机器上的总处理时间 `sumA`。
4. **初始化动态规划数组**：然后，函数初始化一个二维数组 `p`，用于存储中间结果。
5. **填充动态规划数组**：接着，函数使用两个嵌套循环来填充数组 `p`。在这个过程中，它计算出每种可能的任务分配方式的总时间。

6. **找到最优解**：然后，函数遍历所有可能的 `i`，找出使得两台机器的最大工作时间最小的 `i`。这个 `i` 就是我们要找的最优解。
7. **回溯找出具体任务**：最后，函数使用回溯的方式找出每台机器处理的具体任务，并将它们存储在向量 `A` 和 `B` 中。
8. **定义函数 `problem2`**：这个函数的目的是调用 `getMinTimeWithTaskRecord` 函数，并打印出结果。它首先定义了两个表示每个零件在每台机器上的处理时间的向量 `time1` 和 `time2`，以及两个用于存储每台机器处理的零件的向量 `A` 和 `B`。然后，它调用 `getMinTimeWithTaskRecord` 函数，并打印出最短的总时间，以及每台机器处理的零件的顺序。

这个设计流程的关键在于使用动态规划来遍历所有可能的任务分配方式，并通过状态转移来找出最优解。这种方法可以有效地解决这种类型的调度问题。

代码分析

```
namespace scheduling {
    // Define the maximum value for a double
    const double doubleMax = std::numeric_limits<double>::max();

    // Function to get the minimum time with task record
    double getMinTimeWithTaskRecord(const std::vector<double>& a, const
std::vector<double>& b, std::vector<int>& A, std::vector<int>& B) {
        // Get the number of tasks
        auto n = a.size();

        // Calculate the total time for machine A
        double sumA = 0;
        for (int i = 0; i < n; i++) {
            sumA += a[i];
        }

        // Initialize the dynamic programming table
        std::vector<std::vector<double>> p(static_cast<int>(sumA) + 1,
std::vector<double>(n + 1, 0));

        // Fill the dynamic programming table
        for (int k = 0; k < n; k++) {
            for (int i = static_cast<int>(sumA); i >= 0; i--) {
                // Calculate the time for machine B
                p[i][k + 1] = p[i][k] + b[k];
                // Check if machine A can handle the task
                if (i >= a[k])
                    // Update the time for machine B
                    p[i][k + 1] = std::min(p[i][k + 1], p[i - static_cast<int>
(a[k])][k]);
            }
```

```
        }

        // Initialize the minimum time
        double ret = doubleMax;
        int i_; // A's time spent currently

        // Find the minimum time
        for (int i = 0; i ⩽ static_cast<int>(sumA); i++) {
            double temp = std::max(static_cast<double>(i), p[i][n]);
            if (temp < ret) {
                ret = temp;
                i_ = i;
            }
        }

        // Backtrack to find the tasks for each machine
        for (int k = n; k ⩾ 1; k--) {
            double tmp = p[i_][k - 1] + b[k - 1];
            if (i_ ⩾ a[k - 1] && p[i_ - static_cast<int>(a[k - 1])][k - 1] ⩽
tmp) {
                A.insert(A.begin(), k);
                i_ -= static_cast<int>(a[k - 1]);
            } else
                B.insert(B.begin(), k);
        }

        // Return the minimum time
        return ret;
    }
}

// Test the function
void problem2(){
    // Define the time for each task on each machine
    std::vector<double> time1 = {1, 3, 5, 7, 4, 2, 8, 4, 9, 9};
    std::vector<double> time2 = {1.5, 2, 6, 8, 2, 1, 3, 8, 11, 4};

    // Initialize the tasks for each machine
    std::vector<int> A, B;

    // Print the minimum time
    std::cout << scheduling::getMinTimeWithTaskRecord(time1, time2, A, B) <<
std::endl;

    // Print the tasks for machine A
    std::cout << "A's task:\n";
```

```
        for(int &i: A)
            std::cout << i << " ";

        // Print the tasks for machine B
        std::cout << "\nB's task:\n";
        for(int &i: B)
            std::cout << i << " ";
        std::cout << "\n";
    }
```

最小整数

题目分析

这个问题的目的是找出一种策略，通过删除给定数字中的k个数字，使得剩下的数字组成的新数最小。这个问题的意义在于，它可以帮助我们理解和掌握如何使用贪心算法和数据结构（如双端队列）来解决实际问题。

在这个问题中，我们使用了一个贪心策略：每次都删除当前数字的前一个比它大的数字，这样可以保证删除k个数字后得到的新数最小。我们使用双端队列来存储当前的数字，并在需要删除数字时从队列的后端删除。

这个问题的实际应用可能包括数字优化、资源分配、经济决策等领域，其中需要通过删除或减少某些元素来优化结果。

设计流程

这段代码的设计流程如下：

1. 初始化一个双端队列 `dq` 来存储当前的数字。
2. 遍历输入的数字字符串 `num`，对于每一个字符（数字）：

   2.1. 如果队列不为空，且我们还可以删除数字（ `k > 0` ），且队列尾部的数字大于当前数字，那么我们就删除队列尾部的数字（ `dq.pop_back()` ），并将可以删除的数字数量减一（ `--k` ）。

   2.2. 将当前数字添加到队列尾部（ `dq.push_back(digit)` ）。
3. 如果队列不为空，且我们还可以删除数字，那么我们就继续删除队列尾部的数字，直到不能再删除。
4. 初始化一个空字符串 `res` 来存储结果，以及一个标志位 `leadingZero` 来标记是否遇到前导零。
5. 遍历队列中的数字，对于每一个数字：

   5.1. 如果当前是前导零，那么我们就跳过这个数字。

5.2. 否则，我们就将这个数字添加到结果字符串中，并将标志位设置为 `false`，表示已经不再是前导零。

6. 如果结果字符串为空，那么我们就返回"0"，否则返回结果字符串。

这个设计流程的主要思想是使用贪心算法和双端队列，每次都删除当前数字的前一个比它大的数字，这样可以保证删除k个数字后得到的新数最小。

代码分析

```
std::string problem3(const std::string &num, int k) {
    // 创建一个双端队列来存储数字
    std::deque<char> dq;

    // 遍历输入的数字
    for (char digit: num) {
        // 当队列不为空，我们还可以删除数字，且队列尾部的数字大于当前数字时
        // 我们就删除队列尾部的数字，并将可以删除的数字数量减一
        while (!dq.empty() && k > 0 && dq.back() > digit) {
            dq.pop_back();
            --k;
        }
        // 将当前数字添加到队列尾部
        dq.push_back(digit);
    }

    // 如果队列不为空，且我们还可以删除数字，那么我们就继续删除队列尾部的数字
    while (!dq.empty() && k > 0) {
        dq.pop_back();
        --k;
    }

    // 创建一个空字符串来存储结果
    std::string res = "";
    // 创建一个标志位来标记是否遇到前导零
    bool leadingZero = true;

    // 遍历队列中的数字
    for (char digit: dq) {
        // 如果当前是前导零，那么我们就跳过这个数字
        if (leadingZero && digit == '0') continue;
        // 否则，我们就将这个数字添加到结果字符串中，并将标志位设置为false，表示已经不再是前
导零
        leadingZero = false;
        res += digit;
    }
```

```
        // 如果结果字符串为空，那么我们就返回"0"，否则返回结果字符串
        return res.empty() ? "0" : res;
    }
```

数独问题

题目分析

这个问题的目的是使用回溯算法来解决数独问题。数独是一种著名的逻辑填字游戏，玩家需要根据游戏规则在9x9的网格中填入数字1-9。规则包括：每一行、每一列以及每一个3x3的子网格（共9个）中的数字1-9都不能重复。

这个问题的意义在于，它可以帮助我们理解和掌握如何使用回溯算法来解决实际问题。回溯算法是一种试探性的解决问题方法，它尝试各种可能的解决方案，直到找到满足条件的解决方案或者已经尝试了所有可能的解决方案。

设计流程

这段代码的设计流程如下：

1. 定义一个常量N，表示数独的大小（9x9）。
2. 定义一个函数 `isSafe`，用于检查在指定的行、列填入一个数字是否安全（即满足数独的规则）。这个函数首先检查同一行和同一列是否已经有相同的数字，然后检查3x3的子网格是否已经有相同的数字。
3. 定义一个函数 `solveSudoku`，用于解决数独问题。这个函数使用回溯算法，对于每一个空格，尝试填入1-9中的一个数字，然后调用 `isSafe` 函数检查是否满足数独的规则。如果满足，就继续填充下一个空格。如果不满足或者已经填充了所有空格，就回溯到上一个空格，尝试填入下一个数字。
4. 定义一个函数 `solveSudoku`，用于解决数独问题。这个函数首先复制一份数独网格，然后调用 `solveSudoku` 函数尝试解决数独问题。如果解决成功，就返回解决后的数独网格，否则返回一个空的optional。
5. 在 `problem4` 函数中，定义一个数独网格，然后调用 `solveSudoku` 函数尝试解决数独问题。如果解决成功，就打印解决后的数独网格，否则打印"No solution exists"。

这个设计流程的主要思想是使用回溯算法来解决数独问题。对于每一个空格，我们尝试填入1-9中的一个数字，然后检查是否满足数独的规则。如果满足，我们就继续填充下一个空格。如果不满足或者已经填充了所有空格，我们就回溯到上一个空格，尝试填入下一个数字。

代码分析

```
namespace sodoku {
    constexpr int N = 9;   // 定义数独的大小为9x9
```

```
    // 检查在指定的行、列填入一个数字是否安全（即满足数独的规则）
    bool isSafe(std::vector<std::vector<int>> &board, int row, int col, int
num) {
        // 检查同一行是否已经有相同的数字
        for (int x = 0; x <= 8; x++)
            if (board[row][x] == num)
                return false;

        // 检查同一列是否已经有相同的数字
        for (int x = 0; x <= 8; x++)
            if (board[x][col] == num)
                return false;

        // 检查3x3的子网格是否已经有相同的数字
        int startRow = row - row % 3, startCol = col - col % 3;
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++)
                if (board[i + startRow][j + startCol] == num)
                    return false;

        return true;
    }

    // 使用回溯算法解决数独问题
    bool solveSudoku(std::vector<std::vector<int>> &board, int row, int col) {
        // 如果已经填充了所有空格，就返回true
        if (row == N - 1 && col == N)
            return true;

        // 如果已经填充了当前行的所有空格，就转到下一行
        if (col == N) {
            row++;
            col = 0;
        }

        // 如果当前空格已经填充了数字，就跳过这个空格
        if (board[row][col] > 0)
            return solveSudoku(board, row, col + 1);

        // 尝试填入1-9中的一个数字
        for (int num = 1; num <= N; num++) {
            // 如果填入这个数字是安全的，就填入这个数字
            if (isSafe(board, row, col, num)) {
                board[row][col] = num;
                // 如果能够成功填充剩下的空格，就返回true
                if (solveSudoku(board, row, col + 1))
```

```
                    return true;
                }
                // 如果填入这个数字后不能成功填充剩下的空格，就删除这个数字（即回溯）
                board[row][col] = 0;
            }
            return false;
        }

        // 尝试解决数独问题
        std::optional<std::vector<std::vector<int>>>
solveSudoku(std::vector<std::vector<int>> board) {
            // 如果能够成功解决数独问题，就返回解决后的数独网格
            if (solveSudoku(board, 0, 0))
                return board;
            // 否则返回一个空的optional
            else
                return std::nullopt;
        }
}

void problem4() {
        std::vector<std::vector<int>> board = {
                {3, 0, 6, 5, 0, 8, 4, 0, 0},
                {5, 2, 0, 0, 0, 0, 0, 0, 0},
                {0, 8, 7, 0, 0, 0, 0, 3, 1},
                {0, 0, 3, 0, 1, 0, 0, 8, 0},
                {9, 0, 0, 8, 6, 3, 0, 0, 5},
                {0, 5, 0, 0, 9, 0, 6, 0, 0},
                {1, 3, 0, 0, 0, 0, 2, 5, 0},
                {0, 0, 0, 0, 0, 0, 0, 7, 4},
                {0, 0, 5, 2, 0, 6, 3, 1, 0}
        };
        auto res = sodoku::solveSudoku(board);
        if (!res) {
            std::cout << "No solution exists";
            return;
        }
        for (const auto &row: res.value()) {
            for (const int &elem: row) {
                std::cout << elem << "\t";
            }
            std::cout << "\n";
        }

}
```

实验结果

```cpp
int main() {
    problem1(4);
    std::cout << "----------------------------------------\n";
    problem2();
    std::cout << "----------------------------------------\n";
    std::cout << problem3("1432219", 4) << std::endl;
    std::cout << "----------------------------------------\n";
    problem4();
    return 0;
}
```

```
❱ ./algo
N\D 1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
2   1   4   3   6   5   8   7   10  9   12  11  14  13  16  15
3   4   1   2   7   8   5   6   11  12  9   10  15  16  13  14
4   3   2   1   8   7   6   5   12  11  10  9   16  15  14  13
5   6   7   8   1   2   3   4   13  14  15  16  9   10  11  12
6   5   8   7   2   1   4   3   14  13  16  15  10  9   12  11
7   8   5   6   3   4   1   2   15  16  13  14  11  12  9   10
8   7   6   5   4   3   2   1   16  15  14  13  12  11  10  9
9   10  11  12  13  14  15  16  1   2   3   4   5   6   7   8
10  9   12  11  14  13  16  15  2   1   4   3   6   5   8   7
11  12  9   10  15  16  13  14  3   4   1   2   7   8   5   6
12  11  10  9   16  15  14  13  4   3   2   1   8   7   6   5
13  14  15  16  9   10  11  12  5   6   7   8   1   2   3   4
14  13  16  15  10  9   12  11  6   5   8   7   2   1   4   3
15  16  13  14  11  12  9   10  7   8   5   6   3   4   1   2
16  15  14  13  12  11  10  9   8   7   6   5   4   3   2   1
----------------------------------------
20
A's task:
1 3 8 9
B's task:
2 4 5 6 7 10
----------------------------------------
119
----------------------------------------
3   1   6   5   7   8   4   9   2
5   2   9   1   3   4   7   6   8
4   8   7   6   2   9   5   3   1
2   6   3   4   1   5   9   8   7
```

```
9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9
```

实验体会

在这个算法课程中，我深深地体会到了算法的重要性和魅力。算法不仅是解决问题的关键，也是提升编程技能、优化代码性能的必备工具。通过学习和实践各种算法，我对问题解决的思路有了更深入的理解，也更加熟练地掌握了如何将抽象的问题转化为具体的代码实现。

特别是在解决数独问题的过程中，我体验到了回溯算法的强大。它能够通过系统地搜索所有可能的解决方案来找到问题的答案。虽然这个过程可能会涉及到大量的计算，但是通过合理的剪枝和优化，我们可以在可接受的时间内得到结果。这让我深刻地认识到，一个好的算法不仅需要解决问题，还需要考虑到效率和性能。

此外，我也认识到了编程实践的重要性。理论知识虽然重要，但是只有通过实际的编程实践，才能真正地理解和掌握这些知识。在编程的过程中，我遇到了各种各样的问题，但是通过查阅资料、思考和尝试，我最终都成功地解决了这些问题。这个过程不仅提升了我的编程技能，也锻炼了我的问题解决能力。

总的来说，这个算法课程让我收获颇丰。我不仅学到了许多有用的算法知识，也提升了我的编程和问题解决能力。我相信，这些知识和技能将对我的未来学习和工作产生深远的影响。