回溯算法

回溯法

- 回溯算法实际上一个类似枚举的搜索尝试过程，主要是在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时，就“回溯”返回，尝试别的路径。
- 回溯法是一种选优搜索法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为“回溯点”。也可以称为剪枝点，所谓的剪枝，指的是把不会找到目标，或者不必要的路径裁剪掉。
- 许多复杂的，规模较大的问题都可以使用回溯法，有“通用解题方法”的美称。
- 在包含问题的所有解的解空间树中，按照深度优先搜索的策略，从根结点出发深度探索解空间树。当探索到某一结点时，要先判断该结点是否包含问题的解，如果包含，就从该结点出发继续探索下去，如果该结点不包含问题的解，则逐层向其祖先结点回溯。（其实回溯法就是对隐式图的深度优先搜索算法）。
- 若用回溯法求问题的所有解时，要回溯到根，且根结点的所有可行的子树都要已被搜索遍才结束。
- 而若使用回溯法求任一个解时，只要搜索到问题的一个解就可以结束。
- 除过深度优先搜索，常用的还有广度优先搜索。

深度优先搜索（Depth First Search）------ 一条道走到黑

同学们先思考这样一个问题：

假如有编号为1~3的3张扑克牌和编号为1~3的3个盒子，现在需要将3张牌分别放到3个盒子中去，且每个盒子只能放一张牌，一共有多少种不同的放法。

- 当走到一个盒子面前的时候，到底要放那一张牌呢？在这里应该把所有的牌都尝试一遍。假设这里约定一个顺序，按牌面值从小到大依次尝试。在这样的假定下，当走到第一个盒子的时候，放入1号牌。
- 放好之后，继续向后走，走到第二个盒子面前，此时还剩2张牌，牌面值最小的为2号牌，按照约定的规则，把2号牌放入第二个盒子。
- 此时，来到第三个盒子面前，只剩一张牌，放入第三个盒子。此时手中的牌已经用完。
- 继续向后走，走到了盒子的尽头，后面再也没有盒子，并且也没有可用的牌了，此时，一种放法已经完成了，但是这只是一种放法，这条路已经走到了尽头，还需要折返，重新回到上一个盒子。
- 这里回到第三个盒子，把第三个盒子中的牌取出来，再去尝试能否再放其它的牌，这时候手里仍然只有一张3号牌，没有别的选择了，所以还需要继续向后回退，回到2号盒子面前。
- 收回2号盒子中的2号牌，现在手里有两张牌，2，3，按照约定，再把3号牌放入2号盒子，放好之后，继续向后走，来到3号盒子。
- 此事手里只有一张2号牌，把它放入3号盒子，继续向后走。
- 此时这条路又一次走到了尽头，一个新的放法又产生了，继续向上折返，尝试其它可能，按照上述步骤，依次会产生所有结果。

代码如何实现这种过程呢？最主要的事情，向面前的盒子里放每一种牌，一个for循环搞定。这里还需考虑，现在手里有没有这张牌，用一个数组book标记手里是否有这张牌

```
for(int i = 1; i <= n; i++)
{
    if(book[i] == 0)  //第i号牌仍在手上
    {
        boxs[index] = i;
        book[i] = 1;  //现在第i号牌已经被用了
    }
}
```

面前的盒子处理完成之后，继续处理下一个盒子，下一个盒子的处理方法和当前一样。那么把上面的代码块封装一下，给它取一个名字，即为Dfs(Depth First Search)。

```
//index表示现在走到哪一个盒子面前
void Dfs(int index, int n, vector<int>& boxs, vector<int>& book)
{
    for(int i = 1; i <= n; i++)
    {
        if(book[i] == 0)  //第i号牌仍在手上
        {
            boxs[index] = i;
            book[i] = 1;  //现在第i号牌已经被用了
        }
    }
}
```

现在再去处理下一个盒子，直接调用Dfs(index + 1, boxs, book)即可。

```
void Dfs(int index, int n, vector<int>& boxs, vector<int>& book)
{
    for(int i = 1; i <= n; i++)
    {
        if(book[i] == 0)  //第i号牌仍在手上
        {
            boxs[index] = i;
            book[i] = 1;  //现在第i号牌已经被用了
            //处理下一个盒子
            Dfs(index + 1, n, boxs, book);
            //从下一个盒子回退到当前盒子，取出当前盒子的牌，
            //尝试放入其它牌。
            book[i] = 0;
        }
    }
}
```

现在考虑什么时候得到一种方法呢，走到尽头，也就是第n+1个盒子的时候，表明前面的盒子已经放好牌了，这时候直接打印每个盒子中的牌即可。已走到尽头，向上回退。

```
void Dfs(int index, int n, vector<int>& boxs, vector<int>& book)
{
    if(index == n + 1)
    {
```

```
        for(int i = 1; i <= n; i++)
            cout<<boxs[i]<<" ";
        cout<<endl;
        return; //向上回退
    }

    for(int i = 1; i <= n; i++)
    {
        if(book[i] == 0)  //第i号牌仍在手上
        {
            boxs[index] = i;
            book[i] = 1;  //现在第i号牌已经被用了
            //处理下一个盒子
            Dfs(index + 1, n, boxs, book);
            //从下一个盒子回退到当前盒子，取出当前盒子的牌，
            //尝试放入其它牌。
            book[i] = 0;
        }
    }
}
```

```
//完整代码
#include <vector>
#include <iostream>
using namespace std;
void Dfs(int index, int n, vector<int>& boxs, vector<int>& book)
{
    if (index == n + 1)
    {
        for (int i = 1; i <= n; i++)
            cout << boxs[i] << " ";
        cout << endl;
        return; //向上回退
    }

    for (int i = 1; i <= n; i++)
    {
        if (book[i] == 0)  //第i号牌仍在手上
        {
            boxs[index] = i;
            book[i] = 1;  //现在第i号牌已经被用了
            //处理下一个盒子
            Dfs(index + 1, n, boxs, book);
            //从下一个盒子回退到当前盒子，取出当前盒子的牌，
            //尝试放入其它牌。
            book[i] = 0;
        }
    }
}

int main()
{
    int n;
```

```
    vector<int> boxs;
    vector<int> books;

    cin >> n;
    boxs.resize(n + 1, 0);
    books.resize(n + 1, 0);
    Dfs(1, n, boxs, books);
    return 0;
}

/*java*/
public static void Dfs(int index, int n, int[] boxs, int[] book)
    {
        if (index == n + 1)
        {
            for (int i = 1; i <= n; i++)
                System.out.print(boxs[i] + " ");
            System.out.println();
            return; //向上回退
        }

        for (int i = 1; i <= n; i++)
        {
            if (book[i] == 0)  //第i号牌仍在手上
            {
                boxs[index] = i;
                book[i] = 1;  //现在第i号牌已经被用了
                //处理下一个盒子
                Dfs(index + 1, n, boxs, book);
                //从下一个盒子回退到当前盒子，取出当前盒子的牌，
                //尝试放入其它牌。
                book[i] = 0;
            }
        }
    }

    public static void main(String[] args) {
        int n;

        Scanner sc = new Scanner(System.in);
        n = sc.nextInt();
        int[] boxs = new int[n + 1];
        int[] books = new int[n + 1];
        Dfs(1, n, boxs, books);
    }
```

从上面的代码可以看出，深度优先搜索的关键是解决"当下该如何做"，下一步的做法和当下的做法是一样的。"当下如何做"一般是尝试每一种可能，用for循环遍历，对于每一种可能确定之后，继续走下一步，当前的剩余可能等到从下一步回退之后再处理。我们可以抽象出深度优先搜索的模型。

```
Dfs(当前这一步的处理逻辑)
{
    1. 判断边界，是否已经一条道走到黑了：向上回退
    2. 尝试当下的每一种可能
    3. 确定一种可能之后，继续下一步 Dfs(下一步)
}
```

员工的重要性

```cpp
/*
    边界：下属为空
    每次先加第一个下属的重要性
    按照相同的操作再去加下属的第一个下属的重要性
*/
class Solution {
public:
    int DFS(unordered_map<int,Employee*>& info, int id)
    {
        int curImpo = info[id]->importance;
        for(const auto& sid : info[id]->subordinates)
        {
            curImpo += DFS(info, sid);
        }
        return curImpo;
    }
    int getImportance(vector<Employee*> employees, int id) {
        if(employees.empty())
            return 0;
        unordered_map<int,Employee*> info;
        for(const auto& e : employees)
        {
            info[e->id] = e;
        }
        return DFS(info, id);
    }
};

/*java*/
class Solution {
    public int DFS(Map<Integer, Employee> info, int id)
    {
        Employee curE = info.get(id);
        int curSum = curE.importance;
        for(int curId : curE.subordinates)
        {
            curSum += DFS(info, curId);
        }
        return curSum;
    }

    public int getImportance(List<Employee> employees, int id) {
        if(employees.isEmpty())
```

```
            return 0;

        Map<Integer, Employee> info = new HashMap<>();
        for(Employee e : employees)
            info.put(e.id, e);
        return DFS(info, id);
    }
}
```

图像渲染

```
/*
    把和初始坐标开始，颜色值相同的点的颜色全部改为新的颜色
    并且只要某个点颜色被更改，则继续以此点向周围渲染

    比如题目的意思：以位置（1，1）开始，向外渲染，只要渲染点的颜色值和（1，1）位置的颜色值相同，则继续向外渲染
    1   1   1
    1   1   0
    1   0   1

    2   2   2
    2   2   0
    2   0   1

这里每一个符合要求的点都要向四个方向渲染
边界：位置是否越界
这里需要用的标记，避免重复修改，使时间复杂度不超过O(row * col)

*/

#include <vector>
#include <iostream>
using namespace std;

//四个方向的位置更新：顺时针更新
int nextPosition[4][2] = { { 0, 1 }, { 1, 0 }, { 0, -1 }, { -1, 0 } };
class Solution {
public:
    void dfs(vector<vector<int>>& image, int row, int col, vector<vector<int>>& book, int sr, int sc, int oldColor, int newColor)
    {

        //处理当前逻辑，修改颜色，并且标记已经修改过了
        image[sr][sc] = newColor;
        book[sr][sc] = 1;
        //遍历每一种可能，四个方向
        for (int k = 0; k < 4; ++k)
        {
            int newSr = sr + nextPosition[k][0];
            int newSc = sc + nextPosition[k][1];
            //判断新位置是否越界
            if (newSr >= row || newSr < 0
```

```
                || newSc >= col || newSc < 0)
                continue;
            //如果颜色符合要求，并且之前也没有渲染过，则继续渲染
            if (image[newSr][newSc] == oldColor && book[newSr][newSc] == 0)
            {
                dfs(image, row, col, book, newSr, newSc, oldColor, newColor);
            }
        }
    }
    vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int newColor)
{
        if (image.empty())
            return image;
        int row = image.size();
        int col = image[0].size();
        //建立标记
        vector<vector<int>> book;
        book.resize(row);
        for (int i = 0; i < row; ++i)
        {
            book[i].resize(col, 0);
        }
        //获取旧的颜色
        int oldColor = image[sr][sc];
        dfs(image, row, col, book, sr, sc, oldColor, newColor);
        return image;
    }
};

/*java*/
class Solution {
   //四个方向的位置更新：顺时针更新
    int[][] nextPosition = { { 0, 1 }, { 1, 0 }, { 0, -1 }, { -1, 0 } };
    public void dfs(int[][] image, int row, int col, int[][] book,
                    int sr, int sc, int oldColor, int newColor)
    {
        //处理当前逻辑，修改颜色，并且标记已经修改过了
        image[sr][sc] = newColor;
        book[sr][sc] = 1;

        //遍历每一种可能，四个方向
        for (int k = 0; k < 4; ++k)
        {
            int newSr = sr + nextPosition[k][0];
            int newSc = sc + nextPosition[k][1];
            //判断新位置是否越界
            if (newSr >= row || newSr < 0
                    || newSc >= col || newSc < 0)
                continue;
            //如果颜色符合要求，并且之前也没有渲染过，则继续渲染
            if (image[newSr][newSc] == oldColor && book[newSr][newSc] == 0)
            {
                dfs(image, row, col, book, newSr, newSc, oldColor, newColor);
```

```
            }
        }
    }

    public int[][] floodFill(int[][] image, int sr, int sc, int newColor) {
        int oldColor = image[sr][sc];
        int row = image.length;
        int col = image[0].length;
        //建立标记
        int[][] book = new int[row][col];
        dfs(image, row, col, book, sr, sc, oldColor, newColor);
        return image;
    }

}
```

相似题目：

[岛屿的周长]

[被围绕的区域]

```
/*
本题的意思被包围的区间不会存在于边界上，所以边界上的o以及与o联通的都不算做包围，只要把边界上的o以及与之联通的o进行特殊处理，剩下的o替换成x即可。故问题转化为，如何寻找和边界联通的o，我们需要考虑如下情况。

X X X X
X O O X
X X O X
X O O X
从每一个边缘的o开始，只要和边缘的o联通，则它就没有被包围。
1.首先寻找边上的每一个o，如果没有，表示所有的o都被包围
2.对于边上的每一个o进行dfs进行扩散，先把边上的每一个o用特殊符号标记，比如*，#等，
3.把和它相邻的o都替换为特殊符号，每一个新的位置都做相同的dfs操作
4.所有扩散结束之后，把特殊符号的位置（和边界连通）还原为o,原来为o的位置（和边界不连通）替换为x即可。

这里一定要注意这里是大'O'和大'X'

*/

int nextPosition[4][2] = { { 0, 1 }, { 1, 0 }, { 0, -1 }, { -1, 0 } };
class Solution {
public:
    void dfs(vector<vector<char>>& board, int row, int col, int i, int j)
    {
        //当前位置设为'*'
        board[i][j] = '*';
        for (int k = 0; k < 4; ++k)
        {
            //向四个方向扩散
            int ni = i + nextPosition[k][0];
```

```
            int nj = j + nextPosition[k][1];
            //判断边界
            if (ni < 0 || ni >= row
                || nj < 0 || nj >= col)
                continue;
            //是'O'说明和边联通，继续搜索是否还有联通的
            if (board[ni][nj] != '*' && board[ni][nj] != 'X')
                dfs(board, row, col, ni, nj);
        }
    }
    void solve(vector<vector<char>>& board) {
        if (board.empty())
            return;
        //寻找边上的每一个0，如果没有，
        //表示所有的0都被包围
        int row = board.size();
        int col = board[0].size();

        //寻找第一行和最后一行
        for (int j = 0; j < col; ++j)
        {
            if (board[0][j] == 'O')
                dfs(board, row, col, 0, j);
            if (board[row - 1][j] == 'O')
                dfs(board, row, col, row - 1, j);
        }
        //寻找第一列和最后一列
        for (int i = 0; i < row; ++i)
        {
            if (board[i][0] == 'O')
                dfs(board, row, col, i, 0);
            if (board[i][col - 1] == 'O')
                dfs(board, row, col, i, col - 1);
        }

        for (int i = 0; i < row; ++i)
        {
            for (int j = 0; j < col; ++j)
            {
                if (board[i][j] == '*')
                    board[i][j] = 'O';
                else if (board[i][j] == 'O')
                    board[i][j] = 'X';
            }
        }
    }
};

/*java*/
class Solution {
    //四个方向的位置更新：顺时针更新
    int[][] nextPosition = { { 0, 1 }, { 1, 0 }, { 0, -1 }, { -1, 0 } };
    public void dfs(char[][] board, int row, int col, int i, int j)
```

```
{
    //当前位置设为'*'
    board[i][j] = '*';
    for (int k = 0; k < 4; ++k)
    {
        //向四个方向扩散
        int ni = i + nextPosition[k][0];
        int nj = j + nextPosition[k][1];
        //判断边界
        if (ni < 0 || ni >= row
                || nj < 0 || nj >= col)
            continue;
        //是'O'说明和边联通，继续搜索是否还有联通的
        if (board[ni][nj] != '*' && board[ni][nj] != 'X')
            dfs(board, row, col, ni, nj);
    }
}

public void solve(char[][] board) {
    if (board.length == 0)
        return;
    //寻找边上的每一个0，如果没有，
    //表示所有的0都被包围
    int row = board.length;
    int col = board[0].length;

    //寻找第一行和最后一行
    for (int j = 0; j < col; ++j)
    {
        if (board[0][j] == 'O')
            dfs(board, row, col, 0, j);
        if (board[row - 1][j] == 'O')
            dfs(board, row, col, row - 1, j);
    }
    //寻找第一列和最后一列
    for (int i = 0; i < row; ++i)
    {
        if (board[i][0] == 'O')
            dfs(board, row, col, i, 0);
        if (board[i][col - 1] == 'O')
            dfs(board, row, col, i, col - 1);
    }

    for (int i = 0; i < row; ++i)
    {
        for (int j = 0; j < col; ++j)
        {
            if (board[i][j] == '*')
                board[i][j] = 'O';
            else if (board[i][j] == 'O')
                board[i][j] = 'X';
        }
    }
```

```
    }
}
```

岛屿数量

```
/*
本题的意思是连在一起的陆地都算做一个岛屿，本题可以采用类似渲染的做法，尝试以每个点作为渲染的起点，可以渲染的陆地都算做一个岛屿，最后看渲染了多少次，即深度优先算法执行了多少次
*/

int nextPosition[4][2] = { { 0, 1 }, { 1, 0 }, { 0, -1 }, { -1, 0 } };
class Solution {
public:

    void dfs(vector<vector<char>>& grid, int row, int col, vector<vector<int>>& book, int x,
int y)
    {

        //处理当前逻辑
        book[x][y] = 1;
        //遍历每一种可能，四个方向
        for (int k = 0; k < 4; ++k)
        {
            int nX = x + nextPosition[k][0];
            int nY = y + nextPosition[k][1];
            //判断新位置是否越界
            if (nX >= row || nX < 0
                || nY >= col || nY < 0)
                continue;
            //如果符合要求，并且之前也没有渲染过，则继续渲染
            if (grid[nX][nY] == '1' && book[nX][nY] == 0)
            {
                dfs(grid, row, col, book, nX, nY);
            }
        }
    }


    int numIslands(vector<vector<char>>& grid) {
        if (grid.empty())
            return 0;
        int ret = 0;
        int row = grid.size();
        int col = grid[0].size();

        vector<vector<int>> book;
        book.resize(row);
        for (int i = 0; i < row; ++i)
            book[i].resize(col, 0);

        //以每一个网格点为渲染起点开始
        for (int i = 0; i < row; ++i)
        {
```

```
            for (int j = 0; j < col; ++j)
            {
                if (grid[i][j] == '1' && book[i][j] == 0)
                {
                    ++ret;
                    dfs(grid, row, col, book, i, j);
                }
            }
        }
        return ret;
    }
};

/*java*/
class Solution {
    int[][] nextPosition = { { 0, 1 }, { 1, 0 }, { 0, -1 }, { -1, 0 } };

    public void dfs(char[][] grid, int row, int col, int[][] book, int x, int y)
    {

        //处理当前逻辑
        book[x][y] = 1;
        //遍历每一种可能，四个方向
        for (int k = 0; k < 4; ++k)
        {
            int nX = x + nextPosition[k][0];
            int nY = y + nextPosition[k][1];
            //判断新位置是否越界
            if (nX >= row || nX < 0
                    || nY >= col || nY < 0)
                continue;
            //如果符合要求，并且之前也没有渲染过，则继续渲染
            if (grid[nX][nY] == '1' && book[nX][nY] == 0)
            {
                dfs(grid, row, col, book, nX, nY);
            }
        }
    }


    public int numIslands(char[][] grid) {
        if (grid.length == 0)
            return 0;
        int ret = 0;
        int row = grid.length;
        int col = grid[0].length;

        int[][] book = new int[row][col];
        //以每一个网格点为渲染起点开始
        for (int i = 0; i < row; ++i)
        {
            for (int j = 0; j < col; ++j)
            {
```

```
                if (grid[i][j] == '1' && book[i][j] == 0)
                {
                    ++ret;
                    dfs(grid, row, col, book, i, j);
                }
            }
        }
        return ret;
    }
}
```

相似题目：

[岛屿的最大面积]

[电话号码的字母组合]

```
/*
此题DFS + 回溯
*/
static string mapString[] = {"", "", "abc", "def", "ghi", "jkl", "mno",
                             "pqrs", "tuv", "wxyz"};
class Solution {
public:
    void backTrace(string& digits, vector<string>& ret, string curStr, int curDepth)
    {
        //边界，找到一种组合，放入数组中，结束此路径，向上回溯
        if(curDepth == digits.size())
        {
            if(!curStr.empty())
            {
                ret.push_back(curStr);
            }
            return;
        }
        //找到当前字符在string映射表中的位置
        int curMapIndex = digits[curDepth] - '0';
        string curMap = mapString[curMapIndex];
        //遍历每一种可能的组合
        for(auto& ch : curMap)
        {
            backTrace(digits, ret, curStr + ch, curDepth + 1);
        }
    }

    vector<string> letterCombinations(string digits) {
        vector<string> ret;
        backTrace(digits, ret, "", 0);
        return ret;
    }
};
```

```
/*java*/
class Solution {

    String[] mapString = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};

    public void backTrace(String digits, List<String> ret, StringBuilder curStr, int
curDepth) {
        //边界，找到一种组合，放入数组中，结束此路径，向上回溯
        if (curDepth == digits.length()) {
            if (curStr.length() != 0) {
                ret.add(curStr.toString());
            }
            return;
        }
        //找到当前字符在string映射表中的位置
        int curMapIndex = digits.charAt(curDepth) - '0';
        String curMap = mapString[curMapIndex];
        //遍历每一种可能的组合
        for (int i = 0; i < curMap.length(); ++i)
        {
            backTrace(digits, ret, curStr.append(curMap.charAt(i)), curDepth + 1);
            curStr.deleteCharAt(curStr.length() - 1);
        }
    }

    public List<String> letterCombinations(String digits) {
        List<String> ret = new ArrayList<>();
        StringBuilder curStr = new StringBuilder("");
        backTrace(digits, ret, curStr, 0);
        return ret;
    }
}
```

相似题目：

[二进制手表]

[组合总和]

```
/*
此题相加的元素可以重复，所以去下一个元素的位置可以从当前位置开始， DFS + 回溯
为了保证组合不重复(顺序不同，元素相同，也算重复)，不再从当前位置向前看
1. 从第一个元素开始相加
2. 让局部和继续累加候选的剩余值
3. 局部和等于目标值，保存组合，向上回退，寻找其它组合
*/

class Solution {
public:
    void dfs(vector<int>& candidates, vector<vector<int>>& solutions,
```

```
             vector<int>& solution, int curSum,
             int prevPosition, int target)
    {
        //边界，如果大于等于目标，则结束
        if(curSum >= target)
        {
            //等于目标，找到一个组合
            if(curSum == target)
                solutions.push_back(solution);
            return;
        }
        //可以从上一个位置开始，因为元素可以重复
        for(int i = prevPosition; i < candidates.size(); ++i)
        {
            //单个值已经大于目标，直接跳过
            if(candidates[i] > target)
                continue;
            solution.push_back(candidates[i]);
            dfs(candidates, solutions, solution, curSum + candidates[i], i, target);
            //回溯，向上回退
            solution.pop_back();
        }
    }
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        vector<vector<int>> solutions;
        vector<int> solution;
        if(candidates.empty())
            return solutions;
        int curSum = 0;
        dfs(candidates, solutions, solution, curSum, 0, target);
        return solutions;
    }
};

/*java*/
class Solution {
    public void dfs(int[] candidates, List<List<Integer>> solutions, List<Integer> solution,
int curSum, int prevPosition, int target)
    {
        //边界，如果大于等于目标，则结束
        if(curSum >= target)
        {
            //等于目标，找到一个组合
            if(curSum == target)
            {
                List<Integer> newS = new ArrayList<>();
                for(int e : solution)
                    newS.add(e);
                solutions.add(newS);
            }

            return;
        }
```

```
        //可以从上一个位置开始，因为元素可以重复
        for(int i = prevPosition; i < candidates.length; ++i)
        {
            //单个值已经大于目标，直接跳过
            if(candidates[i] > target)
                continue;
            solution.add(candidates[i]);
            dfs(candidates, solutions, solution, curSum + candidates[i], i, target);
            //回溯，向上回退
            solution.remove(solution.size() - 1);
        }
    }

    public List<List<Integer>> combinationSum(int[] candidates, int target) {
        List<List<Integer>> solutions = new ArrayList<>();
        List<Integer> solution = new ArrayList<>();
        if(candidates.length == 0)
            return solutions;
        int curSum = 0;
        dfs(candidates, solutions, solution, curSum, 0, target);
        return solutions;
    }
}
```

活字印刷

```
/*
此题组合的长度不唯一，最小组合长度为1， 最大组合长度为tiles的长度。
按照题意tiles中每一个位置的字符在组合中只能出现一次， 所以可以用一个标记辅助
当去组合新的组合时，可以与tiles中的每一个位置组合，但是如果当前位置已经在当前组合中出现过，则跳过
虽然此题中每一个位置的字符在组合中只能出现一次，但是tiles中可能有相同的字符，所以需要考虑重复的组合
而unordered_set可以天然去重，可以用其去重
DFS + 回溯:
    1.当前组合不为空，则插入set中
    2.继续给当前组合拼接新的组合，尝试拼接tiles每一个位置的字符
    3.如果当前位置已在组合中出现过，返回到2，否则标记当前位置，继续拼接更长的组合
    4.回溯，尝试组合其它位置，返回2
当所有位置都已经使用过时，当前递归就结束了，继续向上层DFS回退

最终返回set的大小即为组合数目。

*/

class Solution {
public:
    void dfs(string& tiles, string curStr, vector<int>& usedIdx, unordered_set<string>&
totalString)
    {
        if(!curStr.empty())
        {
           totalString.insert(curStr);
        }
        //标记保证所有位都用完之后，就结束了
```

```
        for(int i = 0; i < tiles.size(); ++i)
        {
            //当前位置的字符已用过，直接跳过
            if(usedIdx[i])
                continue;
            usedIdx[i] = 1;
            dfs(tiles, curStr + tiles[i], usedIdx, totalString);
            //回溯，尝试其它字符
            usedIdx[i] = 0;
        }

    }
    int numTilePossibilities(string tiles) {
        if(tiles.empty())
            return 0;
        unordered_set<string> totalString;
        //标记全部初始化为未使用
        vector<int> usedIdx(tiles.size(), 0);
        dfs(tiles, "", usedIdx, totalString);
        return totalString.size();
    }
};

/*java*/
class Solution {
    public void dfs(String tiles, StringBuilder curStr, List<Integer> usedIdx, Set<String>
totalString)
    {
        if(curStr.length() != 0)
        {
            totalString.add(curStr.toString());
        }
        //标记保证所有位都用完之后，就结束了
        for(int i = 0; i < tiles.length(); ++i)
        {
            //当前位置的字符已用过，直接跳过
            if(usedIdx.get(i) == 1)
                continue;
            usedIdx.set(i, 1);
            dfs(tiles, curStr.append(tiles.charAt(i)), usedIdx, totalString);
            //回溯，尝试其它字符
            usedIdx.set(i, 0);
            curStr.deleteCharAt(curStr.length() - 1);
        }

    }

    public int numTilePossibilities(String tiles) {
        if(tiles.length() == 0)
            return 0;
        Set<String> totalString = new HashSet<>();

        //标记全部初始化为未使用
```

```
        List<Integer> usedIdx = new ArrayList<>();
        for(int i = 0; i < tiles.length(); ++i)
        {
            usedIdx.add(0);
        }
        StringBuilder curStr = new StringBuilder("");
        dfs(tiles, curStr, usedIdx, totalString);
        return totalString.size();
    }
}
```

N皇后

```
/*
N皇后问题：把N个皇后放值N*N的二维矩阵中，保证他们相互不能攻击：  即不在同一行，同一列，同一个斜线上(撇捺)
思想：DFS + 回溯
从第一行开始放置皇后，每确定一个位置，判断是否会冲突：  是否在同一列，撇，捺，  不可能在同一行，

同一列：纵坐标相同
“撇”，对应的位置，横坐标加上纵坐标的值是相同的。
“捺”，对应的位置，横坐标减去纵坐标的值也是相同的。

当前行位置确定之后，  继续确定下一行的位置
回退，尝试当前行的其它位置

*/

class Solution {
public:

    vector<vector<string>> solveNQueens(int n) {

        //按坐标位置存放所有解决方案
        vector<vector<pair<int, int>>> solutions;
        //存放一种解决方案中的所有皇后的位置
        vector<pair<int, int>> solution;

        nQueensBacktrack(solutions, solution, 0, n);

        //把坐标位置转成string
        return transResult(solutions, n);
    }

    void nQueensBacktrack(vector<vector<pair<int, int>>>& solutions,
        vector<pair<int, int>>& solution, int curRow, int n) {

            if (curRow == n) solutions.push_back(solution);
            //尝试当前行的每一个位置是否可以放置一个皇后
            for (int col = 0; col < n; ++col) {
                if (isValid(solution, curRow, col)) {
                    //如果可以，在保存当前位置，继续确定下一行皇后的位置
                    //直接调用构造函数，内部构造pair，或者调用make_pair
                    solution.emplace_back(curRow, col);
```

```
                    nQueensBacktrack(solutions, solution, curRow + 1, n);
                    //回溯，删除当前位置，尝试当前行的其它位置
                    solution.pop_back();
                }
            }
        }


    // solution: 一个解决方案，从第一行开始到当前行的上一行每一行已经放置皇后的点
    bool isValid(vector<pair<int, int>>& solution, int row, int col) {
        // 判断当前行尝试的皇后位置是否和前面几行的皇后位置有冲突
        // i.second == col: 第i个皇后与当前这个点在同一列
        // i.first + i.second == row + col: 第i个皇后与当前点在撇上，横坐标+纵坐标值相同
        // i.first - i.second == row - col: 第i个皇后与当前点在捺上，横坐标-纵坐标值相同
        for (pair<int, int> &i : solution)
            if (i.second == col || i.first + i.second == row + col
                || i.first - i.second == row - col)
                return false;
        return true;
    }

    vector<vector<string>> transResult(vector<vector<pair<int, int>>>& solutions, int n) {

        vector<string> tmp();
        //把每一种解决方案都转换为string形式，最终结果
        vector<vector<string>> ret;
        for (vector<pair<int, int>>& solution : solutions) {
            //n*n char: 每行有n个元素，把皇后的位置修改为Q
            vector<string> solutionString(n, string(n, '.'));
            for (pair<int, int>& i : solution) {
                solutionString[i.first][i.second] = 'Q';
            }
            ret.push_back(solutionString);
        }
        return ret;
    }
};

/*java*/
class pair
{
    public  int x;
    public  int y;

    public pair(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

class Solution {
    public List<List<String>> solveNQueens(int n) {
```

```
    //按坐标位置存放所有解决方案
    List<List<pair>> solutions = new ArrayList<>();
    //存放一种解决方案中的所有皇后的位置
    List<pair> solution = new ArrayList<>();

    nQueensBacktrack(solutions, solution, 0, n);

    //把坐标位置转成string
    return transResult(solutions, n);
}

void nQueensBacktrack(List<List<pair>> solutions,
                      List<pair> solution, int curRow, int n) {

    if (curRow == n)
    {
        List<pair> newS = new ArrayList<>();
        for(pair p : solution)
            newS.add(p);
        solutions.add(newS);
    }
    //尝试当前行的每一个位置是否可以放置一个皇后
    for (int col = 0; col < n; ++col) {
        if (isValid(solution, curRow, col)) {
            //如果可以，在保存当前位置，继续确定下一行皇后的位置
            //直接调用构造函数，内部构造pair，或者调用make_pair
            solution.add(new pair(curRow, col));
            nQueensBacktrack(solutions, solution, curRow + 1, n);
            //回溯，删除当前位置，尝试当前行的其它位置
            solution.remove(solution.size() - 1);
        }
    }
}


// solution: 一个解决方案，从第一行开始到当前行的上一行每一行已经放置皇后的点
boolean isValid(List<pair> solution, int row, int col) {
    // 判断当前行尝试的皇后位置是否和前面几行的皇后位置有冲突
    // i.second == col: 第i个皇后与当前这个点在同一列
    // i.first + i.second == row + col: 第i个皇后与当前点在撇上，横坐标+纵坐标值相同
    // i.first - i.second == row - col：第i个皇后与当前点在捺上，横坐标-纵坐标值相同
    for (pair i : solution)
    if (i.y == col || i.x + i.y == row + col
            || i.x - i.y == row - col)
        return false;
    return true;
}

List<List<String>> transResult(List<List<pair>> solutions, int n) {

    List<String> tmp = new ArrayList<>();
    //把每一种解决方案都转换为string形式，最终结果
    List<List<String>> ret = new ArrayList<>();
```

```
        for (List<pair> solution : solutions) {
            //n*n char: 每行有n个元素，把皇后的位置修改为Q
            List<StringBuilder> solutionString = new ArrayList<>();

            for(int i = 0; i < n; ++i)
            {
                StringBuilder sb = new StringBuilder();
                for(int j = 0; j < n; ++j)
                    sb.append('.');
                solutionString.add(sb);
            }
            for (pair i : solution) {
                solutionString.get(i.x).setCharAt(i.y, 'Q');
            }
            List<String> curRet = new ArrayList<>();
            for(StringBuilder sb : solutionString)
                curRet.add(sb.toString());
            ret.add(curRet);
        }
        return ret;
    }
}
```

相似题目：

[N皇后 II]

广度优先搜索（Breadth First Search) ------ 一石激起千层浪

同学们现在来思考这样一个问题， 迷宫问题：

假设有一个迷宫，里面有障碍物，迷宫用二维矩阵表示，标记为0的地方表示可以通过，标记为1的地方表示障碍物，不能通过。现在给一个迷宫出口，让你判断是否可以从入口进来之后，走出迷宫，每次可以向任意方向走。

- 假设是一个10*10的迷宫，入口在(1,1)的位置，出口在(8,10)的位置，通过(1,1)一步可以走到的位置有两个(1,2)，(2,1)
- 但是这两个点并不是出口，需要继续通过这两个位置进一步搜索，假设现在在(1,2),下一次一步可以到达的新的位置为(1,3)，(2,2)。而通过（2，1）可以一步到达的新的位置为（2，2），（3，1），但是这里（2，2）是重复的，所以每一个点在走的过程中需要标记是否已经走过了。
- 两步之后，还没没有走到出口，这时候需要通过新加入的点再去探索下一步能走到哪些新的点上，重复这个过程，直到走到出口为止。

代码解析这个过程，最关键的步骤用当前位置带出新的位置，新的位置可以存放在一个vector或者队列中。位置需要用坐标表示，这里封装出一个node。

```
struct node
{
    int x;
    int y;
};

//queue实现
bool Bfs(vector<vector<int>> graph, int startx, int starty, int destx, int desty)
```

```
{
    //迷宫的大小
    int m = graph.size();
    int n = graph[0].size();

    //存储迷宫中的位置
    queue<node> q;



    //标记迷宫中的位置是否被走过
    vector<vector<int>> book;
    book.resize(m);
    for (size_t i = 0; i < m; i++)
        book[i].resize(n, 0);

    q.push(node(startx, starty));
    //标记已经走过
    book[startx][starty] = 1;
    //四个行走的方向，上下左右
    int next[4][2] = { { -1, 0 }, { 1, 0 }, { 0, -1 }, { 0, 1 } };
    //标记是否可以出去
    bool flag = false;

    while (!q.empty())
    {
        //当前位置带出所有新的位置，可以向上下左右走
        for (size_t i = 0; i < 4; ++i)
        {
            //计算新的位置
            int nx = q.front()._x + next[i][0];
            int ny = q.front()._y + next[i][1];
            //新的位置越界，继续下一个
            if (nx >= m || nx < 0
                || ny >= n || ny < 0)
            {
                continue;
            }
            //如果新的位置无障碍并且之前也没走过，保存新的位置
            if (graph[nx][ny] == 0 && book[nx][ny] == 0)
            {
                q.push(node(nx, ny));
                //标记已被走过
                book[nx][ny] = 1;
            }
            //如果新的位置为目标位置,则结束查找
            if (nx == destx && ny == desty)
            {
                flag = true;
                break;
            }
        }
        if (flag)
```

```
            break;
        //否则，用新的位置继续向后走
        q.pop();
    }
    return flag;
}

int main()
{
    int sx, sy, ex, ey;
    vector<vector<int>> graph;
    int m, n;
    cout << "请输入迷宫的大小：行，列" << endl;
    cin >> m >> n;
    graph.resize(m);
    for (size_t i = 0; i < m; ++i)
    {
        graph[i].resize(n);
    }
    cout << "请输入迷宫的元素" << endl;
    for (size_t i = 0; i < m; ++i)
    {
        for (size_t j = 0; j < n; ++j)
        {
            cin >> graph[i][j];
        }
    }
    while (1)
    {
        cout << "请输入迷宫入口和出口" << endl;
        cin >> sx >> sy >> ex >> ey;

        cout << "是否可以走出迷宫：  " << Bfs(graph, sx, sy, ex, ey) << endl;
    }

    return 0;
}

/*java*/
class node
{
    public  int x;
    public  int y;

    public node(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

class test
{
```

```
//queue实现
public static boolean Bfs(int[][] graph, int startx, int starty, int destx, int desty)
{
    //迷宫的大小
    int m = graph.length;
    int n = graph[0].length;

    //存储迷宫中的位置
    Queue<node> q = new LinkedList<>();
    //标记迷宫中的位置是否被走过
    int[][] book = new int[m][n];

    q.offer(new node(startx, starty));

    //标记已经走过
    book[startx][starty] = 1;
    //四个行走的方向，上下左右
    int[][] next = { { -1, 0 }, { 1, 0 }, { 0, -1 }, { 0, 1 } };
    //标记是否可以出去
    boolean flag = false;

    while (!q.isEmpty())
    {
        //当前位置带出所有新的位置，可以向上下左右走
        for (int i = 0; i < 4; ++i)
        {
            //计算新的位置
            int nx = q.peek().x + next[i][0];
            int ny = q.peek().y + next[i][1];
            //新的位置越界，继续下一个
            if (nx >= m || nx < 0
                    || ny >= n || ny < 0)
            {
                continue;
            }
            //如果新的位置无障碍并且之前也没走过，保存新的位置
            if (graph[nx][ny] == 0 && book[nx][ny] == 0)
            {
                q.offer(new node(nx, ny));
                //标记已被走过
                book[nx][ny] = 1;
            }
            //如果新的位置为目标位置,则结束查找
            if (nx == destx && ny == desty)
            {
                flag = true;
                break;
            }
        }
        if (flag)
            break;
        //否则，用新的位置继续向后走
        q.poll();
```

```
        }
        return flag;
    }

    public static void main(String[] args) {
        int sx, sy, ex, ey;
        int m, n;
        System.out.println("请输入迷宫的大小：行，列" );
        Scanner sc = new Scanner(System.in);
        m = sc.nextInt();
        n = sc.nextInt();

        int[][] graph = new int[m][n];

        System.out.println("请输入迷宫的元素");

        for (int i = 0; i < m; ++i)
        {
            for (int j = 0; j < n; ++j)
            {
                graph[i][j] = sc.nextInt();
            }
        }
        while (true)
        {
            System.out.println("请输入迷宫入口和出口");
            sx = sc.nextInt();
            sy = sc.nextInt();
            ex = sc.nextInt();
            ey = sc.nextInt();

            System.out.print( "是否可以走出迷宫：  " );
            Bfs(graph,sx, sy, ex, ey);
            System.out.println();
        }
    }
}
```

广度优先搜索模型

```
Bfs()
{
    1. 建立起始步骤，队列初始化
    2. 遍历队列中的每一种可能，whlie(队列不为空)
    {
        通过队头元素带出下一步的所有可能，并且依次入队
        {
            判断当前情况是否达成目标：按照目标要求处理逻辑
        }
        继续遍历队列中的剩余情况

    }
}
```

N叉树的层序遍历

```
class Solution {
public:
    vector<vector<int>> levelOrder(Node* root) {

        vector<vector<int>> treeVec;
        if (root == nullptr)
            return treeVec;

        //临时存放每一层的元素
        vector<int> newFloor;

        queue<Node*> q;
        q.push(root);
        while (q.size()){
            //获取当前层元素个数，即整个队列元素
            int size = q.size();
            //存放新层元素之前先清空
            newFloor.clear();
            while (size--){
                auto node = q.front();
                q.pop();
                newFloor.push_back(node->val);
                //孩子入队
                for (auto& child : node->children){
                    if (child)
                        q.push(child);
                }
            }
            //新层有元素,则放入vector
            if (!newFloor.empty())
                treeVec.push_back(newFloor);
        }
        return treeVec;
    }
};
```

```java
/*java*/
class Solution {
    public List<List<Integer>> levelOrder(Node root) {
        List<List<Integer>> treeVec = new ArrayList<>();
        if (root == null)
            return treeVec;

        Queue<Node> q = new LinkedList<>();
        q.offer(root);
        while (!q.isEmpty()){
            //获取当前层元素个数，即整个队列元素
            int size = q.size();
            //临时存放每一层的元素
            List<Integer> newFloor = new ArrayList<>();
            while (size-- != 0){
                Node node = q.peek();
                q.poll();
                newFloor.add(node.val);
                //孩子入队
                for (Node child : node.children){
                    if (child != null)
                        q.offer(child);
                }
            }
            //新层有元素,则放入vector
            if (!newFloor.isEmpty())
                treeVec.add(newFloor);
        }
        return treeVec;
    }
}
```

腐烂的橘子

```java
/*
本题可以先找到所有的腐烂橘子，入队，用第一批带出新一批腐烂的橘子
每以匹橘子都会在一分钟之内腐烂,所以此题可以转化为求BFS执行的大循环的次数
这里的step的更新需要有一个标记，只有新的腐烂的橘子加入，step才能自加

最后BFS执行完之后，说明所有可以被腐烂的都完成了，再去遍历grid,如何还有
值为1的，说明没有办法完全腐烂，返回-1,如果没有，则返回step
*/

class Solution {
public:
    int orangesRotting(vector<vector<int>>& grid) {
        //用pair存放位置
        queue<pair<int, int>> q;
        int row = grid.size();
        int col = grid[0].size();
        //已经腐烂的位置入队
        for (int i = 0; i < row; ++i)
        {
```

```
        for (int j = 0; j < col; ++j)
        {
            if (grid[i][j] == 2)
                q.push(make_pair(i, j));
        }
    }
    //可以蔓延的方向
    static int nextP[4][2] = { { 0, 1 }, { 1, 0 }, { 0, -1 }, { -1, 0 } };
    int step = 0;
    while (!q.empty())
    {
        int n = q.size();
        int flag = 0;
        //用当前这一批已经腐烂的橘子带出下一批要腐烂的橘子
        //故要遍历队列中的所有位置
        while (n--)
        {
            auto Curpos = q.front();
            q.pop();
            //当前位置向四个方向蔓延
            for (int i = 0; i < 4; ++i)
            {
                int nx = Curpos.first + nextP[i][0];
                int ny = Curpos.second + nextP[i][1];
                //如果位置越界或者是空格，或者已经是腐烂的位置，则跳过
                if (nx >= row || nx < 0
                    || ny >= col || ny < 0
                    || grid[nx][ny] != 1)
                    continue;
                //标记有新的被腐烂
                flag = 1;
                grid[nx][ny] = 2;
                q.push(make_pair(nx, ny));


            }
        }
        //如果有新的腐烂，才++
        if(flag)
            ++step;
    }

    //判断是否还有无法腐烂的
    for (int i = 0; i < row; ++i)
    {
        for (int j = 0; j < col; ++j)
        {
            if (grid[i][j] == 1)
                return -1;
        }
    }

    return step;
```

```
    }
};

/*java*/
class pair
{
    public  int x;
    public  int y;

    public pair(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

class Solution {
    public int orangesRotting(int[][] grid) {
        //用Entry存放位置
        Queue<pair> q = new LinkedList<>();
        int row = grid.length;
        int col = grid[0].length;
        //已经腐烂的位置入队
        for (int i = 0; i < row; ++i)
        {
            for (int j = 0; j < col; ++j)
            {
                if (grid[i][j] == 2)
                    q.offer(new pair(i,j));
            }
        }
        //可以蔓延的方向
        int[][] nextP = { { 0, 1 }, { 1, 0 }, { 0, -1 }, { -1, 0 } };
        int step = 0;
        while (!q.isEmpty())
        {
            int n = q.size();
            int flag = 0;
            //用当前这一批已经腐烂的橘子带出下一批要腐烂的橘子
            //故要遍历队列中的所有位置
            while (n-- != 0)
            {
                pair Curpos = q.peek();
                q.poll();
                //当前位置向四个方向蔓延
                for (int i = 0; i < 4; ++i)
                {
                    int nx = Curpos.x + nextP[i][0];
                    int ny = Curpos.y + nextP[i][1];
                    //如果位置越界或者是空格，或者已经是腐烂的位置，则跳过
                    if (nx >= row || nx < 0
                            || ny >= col || ny < 0
                            || grid[nx][ny] != 1)
```

```
                        continue;
                    //标记有新的被腐烂
                    flag = 1;
                    grid[nx][ny] = 2;
                    q.offer(new pair(nx, ny));
                }
            }
            //如果有新的腐烂，才++
            if(flag == 1)
                ++step;
        }

        //判断是否还有无法腐烂的
        for (int i = 0; i < row; ++i)
        {
            for (int j = 0; j < col; ++j)
            {
                if (grid[i][j] == 1)
                    return -1;
            }
        }

        return step;
    }
}
```

单词接龙

```
/*
1.通过BFS，首先用beginWord带出转换一个字母之后所有可能的结果
2.每一步都要把队列中上一步添加的所有单词转换一遍，最短的转换肯定在这些单词当中， 所有这些词的转换只能算一次转换，因为都是上一步转换出来的，这里对于每个单词的每个位置都可以用26个字母进行转换，所以一个单词一次转换的可能有：单词的长度 * 26
3.把转换成功的新词入队，进行下一步的转换
4.最后整个转换的长度就和BFS执行的次数相同

*/
class Solution {
public:
    int ladderLength(string beginWord, string endWord, vector<string>& wordList) {
        //hash表的查询效率最高
        unordered_set<string> wordDict(wordList.begin(), wordList.end());
        //标记单词是否已经访问过，访问过的不再访问
        unordered_set<string> visited;
        visited.insert(beginWord);
        //初始化队列
        queue<string> q;
        q.push(beginWord);
        int res = 1;
        while (!q.empty()){
            int nextSize = q.size();
            //每一步都要把队列中上一步添加的所有单词转换一遍
            //最短的转换肯定在这些单词当中， 所有这些词的转换只能算一次转换
```

```
            //因为都是上一步转换出来的
            while (nextSize--){
                string curWord = q.front();
                q.pop();
                //尝试转换当前单词的每一个位置
                for (int i = 0; i < curWord.size(); i++){
                    string newWord = curWord;
                    //每一个位置用26个字母分别替换
                    for (auto ch = 'a'; ch <= 'z'; ch++){
                        newWord[i] = ch;
                        //如果列表中没有此单词或者已经访问过（它的转换已经遍历过，无需再次遍历），则跳过
                        if (!wordDict.count(newWord) || visited.count(newWord))
                            continue;
                        //转换成功，则在上一步转换的基础上+1
                        if (newWord == endWord)
                            return res + 1;
                        //还没有转换成功，则新的单词入队
                        visited.insert(newWord);
                        q.push(newWord);
                    }
                }
            }
            res++;
        }
        //转换不成功，返回0
        return 0;
    }
};

/*java*/
class Solution {
    public int ladderLength(String beginWord, String endWord, List<String> wordList) {
        //hash表的查询效率最高
        Set<String> wordDict = new HashSet<>();
        for(String wd : wordList)
        {
            wordDict.add(wd);
        }
        //标记单词是否已经访问过，访问过的不再访问
        Set<String> visited = new HashSet<>();
        visited.add(beginWord);
        //初始化队列
        Queue<String> q = new LinkedList<>();
        q.offer(beginWord);
        int res = 1;
        while (!q.isEmpty()) {
            int nextSize = q.size();
            //每一步都要把队列中上一步添加的所有单词转换一遍
            //最短的转换肯定在这些单词当中， 所有这些词的转换只能算一次转换
            //因为都是上一步转换出来的
            while (nextSize-- != 0) {
                String curWord = q.peek();
                q.poll();
```

```
                //尝试转换当前单词的每一个位置
                for (int i = 0; i < curWord.length(); i++) {
                    StringBuilder newWord = new StringBuilder(curWord);
                    //每一个位置用26个字母分别替换
                    for (char ch = 'a'; ch <= 'z'; ch++) {
                        newWord.setCharAt(i, ch);
                        //如果列表中没有此单词或者已经访问过（它的转换已经遍历过，无需再次遍历），则跳过
                        String changeWord = newWord.toString();
                        if (!wordDict.contains(changeWord) || visited.contains(changeWord))
                            continue;
                        //转换成功，则在上一步转换的基础上+1
                        if (changeWord.equals(endWord))
                            return res + 1;
                        //还没有转换成功，则新的单词入队
                        visited.add(changeWord);
                        q.offer(changeWord);
                    }
                }
            }
            res++;
        }
        //转换不成功，返回0
        return 0;
    }
}
```

相似题目：

[最小基因变化]

[打开转盘锁]

```
/*
深度优先不适合解此题，递归深度太大，会导致栈溢出

本题的密码为4位密码，每位密码可以通过拨动一次进行改变，注意这里的数的回环以及拨动的方向
拨动方向：向前，向后
回环：如果当前是9，0时，向前，向后拨动需要变成最小最大，而不是简单的自加自减
*/

class Solution {
public:
    int openLock(vector<string>& deadends, string target) {

        // 哈希表的查找更快
        unordered_set<string> deadendsSet(deadends.begin(), deadends.end());
        //如果"0000"在死亡字符串中，则永远到达不了
        if(deadendsSet.find("0000") != deadendsSet.end())
            return -1;
        //初始化队列
        queue<string> que;
        que.push("0000");
```

```
        //加标记，已经搜索过的字符串不需要再次搜索
        unordered_set<string> book;
        book.insert("0000");

        int step = 0;
        while (!que.empty()) {
            int n = que.size();
            //从上一步转换之后的字符串都需要进行验证和转换
            //并且只算做一次转换，类似于层序遍历，转换的步数和层相同
            //同一层的元素都是经过一步转换得到的
            for (int i = 0; i < n; i++) {
                string curStr = que.front();
                que.pop();

                if (curStr == target) return step;

                //四位密码锁，每个位置每次都可以转一次
                for (int j = 0; j < 4; j++) {
                    string newStr1 = curStr, newStr2 = curStr;
                    //当前位置可以向前或者向后拨一位
                    newStr1[j] = newStr1[j] == '9' ? '0' : newStr1[j] + 1;
                    newStr2[j] = newStr2[j] == '0' ? '9' : newStr2[j] - 1;
                    if (deadendsSet.find(newStr1) == deadendsSet.end()
                        && book.find(newStr1) == book.end()) {
                        que.push(newStr1);
                        book.insert(newStr1);
                    }
                    if (deadendsSet.find(newStr2) == deadendsSet.end()
                        && book.find(newStr2) == book.end()) {
                        que.push(newStr2);
                        book.insert(newStr2);
                    }
                }
            }
            step++;
        }

        return -1;
    }
};

/*java*/
class Solution {
    public int openLock(String[] deadends, String target) {
        // 哈希表的查找更快
        Set<String> deadendsSet = new HashSet<>();
        for(String str : deadends)
        {
            deadendsSet.add(str);
        }
        //如果"0000"在死亡字符串中，则永远到达不了
        if(deadendsSet.contains("0000"))
            return -1;
```

```
        //初始化队列
        Queue<String> que = new LinkedList<>();
        que.offer("0000");

        //加标记，已经搜索过的字符串不需要再次搜索
        Set<String> book = new HashSet<>();
        book.add("0000");

        int step = 0;
        while (!que.isEmpty()) {
            int n = que.size();
            //从上一步转换之后的字符串都需要进行验证和转换
            //并且只算做一次转换，类似于层序遍历，转换的步数和层相同
            //同一层的元素都是经过一步转换得到的
            for (int i = 0; i < n; i++) {
                String curStr = que.peek();
                que.poll();

                if (curStr.equals(target)) return step;

                //四位密码锁，每个位置每次都可以转一次
                for (int j = 0; j < 4; j++) {
                    char newC1 = curStr.charAt(j);
                    char newC2 = curStr.charAt(j);
                    //当前位置可以向前或者向后拨一位
                    if(newC1 == '9')
                        newC1 = '0';
                    else
                        ++newC1;
                    if(newC2 == '0')
                        newC2 = '9';
                    else
                        --newC2;
                    StringBuilder newStr1 = new StringBuilder(curStr);
                    StringBuilder newStr2 = new StringBuilder(curStr);

                    newStr1.setCharAt(j, newC1);
                    newStr2.setCharAt(j, newC2);
                    if (!deadendsSet.contains(newStr1.toString())
                            && !book.contains(newStr1.toString())) {
                        que.offer(newStr1.toString());
                        book.add(newStr1.toString());
                    }
                    if (!deadendsSet.contains(newStr2.toString())
                            && !book.contains(newStr2.toString())) {
                        que.offer(newStr2.toString());
                        book.add(newStr2.toString());
                    }
                }
            }
            step++;
        }
```

```
        return -1;
    }
}
```