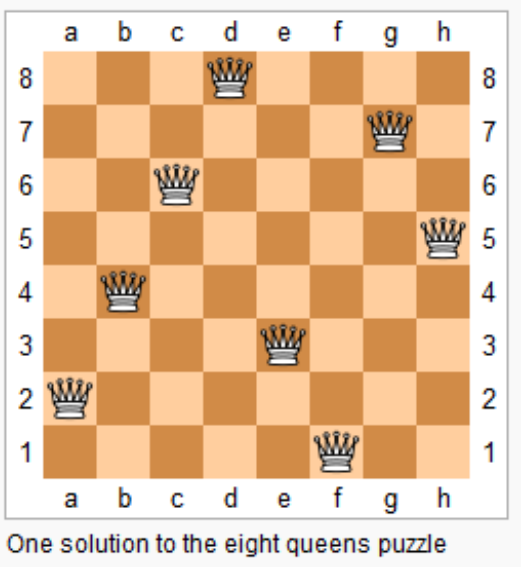
N-Queens

The n-queens puzzle is the problem of placing n queens on an $n \times n$ chessboard such that no two queens attack each other.



One solution to the eight queens puzzle

Given an integer n, return all distinct solutions to the n-queens puzzle.

Each solution contains a distinct board configuration of the n-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space respectively.

For example,
There exist two distinct solutions to the 4-queens puzzle:

```
[
 [".Q..",  // Solution 1
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.",  // Solution 2
  "Q...",
  "...Q",
  ".Q.."]
]
```

Solution 1

In this problem, we can go row by row, and in each position, we need to check if the `column`, the `45° diagonal` and the `135° diagonal` had a queen before.

Solution A: Directly check the validity of each position,*12ms*:

```
class Solution {
public:
    std::vector<std::vector<std::string> > solveNQueens(int n) {
        std::vector<std::vector<std::string> > res;
        std::vector<std::string> nQueens(n, std::string(n, '.'));
        solveNQueens(res, nQueens, 0, n);
        return res;
    }
private:
    void solveNQueens(std::vector<std::vector<std::string> > &res, std::vector<st
d::string> &nQueens, int row, int &n) {
        if (row == n) {
            res.push_back(nQueens);
            return;
        }
        for (int col = 0; col != n; ++col)
            if (isValid(nQueens, row, col, n)) {
                nQueens[row][col] = 'Q';
                solveNQueens(res, nQueens, row + 1, n);
                nQueens[row][col] = '.';
            }
    }
    bool isValid(std::vector<std::string> &nQueens, int row, int col, int &n) {
        //check if the column had a queen before.
        for (int i = 0; i != row; ++i)
            if (nQueens[i][col] == 'Q')
                return false;
        //check if the 45° diagonal had a queen before.
        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; --i, --j)
            if (nQueens[i][j] == 'Q')
                return false;
        //check if the 135° diagonal had a queen before.
        for (int i = row - 1, j = col + 1; i >= 0 && j < n; --i, ++j)
            if (nQueens[i][j] == 'Q')
                return false;
        return true;
    }
};
```

Solution B: Use flag vectors as bitmask, *4ms*:

The number of columns is `n`, the number of 45° diagonals is `2 * n - 1`, the number of 135° diagonals is also `2 * n - 1`. When reach `[row, col]`, the column No. is `col`, the 45° diagonal No. is `row + col` and the 135° diagonal No. is `n - 1 + col - row`. We can use three arrays to indicate if the column or the diagonal had a queen before, if not, we can put a queen in this position and continue.

```
/**    | | |               / / /            \ \ \
  *    0 0 0              0 0 0             0 0 0
  *    | | |             / / / /           \ \ \ \
  *    0 0 0              0 0 0             0 0 0
  *    | | |             / / / /           \ \ \ \
  *    0 0 0              0 0 0             0 0 0
  *    | | |             / / /               \ \ \
  *   3 columns       5 45° diagonals     5 135° diagonals    (when n is 3)
  */
class Solution {
public:
    std::vector<std::vector<std::string> > solveNQueens(int n) {
        std::vector<std::vector<std::string> > res;
        std::vector<std::string> nQueens(n, std::string(n, '.'));
        std::vector<int> flag_col(n, 1), flag_45(2 * n - 1, 1), flag_135(2 * n -
1, 1);
        solveNQueens(res, nQueens, flag_col, flag_45, flag_135, 0, n);
        return res;
    }
private:
    void solveNQueens(std::vector<std::vector<std::string> > &res, std::vector<st
d::string> &nQueens, std::vector<int> &flag_col, std::vector<int> &flag_45, std::
vector<int> &flag_135, int row, int &n) {
        if (row == n) {
            res.push_back(nQueens);
            return;
        }
        for (int col = 0; col != n; ++col)
            if (flag_col[col] && flag_45[row + col] && flag_135[n - 1 + col - row
]) {
                flag_col[col] = flag_45[row + col] = flag_135[n - 1 + col - row]
= 0;
                nQueens[row][col] = 'Q';
                solveNQueens(res, nQueens, flag_col, flag_45, flag_135, row + 1,
n);
                nQueens[row][col] = '.';
                flag_col[col] = flag_45[row + col] = flag_135[n - 1 + col - row]
= 1;
            }
    }
};
```

But we actually do not need to use three arrays, we just need one. Now, when reach `[row, col]`, the subscript of column is `col`, the subscript of 45° diagonal is `n + row + col` and the subscript of 135° diagonal is `n + 2 * n - 1 + n - 1 + col - row`.

```
class Solution {
public:
    std::vector<std::vector<std::string> > solveNQueens(int n) {
        std::vector<std::vector<std::string> > res;
        std::vector<std::string> nQueens(n, std::string(n, '.'));
        /*
        flag[0] to flag[n - 1] to indicate if the column had a queen before.
        flag[n] to flag[3 * n - 2] to indicate if the 45° diagonal had a queen be
fore.
        flag[3 * n - 1] to flag[5 * n - 3] to indicate if the 135° diagonal had a
queen before.
        */
        std::vector<int> flag(5 * n - 2, 1);
        solveNQueens(res, nQueens, flag, 0, n);
        return res;
    }
private:
    void solveNQueens(std::vector<std::vector<std::string> > &res, std::vector<st
d::string> &nQueens, std::vector<int> &flag, int row, int &n) {
        if (row == n) {
            res.push_back(nQueens);
            return;
        }
        for (int col = 0; col != n; ++col)
            if (flag[col] && flag[n + row + col] && flag[4 * n - 2 + col - row])
{
                flag[col] = flag[n + row + col] = flag[4 * n - 2 + col - row] = 0
;
                nQueens[row][col] = 'Q';
                solveNQueens(res, nQueens, flag, row + 1, n);
                nQueens[row][col] = '.';
                flag[col] = flag[n + row + col] = flag[4 * n - 2 + col - row] = 1
;
            }
    }
};
```

written by prime_tang original link here

Solution 2

queens can attack other queen in the same row, same column, but i forget the diagonal.. = 。 =

written by aqin original link here

Solution 3

ideas:
Use the `DFS` helper function to find solutions recursively. A solution will be found when the length of `queens` is equal to `n` (`queens` is a list of the indices of the queens).

In this problem, whenever a location `(x, y)` is occupied, any other locations `(p, q)` where `p + q == x + y` or `p - q == x - y` would be invalid. We can use this information to keep track of the indicators (`xy_dif` and `xy_sum`) of the invalid positions and then call DFS recursively with valid positions only.

At the end, we convert the result (a list of lists; each sublist is the indices of the queens) into the desire format.

```
def solveNQueens(self, n):
    def DFS(queens, xy_dif, xy_sum):
        p = len(queens)
        if p==n:
            result.append(queens)
            return None
        for q in range(n):
            if q not in queens and p-q not in xy_dif and p+q not in xy_sum:
                DFS(queens+[q], xy_dif+[p-q], xy_sum+[p+q])
    result = []
    DFS([],[],[])
    return [ ["."*i + "Q" + "."*(n-i-1) for i in sol] for sol in result]
```

written by cmc original link here