实验题第 1 题

(1) 设计思路

本题要求**在由 3×3 个方格构成的方阵填入 1～n 中的 9 个数字，每个方格填一个整数（不重复）使得所有相邻两个方格内的整数之和为素数**。 解题采用回溯法，逐个尝试填充矩阵的每一格，当所有条件都满足时，将解存储起来。

- 在 Solve 函数中，首先对状态进行初始化，包括解的数量 `count`、标记数组 `tag`，然后调用 Backtrack 函数进行递归搜索。
- Backtrack 函数是回溯的核心逻辑。其基本步骤如下：
  - 当所有格子都填满（`q == 0`）时，将当前解保存到 `ans` 中，回溯返回；
  - 遍历 `[1, ub]` 范围内的数字，利用标记数组 tag，找到未使用的数字；
  - 如果当前格子不是第一个格子，还需要检查是否满足素数条件：左侧数字与当前数字之和、上侧数字与当前数字之和均需为素数；
  - 满足条件时，将数字填入当前格子，并继续递归处理下一个格子（`q - 1`）。处理完成后，需要回溯恢复当前格子的状态，尝试其他可能的数字。

通过回溯法，本题实现了对所有可能解的搜索，并通过素数条件剪枝，大幅减少了搜索空间。实现出代码如下。

(2) 源代码

DigitalGameProblem.h

```
class DigitalGameProblem {
protected:
    int ub = 0; // 数字的上界，表示可用数字的最大值
    int ans[50][9]; // 用于存储所有解的二维数组，每行表示一个解
    int count = 0; // 解的数量
    int tag[100]; // 标记数组，记录数字是否被使用
    void Backtrack(int a[9], int q); // 回溯核心函数
public:
    DigitalGameProblem(int u) { this->ub = u; }; // 构造函数，初始化上界
    int **GetSolution(); // 获取解的二维数组
    int Solve(); // 开始求解函数
};

int Isprime(int i) // 素数判断函数
{
    if (i == 1) return 0; // 1 不是素数
    if (i == 2) return 1; // 2 是素数
    for (int j = 2; j * j <= i; j++) // 从 2 遍历到 sqrt(i)
        if (i % j == 0) // 如果能整除，说明不是素数
            return 0;
    return 1; // 否则是素数
}

int DigitalGameProblem::Solve() {
    count = 0; // 重置解的数量，确保每次求解都是新的
    int a[9] = {0}; // 初始化 9 个格子为空
    for (int i = 0; i < 100; ++i) {
        tag[i] = 0; // 初始化标记数组，表示所有数字未使用
```

```
    }
    Backtrack(a, 9); // 从第 9 格开始回溯搜索
    return count; // 返回找到的解的数量
}

void DigitalGameProblem::Backtrack(int a[9], int q) {
    if (q == 0) { // 如果所有格子已经填满
        for (int i = 0; i < 9; i++) { // 将当前解存储到 ans 数组中
            ans[count][i] = a[i];
        }
        count++; // 解的数量加 1
        return;
    }
    for (int i = 1; i <= ub; i++) { // 尝试每个数字
        if (!tag[i]) { // 如果数字 i 没有被使用
            if (q < 9) { // 检查素数条件
                int index = 9 - q; // 计算当前格子的索引
                // 检查左侧和上侧是否满足素数条件
                if ((index % 3 > 0 && !Isprime(a[index - 1] + i)) || // 左侧格子与当前数
和是否为素数
                    (index >= 3 && !Isprime(a[index - 3] + i))) { // 上侧格子与当前数和
是否为素数
                    continue; // 如果不满足条件，尝试下一个数字
                }
            }
            tag[i] = 1; // 标记数字 i 已使用
            a[9 - q] = i; // 将数字 i 放入当前格子
            Backtrack(a, q - 1); // 递归处理下一个格子
            tag[i] = 0; // 回溯，撤销标记
        }
    }
}

int** DigitalGameProblem::GetSolution() {
    if (count == 0) return nullptr; // 如果没有解，返回空指针

    // 动态分配二维数组存储所有解
    int** result = new int*[count]; // 分配行指针数组
    for (int i = 0; i < count; i++) {
        result[i] = new int[9]; // 分配每行存储 9 个数字
        for (int j = 0; j < 9; j++) {
            result[i][j] = ans[i][j]; // 将 ans 中的解复制到 result 中
        }
    }
    return result; // 返回存储解的二维数组
}
```

(3) 说明

上述代码可以应用测试程序进行测试并得到正确结果。

实验题第 3 题

第 1 小题

(1) 设计思路

本题要求**设计一个算法，采用队列式分支限界法求解 n 皇后问题的一个解**。

- 基于队列式分支限界法来解决 N 皇后问题，按照题目要求实现逐行放置皇后并结合限界条件进行动态搜索的过程。算法以数组 `q` 表示棋盘状态，其中 `q[i]` 记录第 `i` 行皇后的位置。
- 整个方法的核心在于逐行尝试每个位置，动态判断是否满足限界条件。通过 `place` 函数检查当前放置是否合法，主要判断是否与已放置的皇后发生列冲突或对角线冲突。如果当前行找到合法位置，则继续下一行的尝试；若当前行无合法位置，则回溯上一行，调整皇后位置，继续搜索。
- 当成功为所有行找到合法位置时，返回 `true` 表示找到一个解；若穷尽所有可能仍未找到解，则返回 `false`。

根据以上分析，设计出代码如下。

(2) 源代码

NQueensProblem.h **//关键的实现函数**

```
bool NQueensProblem::BranchAndBound() {
    int i = 1;  // 当前处理的行
    q[i] = 0;   // 初始化第一行的列号

    while (i > 0) {
        q[i]++;  // 尝试当前行的下一个列

        // 如果当前位置不合法，则尝试当前行的下一个列
        while (q[i] <= n && !place(i, q)) {
            q[i]++;
        }

        if (q[i] <= n) { // 找到合法位置
            if (i == n) { // 如果已经处理到最后一行
                return true; // 找到一个解
            }
            i++;  // 进入下一行
            q[i] = 0;  // 初始化下一行
        } else { // 当前行没有合法位置
            i--;  // 回溯上一行
        }
    }

    return false; // 没有解
}

bool NQueensProblem::Solve() {
    return BranchAndBound(); // 调用分支限界法求解
}

int* NQueensProblem::GetSolution() {
```

```
    int* result = new int[n + 1];  // 动态分配存储解的数组
    for (int i = 1; i <= n; i++) {
        result[i] = q[i];  // 将解复制到 result 中
    }
    return result;  // 返回解
}
```

(3)说明

上述代码可以应用测试程序进行测试并得到正确结果。

第 2 小题

(1) 设计思路

本题要求设计一个算法，采用优先队列式分支限界法求解 n 皇后问题的一个解。

- 使用优先队列式分支限界法求解 n 皇后问题，需要结合优先级策略对解空间的节点进行管理与扩展。优先队列在此处的核心作用是根据设定的优先级，动态调整节点的处理顺序，以期更高效地找到解或剪枝无效路径。
  - 首先，将问题视作一个扩展解空间树的过程。解空间的每个节点记录当前放置皇后的状态，包括已处理的行数和皇后的位置。根节点对应空棋盘，通过逐步向下扩展，每次将当前节点的皇后放置状态复制到新节点，尝试为下一行选择一个合法的列。
  - 通过优先队列存储这些节点，可以根据设定的优先级对节点进行动态排序。优先级策略的设计至关重要，比如在此代码中简单使用当前节点的层次（行数）作为优先级，确保节点扩展的顺序与解空间树的深度一致。
  - 在每次扩展节点时，先判断当前节点是否已达终止条件（即所有行都放置完毕）。如果满足条件，则找到一个解，直接返回。如果未完成，则继续尝试扩展下一行的所有可能列，同时判断放置是否合法。合法的子节点被加入优先队列，非法节点则直接舍弃以节省内存。

根据以上分析，设计出代码如下。

(2) 源代码

NQueensProblem.h **//关键的实现函数**

```
struct Node {
    int level;      // 当前处理到的行
    int* position;  // 皇后放置情况，position[i] 表示第 i 行皇后所在的列
    int priority;   // 优先级
};

bool NQueensProblem::BranchAndBound() {
    auto compare = [](Node* a, Node* b) { return a->priority > b->priority; };
    // 定义优先队列，优先级高的节点会排在队列前端
    priority_queue<Node*, vector<Node*>, decltype(compare)> pq(compare);

    // 初始化根节点
    Node* root = new Node;
    root->level = 0; // 根节点尚未开始放置皇后
```

```
    root->position = new int[n + 1]{0}; // 动态分配存储位置数组，并初始化为 0
    root->priority = 0; // 根节点的优先级设为 0
    pq.push(root); // 将根节点入队

    while (!pq.empty()) { // 当队列不为空时
        Node* current = pq.top(); // 取出优先级最高的节点
        pq.pop();

        if (current->level == n) { // 如果当前节点已经放置到第 n 行，说明找到解
            solution = current; // 将当前节点保存为解
            return true; // 返回成功
        }

        int nextLevel = current->level + 1; // 计算下一行的编号
        for (int col = 1; col <= n; col++) { // 遍历当前行的所有列
            Node* child = new Node; // 创建新的子节点
            child->level = nextLevel; // 子节点的层次为当前层次 + 1
            child->position = new int[n + 1]; // 动态分配存储位置数组
            for (int i = 1; i <= n; i++) { // 复制父节点的皇后位置
                child->position[i] = current->position[i];
            }
            child->position[nextLevel] = col; // 在下一行放置皇后

            if (place(nextLevel, child->position)) { // 判断当前放置是否合法
                child->priority = nextLevel; // 计算优先级（简单设为层次号）
                pq.push(child); // 合法节点入队
            } else {
                delete[] child->position; // 非法节点，释放内存
                delete child;
            }
        }

        delete[] current->position; // 当前节点已扩展完毕，释放其位置数组
        delete current; // 释放当前节点的内存
    }

    return false; // 如果队列为空仍未找到解，则无解
}
```

(3)说明

上述代码可以应用测试程序进行测试并得到正确结果。

值得注意的是，相较于**队列式**分支限界法，**优先队列法**利用优先级策略动态调整扩展顺序，可以在解空间较大时，更有效地剪枝。(本题直接使用 stl 的 queue 容器进行解题，而没有使用大顶堆)