C++43期第二组第三次小组分享

> ## by feng

修改自 [面向大象编程]

[搜索思想——DFS & BFS（基础基础篇）]

DFS

本期例题为 LeetCode「岛屿问题」系列：

- **LeetCode 463. Island Perimeter** 岛屿的周长（Easy）
- **LeetCode 695. Max Area of Island** 岛屿的最大面积（Medium）
- **LeetCode 827. Making A Large Island** 填海造陆（Hard）

我们所熟悉的 DFS（深度优先搜索）问题通常是在树或者图结构上进行的。而我们今天要讨论的 DFS 问题，是在一种「网格」结构中进行的。岛屿问题是这类网格 DFS 问题的典型代表。网格结构遍历起来要比二叉树复杂一些，如果没有掌握一定的方法，DFS 代码容易写得冗长繁杂。

本文将以岛屿问题为例，展示网格类问题 DFS 通用思路，以及如何让代码变得简洁。主要内容包括：

- 网格类问题的基本性质
- 在网格中进行 DFS 遍历的方法与技巧
- 三个岛屿问题的解法
- 相关题目

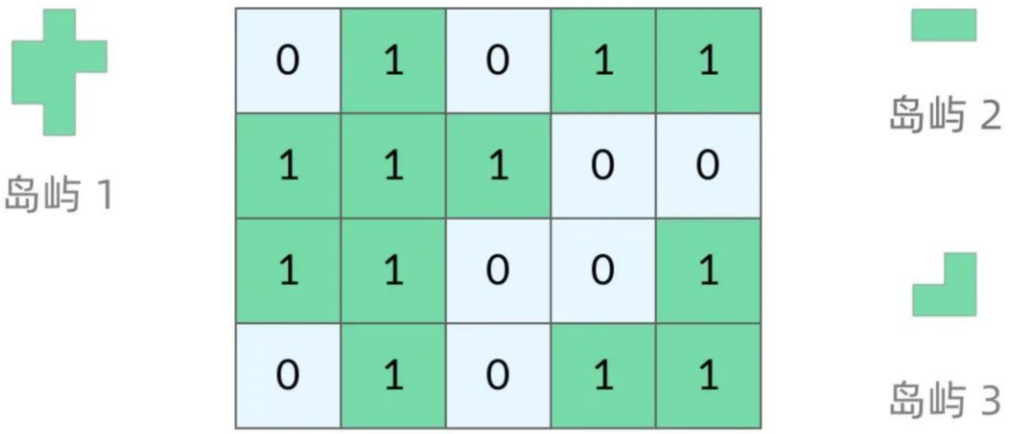
网格类问题的 DFS 遍历方法

网格问题的基本概念

我们首先明确一下岛屿问题中的网格结构是如何定义的，以方便我们后面的讨论。

网格问题是由 个小方格组成一个网格，每个小方格与其上下左右四个方格认为是相邻的，要在这样的网格上进行某种搜索。

岛屿问题是一类典型的网格问题。每个格子中的数字可能是 0 或者 1。我们把数字为 0 的格子看成海洋格子，数字为 1 的格子看成陆地格子，这样相邻的陆地格子就连接成一个岛屿。



岛屿问题示例

在这样一个设定下，就出现了各种岛屿问题的变种，包括岛屿的数量、面积、周长等。不过这些问题，基本都可以用 DFS 遍历来解决。

DFS 的基本结构

网格结构要比二叉树结构稍微复杂一些，它其实是一种简化版的**图**结构。要写好网格上的 DFS 遍历，我们首先要理解二叉树上的 DFS 遍历方法，再类比写出网格结构上的 DFS 遍历。我们写的二叉树 DFS 遍历一般是这样的：

```
void traverse(TreeNode root) {
    // 判断 base case
    if (root == null) {
        return;
    }
    // 访问两个相邻结点：左子结点、右子结点
    traverse(root->left);
    traverse(root->right);
}
```

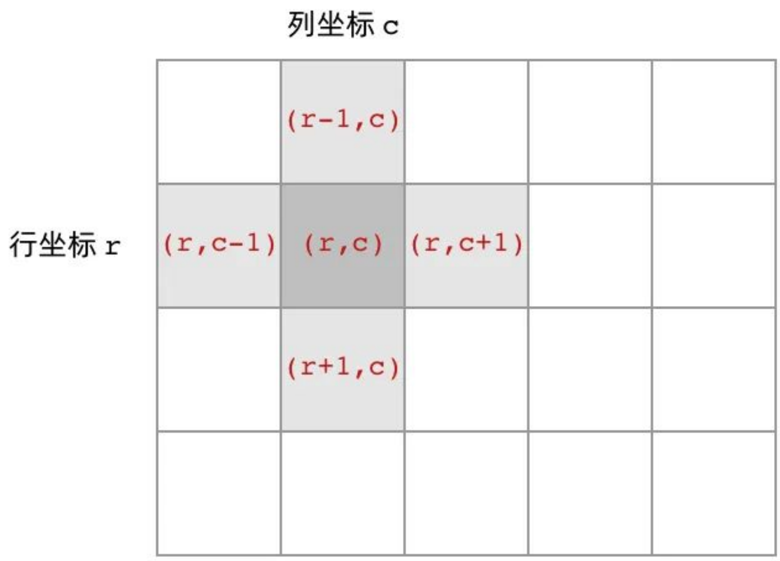
可以看到，二叉树的 DFS 有两个要素：「**访问相邻结点**」和「**判断 base case**」。

第一个要素是**访问相邻结点**。二叉树的相邻结点非常简单，只有左子结点和右子结点两个。二叉树本身就是一个递归定义的结构：一棵二叉树，它的左子树和右子树也是一棵二叉树。那么我们的 DFS 遍历只需要递归调用左子树和右子树即可。

第二个要素是 **判断 base case**。一般来说，二叉树遍历的 base case 是 `root == null`。这样一个条件判断其实有两个含义：一方面，这表示 `root` 指向的子树为空，不需要再往下遍历了。另一方面，在 `root == null` 的时候及时返回，可以让后面的 `root->left` 和 `root->right` 操作不会出现空指针异常。

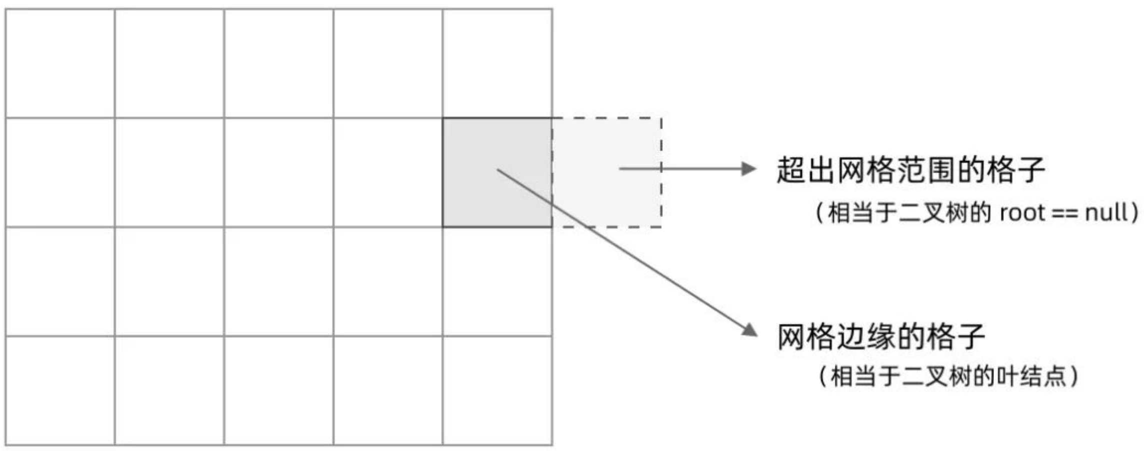
对于网格上的 DFS，我们完全可以参考二叉树的 DFS，写出网格 DFS 的两个要素：

首先，网格结构中的格子有多少相邻结点？答案是上下左右四个。对于格子 `(r, c)` 来说（r 和 c 分别代表行坐标和列坐标），四个相邻的格子分别是 `(r-1, c)`、`(r+1, c)`、`(r, c-1)`、`(r, c+1)`。换句话说，网格结构是「四叉」的。



网格结构中四个相邻的格子

其次，网格 DFS 中的 base case 是什么？从二叉树的 base case 对应过来，应该是网格中不需要继续遍历、`grid[r][c]` 会出现数组下标越界异常的格子，也就是那些超出网格范围的格子。



网格 DFS 的 base case

这一点稍微有些反直觉，坐标竟然可以临时超出网格的范围？这种方法可类比于「先污染后治理」——不管当前是在哪个格子，先往四个方向走一步再说，如果发现走出了网格范围再赶紧返回。这跟二叉树的遍历方法是一样的，先递归调用，发现 `root == null` 再返回。

这样，我们得到了网格 DFS 遍历的框架代码：

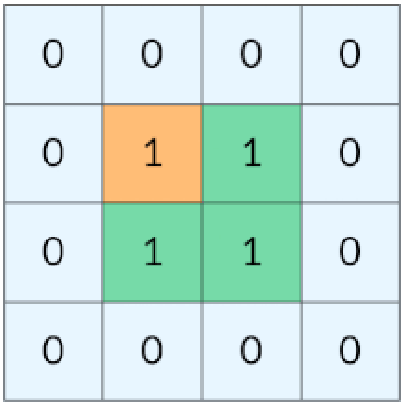
```
void dfs(vector<vector<int>> &grid, int r, int c) {
    // 判断 base case
    // 如果坐标 (r, c) 超出了网格范围，直接返回
    if (!inGraph(grid, r, c)) {
        return;
    }
    // 访问上、下、左、右四个相邻结点
    dfs(grid, r - 1, c);
    dfs(grid, r + 1, c);
    dfs(grid, r, c - 1);
    dfs(grid, r, c + 1);
}

// 判断坐标 (r, c) 是否在网格中
```

```
boolean inGraph(vector<vector<int>> &grid, int r, int c) {
    return 0 <= r && r < grid.size()
           && 0 <= c && c < grid[0].size();
}
```

如何避免重复遍历

网格结构的 DFS 与二叉树的 DFS 最大的不同之处在于，遍历中可能遇到遍历过的结点。这是因为，网格结构本质上是一个「图」，我们可以把每个格子看成图中的结点，每个结点有向上下左右的四条边。在图中遍历时，自然可能遇到重复遍历结点。

这时候，DFS 可能会不停地「兜圈子」，永远停不下来，如下图所示：



DFS 遍历可能会兜圈子

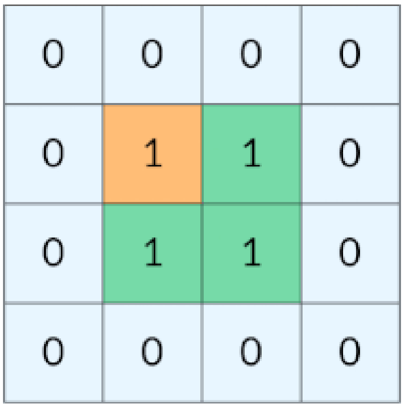
如何避免这样的重复遍历呢？答案是标记已经遍历过的格子。以岛屿问题为例，我们需要在所有值为 1 的陆地格子上做 DFS 遍历。每走过一个陆地格子，就把格子的值改为 2，这样当我们遇到 2 的时候，就知道这是遍历过的格子了。也就是说，每个格子可能取三个值：

- 0 —— 海洋格子
- 1 —— 陆地格子（未遍历过）
- 2 —— 陆地格子（已遍历过）

我们在框架代码中加入避免重复遍历的语句：

```
void dfs(vector<vector<int>> &grid, int r, int c) {
    // 判断 base case
    if (!inGraph(grid, r, c)) {
        return;
    }
    // 如果这个格子不是岛屿，直接返回
    if (grid[r][c] != 1) {
        return;
    }
    grid[r][c] = 2; // 将格子标记为「已遍历过」

    // 访问上、下、左、右四个相邻结点
    dfs(grid, r - 1, c);
    dfs(grid, r + 1, c);
    dfs(grid, r, c - 1);
    dfs(grid, r, c + 1);
}

```

```
// 判断坐标 (r, c) 是否在网格中
boolean inGraph(vector<vector<int>> &grid, int r, int c) {
    return 0 <= r && r < grid.size()
        && 0 <= c && c < grid[0].size();
}
```



标记已遍历的格子

这样，我们就得到了一个岛屿问题、乃至各种网格问题的通用 DFS 遍历方法。以下所讲的几个例题，其实都只需要在 DFS 遍历框架上稍加修改而已。

> **小贴士：**
>
> 在一些题解中，可能会把「已遍历过的陆地格子」标记为和海洋格子一样的 0，美其名曰「陆地沉没方法」，即遍历完一个陆地格子就让陆地「沉没」为海洋。这种方法看似很巧妙，但实际上有很大隐患，因为这样我们就无法区分「海洋格子」和「已遍历过的陆地格子」了。如果题目更复杂一点，这很容易出 bug。

岛屿问题的解法

理解了网格结构的 DFS 遍历方法以后，岛屿问题就不难解决了。下面我们分别看看三个题目该如何用 DFS 遍历来求解。

例题 1：岛屿的最大面积

LeetCode 695. Max Area of Island （Medium）

> 给定一个包含了一些 0 和 1 的非空二维数组 `grid`，一个岛屿是一组相邻的 1（代表陆地），这里的「相邻」要求两个 1 必须在水平或者竖直方向上相邻。你可以假设 `grid` 的四个边缘都被 0（代表海洋）包围着。
>
> 找到给定的二维数组中最大的岛屿面积。如果没有岛屿，则返回面积为 0 。

这道题目只需要对每个岛屿做 DFS 遍历，求出每个岛屿的面积就可以了。求岛屿面积的方法也很简单，代码如下，每遍历到一个格子，就把面积+1。

```
int area(vector<vector<int>> &grid, int r, int c) {
    return 1
        + area(grid, r - 1, c)
        + area(grid, r + 1, c)
        + area(grid, r, c - 1)
        + area(grid, r, c + 1);
}
```

最终我们得到的完整题解代码如下：

```
class Solution {
public:
    int maxAreaOfIsland(vector<vector<int>>& grid) {
        int res = 0;
        for (int r = 0; r < grid.size(); r++) {
            for (int c = 0; c < grid[0].size(); c++) {
                if (grid[r][c] == 1) {
                    int tmpArea = area(grid, r, c);
                    res = max(res, tmpArea);
                }
            }
        }
        return res;
    }

    int area(vector<vector<int>> &grid, int r, int c) {
        if (!inGraph(grid, r, c)) {
            return 0;
        }
        if (grid[r][c] != 1) {
            return 0;
        }

        // 1 才是岛屿

        // 标记 已访问
        grid[r][c] = 2;

        return 1
            + area(grid, r - 1, c)
            + area(grid, r + 1, c)
            + area(grid, r, c - 1)
            + area(grid, r, c + 1);
    }

    bool inGraph(vector<vector<int>> &grid, int r, int c) {
        return 0 <= r && r < grid.size()
             && 0 <= c && c < grid[0].size();
    }
};
```

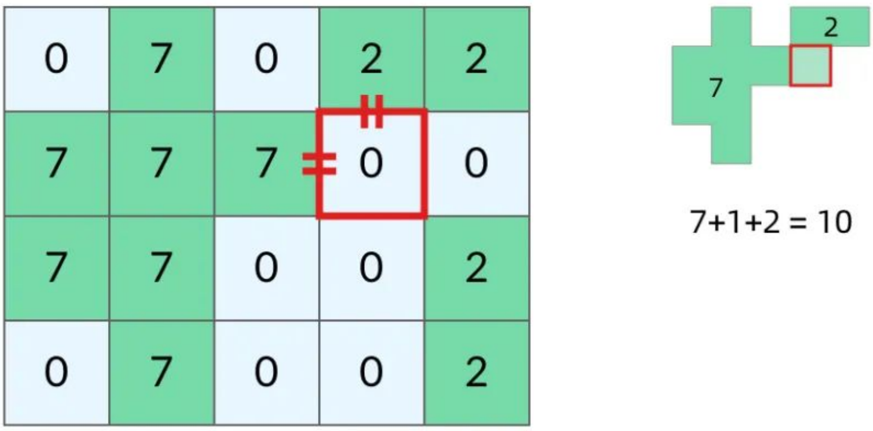
例题 2：填海造陆问题

LeetCode 827. Making A Large Island （Hard）

> 在二维地图上， 0 代表海洋，1代表陆地，我们最多只能将一格 0 （海洋）变成 1 （陆地）。进行填海之后，地图上最大的岛屿面积是多少？

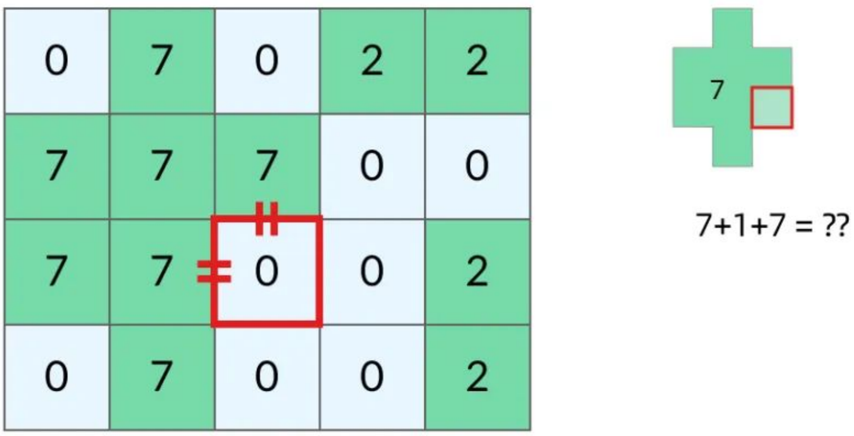
这道题是岛屿最大面积问题的升级版。现在我们有填海造陆的能力，可以把一个海洋格子变成陆地格子，进而让两块岛屿连成一块。那么填海造陆之后，最大可能构造出多大的岛屿呢？

大致的思路我们不难想到，我们先计算出所有岛屿的面积，在所有的格子上标记出岛屿的面积。然后搜索**哪个海洋格子相邻的两个岛屿面积最大**。例如下图中红色方框内的海洋格子，上边、左边都与岛屿相邻，我们可以计算出它变成陆地之后可以连接成的新岛屿面积 。



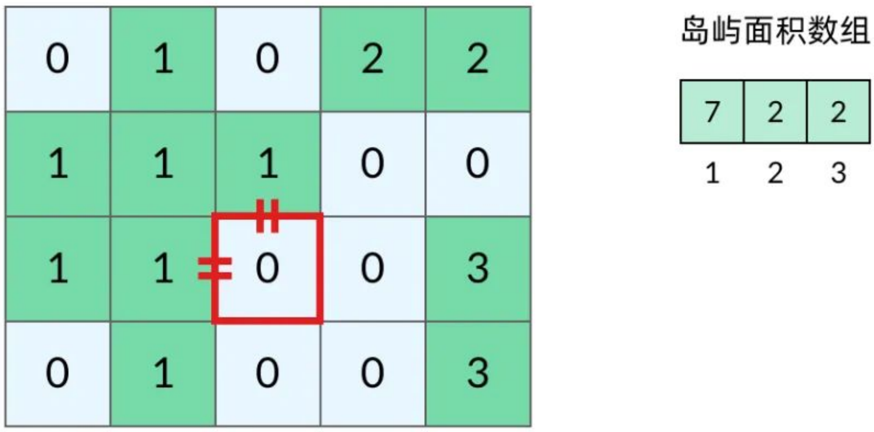
一个海洋格子连接起两个岛屿

然而，这种做法可能遇到一个问题。如下图中红色方框内的海洋格子，它的上边、左边都与岛屿相邻，这时候连接成的岛屿面积难道是 7 + 1 + 7？显然不是。这两个 7 来自同一个岛屿，所以填海造陆之后得到的岛屿面积应该只有 8 。



一个海洋格子与同一个岛屿有两个边相邻

可以看到，要让算法正确，我们得能区分一个海洋格子相邻的两个 7 是不是来自同一个岛屿。那么，我们不能在方格中标记岛屿的面积，而应该标记岛屿的索引（下标），另外用一个数组记录每个岛屿的面积，如下图所示。这样我们就可以发现红色方框内的海洋格子，它的「两个」相邻的岛屿实际上是同一个。



标记每个岛屿的索引（下标）

可以看到，这道题实际上是对网格做了两遍 DFS：第一遍 DFS 遍历陆地格子，计算每个岛屿的面积并标记岛屿；第二遍 DFS 遍历海洋格子，观察每个海洋格子相邻的陆地格子。

```
class Solution {
public:
    int largestIsland(vector<vector<int>>& grid) {
        int row = grid.size();
        int col = grid[0].size();
        //岛屿块索引值 从2开始，避免与0（海洋）,1（陆地）值冲突
        int index = 2;
        int max_area = 0;  // max_are 记录最大的岛屿块，返回值初始化的时候设置这个，处理全是岛屿的case
        if(row == 0) {
            return 1;
        }

        // record(index area) 记录每个岛的索引(从2开始)和面积
        unordered_map<int, int> record;
        //1.深度优先，先遍历出陆地面积
        for(int i = 0; i < row; ++i) {
            for(int j = 0; j < col; ++j) {
                // 1 代表岛屿
                if(grid[i][j] == 1) {
                    // 该岛屿的面积 加入unordered_map
                    int size = dfs(grid, i, j, index);
                    record[index] = size;
                    max_area = max(max_area, size);
                    ++index;
                }
            }
        }
        //可能都没有海洋，全是陆地
        if(max_area == 0) {
            return 1;
        }

        //2.遍历海洋，找到相邻陆地面积最大的海洋格子
        for(int i = 0; i < row; ++i) {
            for(int j = 0; j < col; ++j) {
                if(grid[i][j] == 0) {

```

```
                    set<int> neighbors = findNeighbour(grid, i, j);
                    if(neighbors.size() < 1) {
                        continue;
                    }
                    set<int>::iterator it = neighbors.begin();
                    int area = 1;
                    // 遍历unordered_map 中的所有岛屿，更新最大面积
                    for(; it != neighbors.end(); ++it) {
                        area += record[*it];
                    }
                    max_area = max(max_area, area);
                }
            }
        }
        return max_area;
    }


private:
    bool inArea(vector<vector<int>>& grid, int row, int col) {
        return row >=0 && col >= 0 && row < grid.size() && col < grid[0].size();
    }
    int dfs(vector<vector<int>>& grid, int row, int col, int index) {
        if(!inArea(grid, row, col)) {
            return 0;
        }
        if(grid[row][col] != 1) {
            return 0;
        }
        grid[row][col] = index;//用index来代表同一块陆地存放面积的索引值
        return (1
                + dfs(grid, row + 1, col, index)
                + dfs(grid, row - 1, col, index)
                + dfs(grid, row, col + 1, index)
                + dfs(grid, row, col - 1, index));
    }
    set<int> findNeighbour(vector<vector<int>>& grid, int row, int col) {
        //叠加面积很关键的一步，记得去重，因为海洋四周接壤的土地有可能是同一块陆地
        set<int> hashSet;
        if(inArea(grid, row+1, col) && grid[row+1][col] != 0) {
            hashSet.insert(grid[row+1][col]);//把岛屿编号放到set中去重
        }
        if(inArea(grid, row-1,col) && grid[row-1][col] != 0) {
            hashSet.insert(grid[row-1][col]);
        }
        if(inArea(grid, row, col+1) && grid[row][col+1] != 0) {
            hashSet.insert(grid[row][col+1]);
        }
        if(inArea(grid, row, col-1) && grid[row][col-1] != 0) {
            hashSet.insert(grid[row][col-1]);
        }
        return hashSet;
    }

};
```

例题 3：岛屿的周长

LeetCode 463. Island Perimeter （Easy）

> 给定一个包含 0 和 1 的二维网格地图，其中 1 表示陆地，0 表示海洋。网格中的格子水平和垂直方向相连（对角线方向不相连）。整个网格被水完全包围，但其中恰好有一个岛屿（一个或多个表示陆地的格子相连组成岛屿）。
>
> 岛屿中没有“湖”（“湖” 指水域在岛屿内部且不和岛屿周围的水相连）。格子是边长为 1 的正方形。计算这个岛屿的周长。
>
> 题目示例

实话说，这道题用 DFS 来解并不是最优的方法。对于岛屿，直接用数学的方法求周长会更容易。不过这道题是一个很好的理解 DFS 遍历过程的例题。

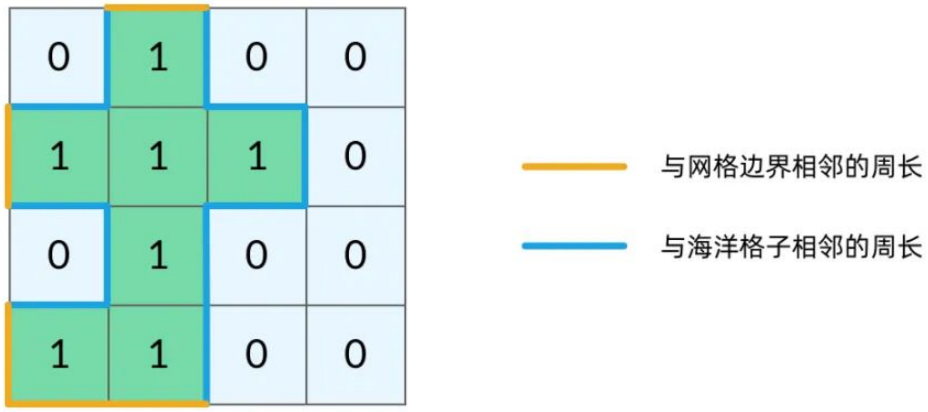
我们再回顾一下 网格 DFS 遍历的基本框架：

```
void dfs(vector<vector<int>> &grid, int r, int c) {
    // 判断 base case
    if (!inArea(grid, r, c)) {
        return;
    }
    // 如果这个格子不是岛屿，直接返回
    if (grid[r][c] != 1) {
        return;
    }
    grid[r][c] = 2; // 将格子标记为「已遍历过」

    // 访问上、下、左、右四个相邻结点
    dfs(grid, r - 1, c);
    dfs(grid, r + 1, c);
    dfs(grid, r, c - 1);
    dfs(grid, r, c + 1);
}

// 判断坐标 (r, c) 是否在网格中
bool inArea(vector<vector<int>> &grid, int r, int c) {
    return 0 <= r && r < grid.size()
          && 0 <= c && c < grid[0].size();
}
```

可以看到，`dfs` 函数直接返回有这几种情况：

- `!inArea(grid, r, c)`，即坐标 `(r, c)` 超出了网格的范围，也就是所谓的「先污染后治理」的情况
- `grid[r][c] != 1`，即当前格子不是岛屿格子，这又分为两种情况：
-
    - `grid[r][c] == 0`，当前格子是海洋格子
    - `grid[r][c] == 2`，当前格子是已遍历的陆地格子

那么这些和我们岛屿的周长有什么关系呢？实际上，岛屿的周长是计算岛屿全部的「边缘」，而这些边缘就是我们在 DFS 遍历中，`dfs` 函数返回的位置。观察题目示例，我们可以将岛屿的周长中的边分为两类，如下图所示。黄色的边是与网格边界相邻的周长，而蓝色的边是与海洋格子相邻的周长。



将岛屿周长中的边分为两类

当我们的 `dfs` 函数因为「坐标 `(r, c)` 超出网格范围」返回的时候，实际上就经过了一条黄色的边；而当函数因为「当前格子是海洋格子」返回的时候，实际上就经过了一条蓝色的边。这样，我们就把岛屿的周长跟 DFS 遍历联系起来了，我们的题解代码如下：

```
class Solution{
public:
    int islandPerimeter(vector<vector<int>> &grid) {
        for (int r = 0; r < grid.size(); r++) {
            for (int c = 0; c < grid[0].size(); c++) {
                if (grid[r][c] == 1) {
                    // 题目限制只有一个岛屿，计算一个即可
                    return dfs(grid, r, c);
                }
            }
        }
        return 0;
    }

private:
    int dfs(vector<vector<int>> &grid, int r, int c) {
        // 函数因为「坐标 (r, c) 超出网格范围」返回，对应一条黄色的边
        if (!inArea(grid, r, c)) {
            return 1;
        }
        // 函数因为「当前格子是海洋格子」返回，对应一条蓝色的边
        if (grid[r][c] == 0) {
            return 1;
        }
        // 函数因为「当前格子是已遍历的陆地格子」返回，和周长没关系
        if (grid[r][c] != 1) {
            return 0;
        }
        grid[r][c] = 2;
        return dfs(grid, r - 1, c)
            + dfs(grid, r + 1, c)
            + dfs(grid, r, c - 1)
            + dfs(grid, r, c + 1);
    }

    // 判断坐标 (r, c) 是否在网格中
    bool inArea(vector<vector<int>> &grid, int r, int c) {
```

```
        return 0 <= r && r < grid.size()
            && 0 <= c && c < grid[0].size();
    }

};
```

总结

对比完三个例题的题解代码，你会发现网格问题的代码真的都非常相似。其实这一类问题属于「会了不难」类型。了解树、图的基本遍历方法，再学会一点小技巧，掌握网格 DFS 遍历就一点也不难了。

岛屿类问题是网格类问题中的典型代表，做了几道岛屿类问题，再看其他的问题其实本质上和岛屿问题是一样的，例如 **LeetCode 130. Surrounded Regions** 这道题，将岛屿的 1 和 0 转换为字母 O 和 X，但本质上是完全一样的。

BFS

DFS（深度优先搜索）和 BFS（广度优先搜索）就像孪生兄弟，提到一个总是想起另一个。然而在实际使用中，我们用 DFS 的时候远远多于 BFS。那么，是不是 BFS 就没有什么用呢?

如果我们使用 DFS/BFS 只是为了遍历一棵树、一张图上的所有结点的话，那么 DFS 和 BFS 的能力没什么差别，我们当然更倾向于更方便写、空间复杂度更低的 DFS 遍历。不过，某些使用场景是 DFS 做不到的，只能使用 BFS 遍历。这就是本文要介绍的两个场景：「层序遍历」、「最短路径」。

本文包括以下内容:

- DFS 与 BFS 的特点比较
- BFS 的适用场景
- 如何用 BFS 进行层序遍历
- 如何用 BFS 求解最短路径问题

DFS 与 BFS

让我们先看看在二叉树上进行 DFS 遍历和 BFS 遍历的代码比较。

DFS 遍历使用**递归**:

```
void dfs(TreeNode root) {
    if (root == NULL) {
        return;
    }
    dfs(root->left);
    dfs(root->right);
}
```

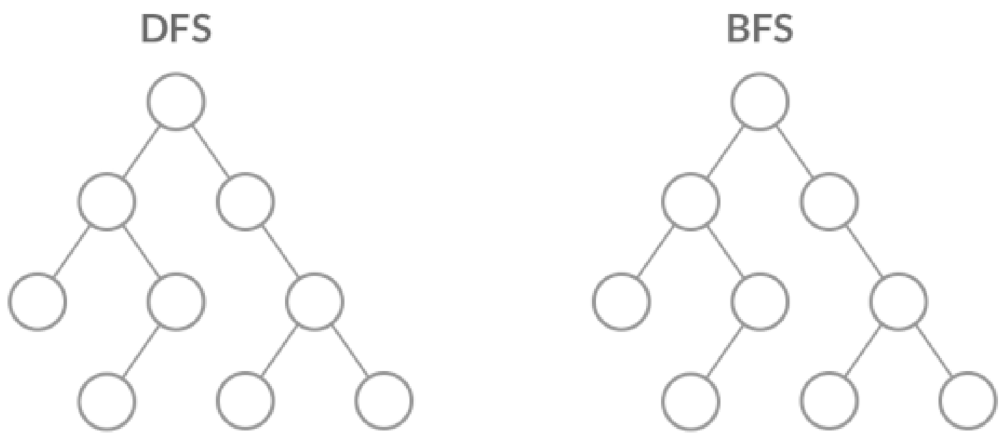
BFS 遍历使用**队列**数据结构:

```
void bfs(TreeNode root) {
    queue<TreeNode> q;
```

```
    q.push(root);
    while (!queue.isEmpty()) {
        // 队首元素 出队
        TreeNode node = q.pop();

        // 该元素如果有左孩子，左孩子入队
        if (node->left != null) {
            q.push(node->left);
        }
        // 该元素如果有右孩子，右孩子入队
        if (node->right != null) {
            q.push(node->right);
        }

        // 广度/宽度遍历，最后剩下的是一些叶子结点(高度大的结点)

    }
    // 结束时， 队列为空
}
```

只是比较两段代码的话，最直观的感受就是：DFS 遍历的代码比 BFS 简洁太多了！这是因为递归的方式隐含地使用了系统的**栈**，我们不需要自己维护一个数据结构。如果只是简单地将二叉树遍历一遍，那么 DFS 显然是更方便的选择。

虽然 DFS 与 BFS 都是将二叉树的所有结点遍历了一遍，但它们遍历结点的顺序不同。



DFS 与 BFS 对比

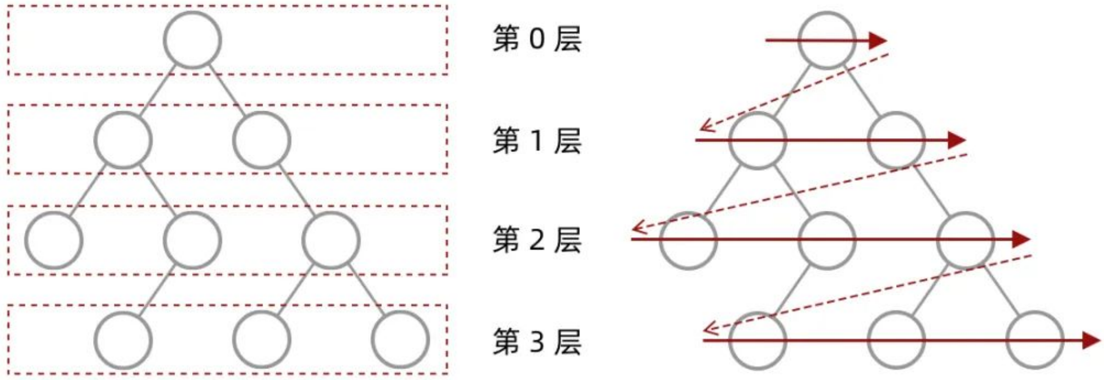
这个遍历顺序也是 BFS 能够用来解「层序遍历」、「最短路径」问题的根本原因。下面，我们结合几道例题来讲讲 BFS 是如何求解层序遍历和最短路径问题的。

BFS 的应用一：层序遍历

LeetCode 102. Binary Tree Level Order Traversal 二叉树的层序遍历（Medium）

> 给定一个二叉树，返回其按层序遍历得到的节点值。层序遍历即逐层地、从左到右访问所有结点。

什么是层序遍历呢？简单来说，层序遍历就是把二叉树分层，然后每一层从左到右遍历：



二叉树的层序遍历

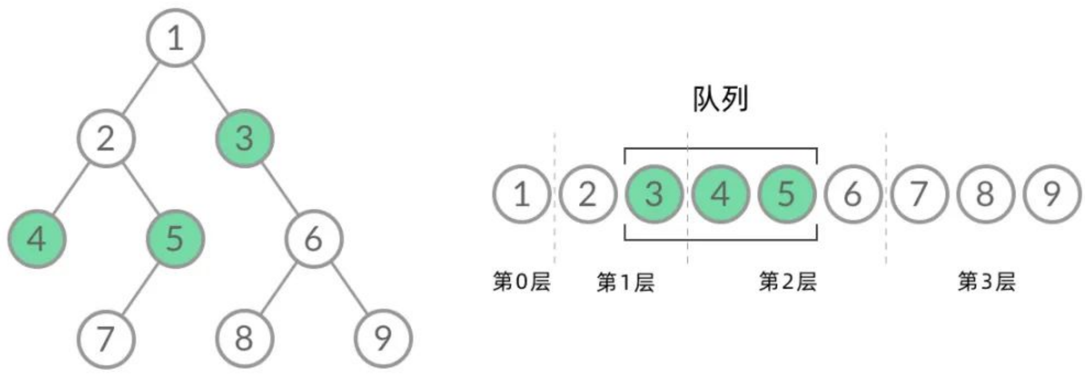
乍一看来，这个遍历顺序和 BFS 是一样的，我们可以直接用 BFS 得出层序遍历结果。然而，层序遍历要求的输入结果和 BFS 是不同的。层序遍历要求我们区分每一层，也就是返回一个二维数组。而 BFS 的遍历结果是一个一维数组，无法区分每一层。

BFS 遍历与层序遍历的输出结果不同

那么，怎么给 BFS 遍历的结果分层呢？我们首先来观察一下 BFS 遍历的过程中，结点进队列和出队列的过程：

BFS 遍历的过程

截取 BFS 遍历过程中的某个时刻：



BFS 遍历中某个时刻队列的状态

可以看到，此时队列中的结点是 3、4、5，分别来自第 1 层和第 2 层。这个时候，第 1 层的结点还没出完，第 2 层的结点就进来了，而且两层的结点在队列中紧挨在一起，我们**无法区分队列中的结点来自哪一层。**

因此，我们需要稍微修改一下代码，在每一层遍历开始前，先记录队列中的结点数量（也就是这一层的结点数量），然后一口气处理完这一层的`n`个结点。

```
// 二叉树的层序遍历
void bfs(TreeNode root) {
    queue<TreeNode> q;
    q.push(root);
    while (!q.empty()) {
        int n = q.size();
        for (int i = 0; i < n; i++) {
```

```
            // 变量 i 无实际意义，只是为了循环 n 次
            TreeNode node = q.pop();
            if (node->left != null) {
                q.push(node->left);
            }
            if (node->right != null) {
                q.push(node->right);
            }
        }
    }
}
```

这样，我们就将 BFS 遍历改造成了层序遍历。在遍历的过程中，结点进队列和出队列的过程为：

BFS 层序遍历的过程

可以看到，在 while 循环的每一轮中，都是将当前层的所有结点出队列，再将下一层的所有结点入队列，这样就实现了层序遍历。

最终我们得到的题解代码为：

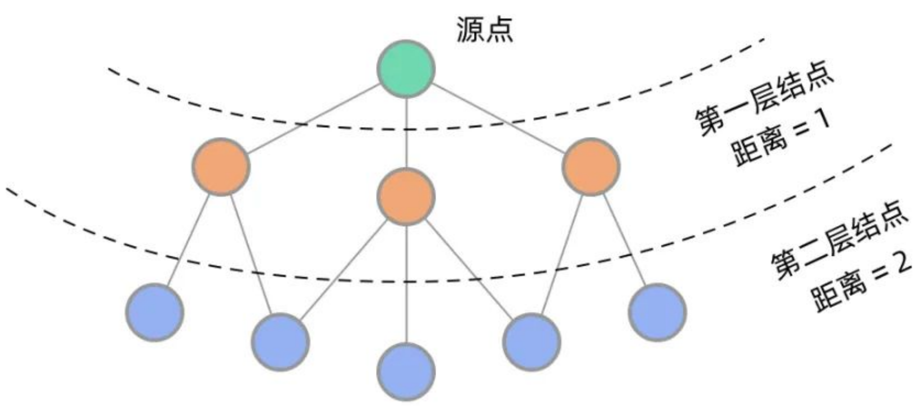
```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        // 保存答案
        vector<vector<int>> res;

        queue<TreeNode*> q;
        if (root != nullptr)
            q.push(root);
        while (!q.empty()) {
```

```
            // 记录此层结点个数
            int n = q.size();
            // 用来保存此层结点的值
            vector<int> level;
            for (int i = 0; i < n; ++i) {
                TreeNode* node = q.front();
                q.pop();
                level.push_back(node->val);
                if (node->left != nullptr)
                    q.push(node->left);
                if (node->right != nullptr)
                    q.push(node->right);
            }
            // 将下一层结点入队，并将此层结果保存下来
            res.push_back(level);
        }
        return res;
    }
};
```

BFS 的应用二：最短路径

在一棵树中，一个结点到另一个结点的路径是唯一的，但在图中，结点之间可能有多条路径，其中哪条路最近呢？这一类问题称为**最短路径问题**。最短路径问题也是 BFS 的典型应用，而且其方法与层序遍历关系密切。

在二叉树中，BFS 可以实现一层一层的遍历。在图中同样如此。从源点出发，BFS 首先遍历到第一层结点，到源点的距离为 1，然后遍历到第二层结点，到源点的距离为 2…… 可以看到，用 BFS 的话，距离源点更近的点会先被遍历到，这样就能找到到某个点的最短路径了。



层序遍历与最短路径

> **小贴士：**
>
> 很多同学一看到「最短路径」，就条件反射地想到「Dijkstra 算法」。为什么 BFS 遍历也能找到最短路径呢？
>
> 这是因为，Dijkstra 算法解决的是**带权最短路径问题**，而我们这里关注的是**无权最短路径问题**。也可以看成每条边的权重都是 1。这样的最短路径问题，用 BFS 求解就行了。
>
> 在面试中，你可能更希望写 BFS 而不是 Dijkstra。毕竟，敢保证自己能写对 Dijkstra 算法的人不多。

最短路径问题属于图算法。由于图的表示和描述比较复杂，本文用比较简单的网格结构代替。网格结构是一种特殊的图，它的表示和遍历都比较简单，适合作为练习题。在 LeetCode 中，最短路径问题也以网格结构为主。

最短路径例题讲解

LeetCode 1162. As Far from Land as Possible 离开陆地的最远距离（Medium）

> 你现在手里有一份大小为 的地图网格 `grid`，上面的每个单元格都标记为 0 或者 1，其中 0 代表海洋，1 代表陆地，请你找出一个海洋区域，这个海洋区域到离它最近的陆地区域的距离是最大的。
>
> 我们这里说的距离是「曼哈顿距离」。 和 这两个区域之间的距离是 。
>
> 如果我们的地图上只有陆地或者海洋，请返回 -1。

这道题就是一个在网格结构中求最短路径的问题。同时，它也是一个「岛屿问题」，即用网格中的 1 和 0 表示陆地和海洋，模拟出若干个岛屿。

在上一篇文章中，我们介绍了网格结构的基本概念，以及网格结构中的 DFS 遍历。其中一些概念和技巧也可以用在 BFS 遍历中：

- 格子 `(r, c)` 的相邻四个格子为：`(r-1, c)`、`(r+1, c)`、`(r, c-1)` 和 `(r, c+1)`；
- 使用函数 `inArea` 判断当前格子的坐标是否在网格范围内；
- 将遍历过的格子标记为 2，避免重复遍历。

网格结构的 BFS 遍历: 要解最短路径问题，我们首先要写出层序遍历的代码，仿照上面的二叉树层序遍历代码，类似地可以写出网格层序遍历：

```
class Solution {
public:
    int maxDistance(vector<vector<int>>& grid) {
        const int M = grid.size();
        const int N = grid[0].size();
        // 使用queue (单个元素是二维坐标，所以用pair<int, int>)
        queue<pair<int, int>> q;
        for (int i = 0; i < M; ++i) {
            for (int j = 0; j < N; ++j) {
                if (grid[i][j] == 1) {
                    // 将所有陆地都放入队列中
                    q.push({i, j});
                }
            }
        }
        // 如果没有陆地或者海洋，返回-1
        if (q.size() == 0 || q.size() == M * N) {
            return -1;
        }
        // 由于BFS的第一层遍历是从陆地开始，因此遍历完第一层之后distance应该是0
        int distance = -1;
        // 对队列的元素进行遍历
        while (q.size() != 0) {
            // 新遍历了一层
            distance ++;
            // 当前层的元素有多少，在该轮中一次性遍历完当前层
            int size = q.size();
            while (size --) {
                // BFS遍历的当前元素永远是队列的开头元素
```

```
                auto cur = q.front();
                q.pop();
                // 对当前元素的各个方向进行搜索
                for (auto& d : directions) {
                    int x = cur.first + d[0];
                    int y = cur.second + d[1];
                    // 如果搜索到的新坐标超出范围/陆地/已经遍历过，则不搜索了
                    if (x < 0 || x >= M || y < 0 || y >= N ||
                        grid[x][y] != 0) {
                        continue;
                    }
                    // 把grid中搜索过的元素设置为2
                    grid[x][y] = 2;
                    // 放入队列中
                    q.push({x, y});
                }
            }
        }
        // 最终走了多少层才把海洋遍历完
        return distance;
    }
private:
    vector<vector<int>> directions = {{-1, 0}, {1, 0}, {0, 1}, {0, -1}};
};
```

以上代码有几个注意点：

- 队列中的元素类型是 `pair<int, int>` ，包含格子的行坐标和列坐标。
- 为了避免重复遍历，这里使用到了和 DFS 遍历一样的技巧：把已遍历的格子标记为 2。注意：我们在将格子放入队列之前就将其标记为 2。防止遍历到访问过的元素。
- 在将格子放入队列之前就检查其坐标是否在网格范围内，避免将「不存在」的格子放入队列。

此类广搜的一个常用技巧：

由于一个格子有四个相邻的格子，代码中判断了四遍格子坐标的合法性，代码稍微有点啰嗦。我们可以用一个 `directions` 二维数组存储相邻格子的四个方向：

```
vector<vector<int>> directions = {{-1, 0}, {1, 0}, {0, 1}, {0, -1}};
```

然后把四个 if 判断变成一个循环：

```
// 对当前元素的各个方向进行搜索
for (auto& d : directions) {
    int x = cur.first + d[0];
    int y = cur.second + d[1];
    // 如果搜索到的新坐标超出范围/陆地/已经遍历过，则不搜索
    if (x < 0 || x >= M || y < 0 || y >= N || grid[x][y] != 0) {
        continue;
    }
    // 把grid中搜索过的元素设置为2
    grid[x][y] = 2;
    // 放入队列中
    q.push({x, y});
}
```

这道题要找的是距离陆地最远的海洋格子。假设网格中只有一个陆地格子，我们可以从这个陆地格子出发做层序遍历，直到所有格子都遍历完。最终遍历了几层，海洋格子的最远距离就是几。

0	0	0	0	0
0	0	0	0	0
0	0	1	0	0
0	0	0	0	0
0	0	0	0	0

从单个陆地格子出发的距离

那么有多个陆地格子的时候怎么办呢？一种方法是将每个陆地格子都作为起点做一次“层序遍历”，但是这样的时间开销太大。

BFS 完全可以以多个格子同时作为起点。我们可以把所有的陆地格子同时放入初始队列，然后开始“层序遍历”，这样遍历的效果如下图所示：

1	0	0	0	0
0	0	0	0	1
0	0	0	0	0
0	0	0	0	0
1	0	0	0	0

从多个陆地格子出发的距离

这种遍历方法实际上叫做「**多源 BFS**」。多源 BFS 的定义不是今天讨论的重点，多源 BFS 写法区别不大，只需要**把多个源点同时放入初始队列**即可。

需要注意的是，虽然上面的图示用 1、2、3、4 表示层序遍历的层数，但是在代码中，我们不需要给每个遍历到的格子标记层数，只需要用一个 `distance` 变量记录当前的遍历的层数（也就是到陆地格子的距离）即可。

总结

可以看到，「BFS 遍历」、「层序遍历」、「最短路径」实际上是递进的关系。在 BFS 遍历的基础上区分遍历的每一层，就得到了层序遍历。在层序遍历的基础上记录层数，就得到了最短路径。

BFS 遍历是一类很值得反复体会和练习的题目。一方面，BFS 遍历是一个经典的基础算法，需要重点掌握。另一方面，我们需要能根据题意分析出题目是要求最短路径，知道是要做 BFS 遍历。

本文讲解的只是两道非常典型的例题。LeetCode 中还有许多层序遍历和最短路径的题目

层序遍历的一些变种题目：

- **LeetCode 103. Binary Tree Zigzag Level Order Traversal** 之字形层序遍历
- **LeetCode 199. Binary Tree Right Side View** 找每一层的最右结点
- **LeetCode 515. Find Largest Value in Each Tree Row** 计算每一层的最大值
- **LeetCode 637. Average of Levels in Binary Tree** 计算每一层的平均值

对于最短路径问题，还有两道题目也是求网格结构中的最短路径，和我们讲解的距离岛屿的最远距离非常类似：

- **LeetCode 542. 01 Matrix**
- **LeetCode 994. Rotting Oranges**

还有一道在真正的图结构中求最短路径的问题：

- **LeetCode 310. Minimum Height Trees**

回溯

本期例题：**LeetCode 46 - Permutations**[1]（Medium）

> 给定一个**不重复**的数字集合，返回其所有可能的全排列。例如：
>
> - **输入：** `[1, 2, 3]`
> - **输出：**
>
> ```
> [
>   [1,2,3],
>   [1,3,2],
>   [2,1,3],
>   [2,3,1],
>   [3,1,2],
>   [3,2,1]
> ]
> ```

回溯法问题用递归求解，可以联系上树的遍历，我们可以将决策路径画成一棵树，回溯的过程就是这棵树的遍历过程。

非常典型且基础的回溯法问题 就是 子集（subset）问题。在面试中，我们需要有能力更加复杂的回溯法问题，并应对题目的各种变种。本篇以经典的排列（permutation）和组合（combination）问题为例，讲讲求解回溯法问题的要点：**候选集合**。

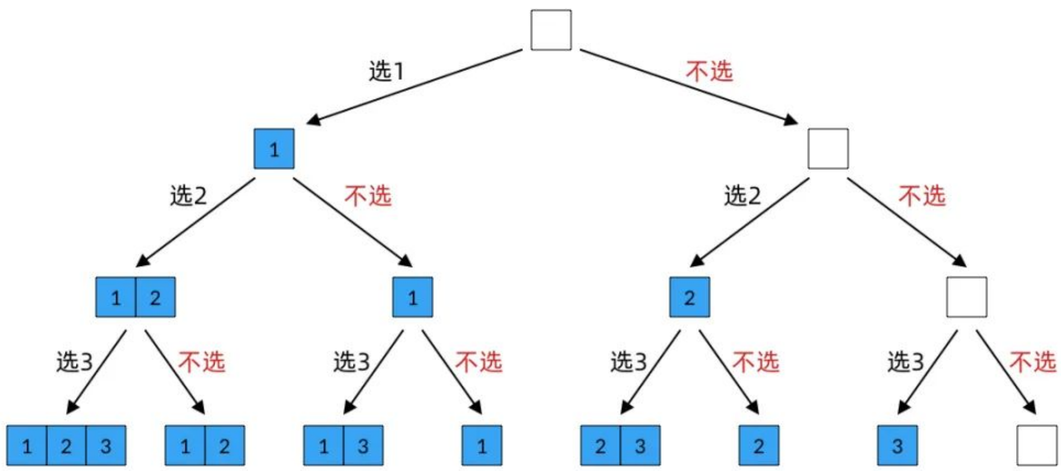
这篇文章将会包含：

- 回溯法的”选什么“问题与候选集合
- 全排列、排列、组合问题的回溯法解法

- 回溯法问题中，如何维护候选集合
- 回溯法问题中，如何处理失效元素

回溯法的重点：“选什么”

我们说过，回溯法实际上就是在一棵决策树上做遍历的过程。那么，求解回溯法题目时，我们首先要思考所有决策路径的形状。例如，子集问题的决策树如下图所示：



子集问题的决策树

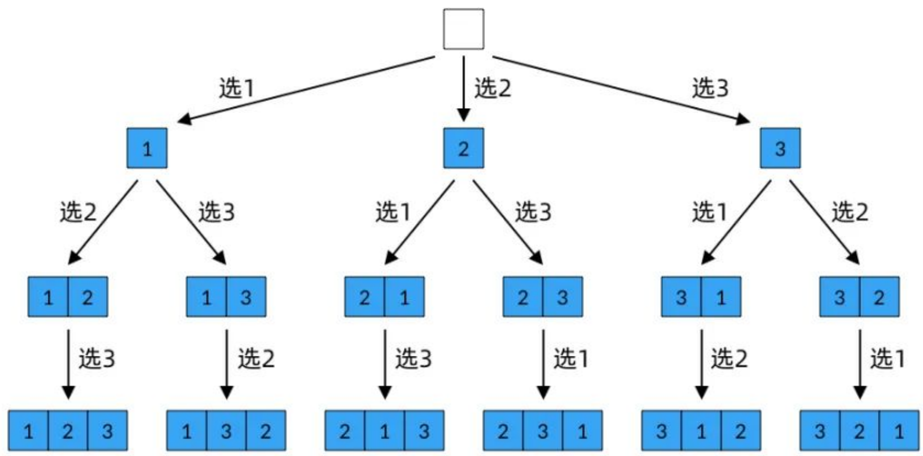
决策树形状主要取决于每个结点处可能的分支，换句话说，就是在每次做决策时，我们 **“可以选什么”**、**“有什么可选的”**。

对于子集问题而言，这个“选什么”的问题非常简单，每次只有一个元素可选，要么选、要么不选。不过，对于更多的回溯法题目，“选什么”的问题并不好回答。这时候，我们就需要分析问题的**候选集合**，以及候选集合的变化，以此得到解题的思路。

全排列问题：如何维护候选集合

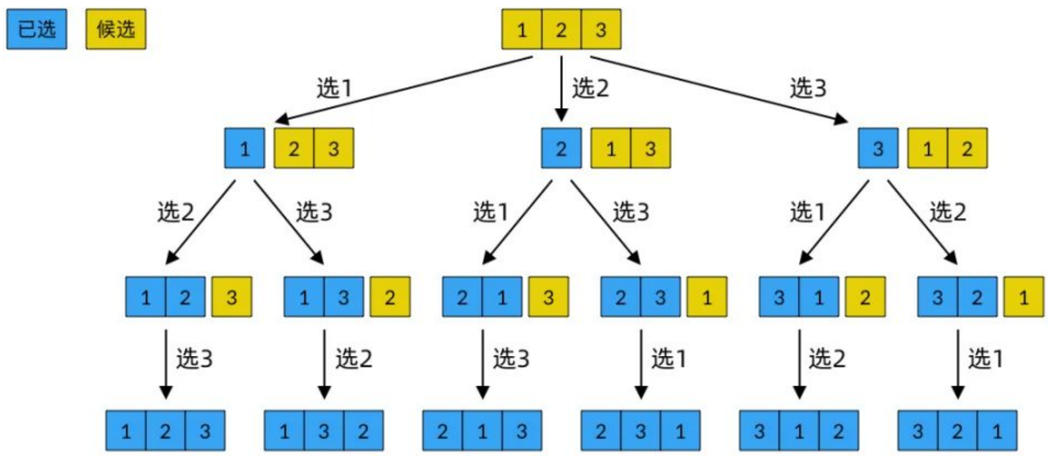
让我们拿经典的全排列问题来讲解回溯法问题的**候选集合**概念。

在全排列问题中，决策树的分支数量并不固定。我们一共做n次决策，第i次决策会选择排列的第i个数。选择第一个数时，全部的n个数都可供挑选。而由于已选的数不可以重复选择，越往后可供选择的数越少。以n=3为例，决策树的形状如下图所示：



全排列问题的决策树

如果从**候选集合**的角度来思考，在进行第一次选择时，全部的 3 个数都可以选择，候选集合的大小为 3。在第二次选择时，候选集合的大小就只有 2 了；第三次选择时，候选集合只剩一个元素。可以看到，全排列问题候选集合的变化规律是：每做一次选择，候选集合就少一个元素，直到候选集合选完为止。我们可以在上面的决策树的每个结点旁画上候选集合的元素，这样看得更清晰。

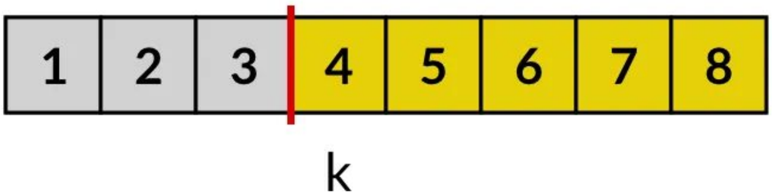


全排列问题有候选集合的决策树

可以看到，**已选集合**与**候选集合**是补集的关系，它们加起来就是全部的元素。而在回溯法的选择与撤销选择的过程中，已选集合和候选集合是此消彼长的关系。

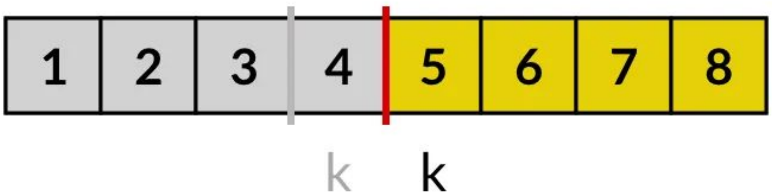
在一般情况下，**候选集合**使用**数组**表示即可。 候选集合上需要做的操作并不是很多，使用数组简单又高效。

在子集问题中，我们定义了变量 `k`，表示当前要对第 `k` 个元素做决策。实际上，变量 `k` 就是候选集合的边界，指针 `k` 之后的元素都是候选元素，而 `k` 之前都是无效元素，不可以再选了。



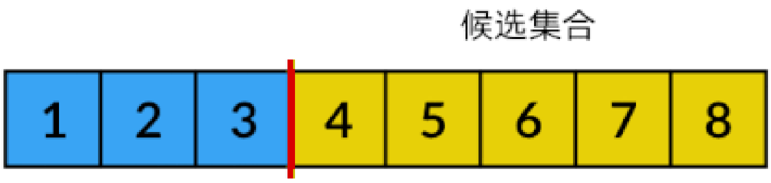
用数组表示候选集合

而每次决策完之后将 `k` 加一，就是将第 k 个元素移出了候选集合。



将第 k 个元素移出候选集合

在全排列问题中，我们要处理的情况更难一些。每次做决策时，候选集合中的所有元素都可以选择，也就是有可能删除候选集合中间的元素，这样数组中会出现“空洞”。这种情况该怎么处理呢？我们可以使用一个巧妙的方法，先将要删除的元素与第 k 个元素交换，再将 `k` 加一，过程如下图所示：



从候选集合中部删除元素

不知道你有没有注意到，上图中候选集合之外的元素画成了蓝色，这些实际上就是已选集合。前面分析过，已选集合与候选集合是互补的。将蓝色部分看成已选集合的话，我们从候选集合中删除的元素，正好加入了已选集合中。也就是说，我们可以只用一个数组同时表示已选集合和候选集合！

理解了图中的关系之后，题解代码就呼之欲出了。我们只需使用一个 `current` 数组，左半边表示已选元素，右半边表示候选元素。指针 `k` 不仅是候选元素的开始位置，还是已选元素的结束位置。我们可以得到一份非常简洁的题解代码：

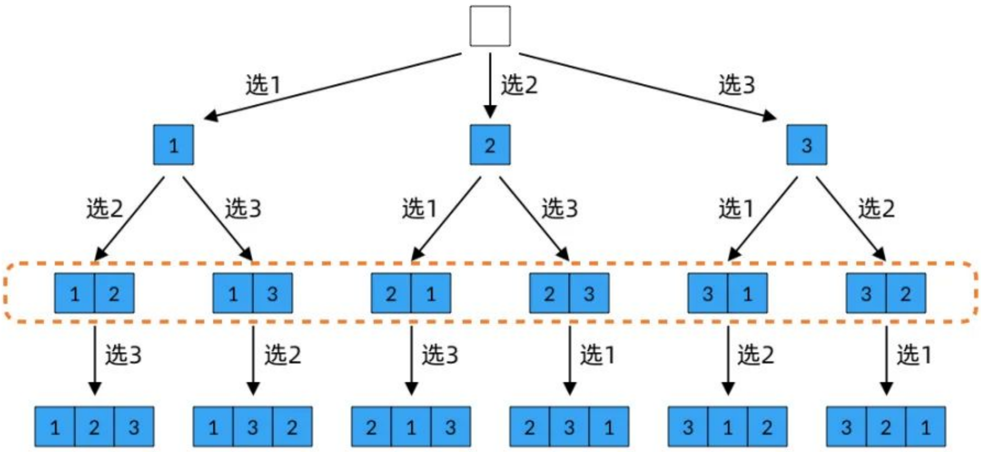
```
class Solution{
public:
    vector<vector<int>> permute(vector<int> &nums) {
        vector<int> current = nums;
        vector<vector<int>> res;
        backtrack(current, 0, res);
        return res;
    }

    // current[0..k) 是已选集合， current[k..N) 是候选集合
    void backtrack(vector<int> &current, int k, vector<vector<int>> res) {
        if (k == current.size()) {
            res.push_back(current);
            return;
        }
        // 从候选集合中选择
        for (int i = k; i < current.size(); ++i) {
            // 选择数字 current[i]
            std::swap(current[k], current[i]);
            // 注意是 k + 1
            backtrack(current, k+1, res);
            // 撤销选择
            std::swap(current[k], current[i]);
        }
    }
};
```

在写回溯法问题的代码时，你需要时刻清楚什么是已选集合，什么是候选集合。注释中的条件叫做“不变式”。一方面，我们在函数中可以参考变量 k 的含义，另一方面，我们在做递归调用的时候，要保证这个条件始终成立。特别注意代码中递归调用传入的参数是 `k+1` ，即删除一个候选元素。

n 中取 k 的排列

在面试中，我们很可能会遇到各种各样的 排列 、组合 的变种题，我们也要掌握。

`P(n,k)` 问题非常简单，我们只需要在全排列的基础上，做完第 k 个决策后就将结果返回。也就是说，只遍历决策树的前 k 层。例如n=3,k=2，决策树的第 2 层，已选集合中有两个元素，将这里的结果返回即可。



n 中取 k 的排列的决策树

题解代码如下所示，只需要修改递归结束的条件即可。

```
class Solution{
public:
    vector<vector<int>> permute(vector<int> &nums, int k) {
        vector<int> &current = nums;
        vector<vector<int>> res ;
        backtrack(k, current, 0, res);
        return res;
    }

    // current[0..m) 是已选集合， current[m..N) 是候选集合
    void backtrack(int k, vector<int> &current, int m, vector<vector<int>> res) {

        // 当已选集合达到 k 个元素时，收集结果并停止选择
        if (m == k) {
            // [ )
            res.insert(res.begin(), current.begin(); current.begin() + k);
            return;
        }
        // 从候选集合中选择
        for (int i = m; i < current.size(); i++) {
            // 选择数字 current[i]
            std::swap(current[m], current[i]);
            // m + 1
            backtrack(k, current, m + 1, res);
            // 撤销选择
            std::swap(current[m], current[i]);
        }
    }
};
```

注意这里 是题目的输入，所以原先我们代码里的变量 `k` 重命名成了 `m`。此外，就是递归函数开头的 if 语句条件发生了变化，当已选集合达到 个元素时，就收集结果停止递归。

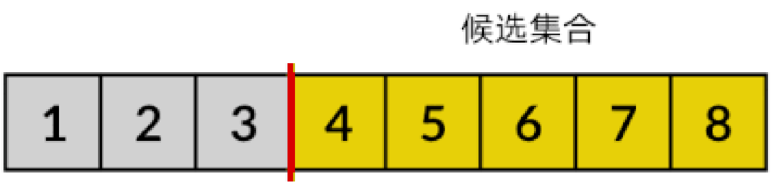
组合问题：失效元素

由于排列组合的密切联系，组合问题`C(n,k)`，即n中取k的组合，可以在`P(n,k)`问题的解法上稍加修改而来。

我们先思考一下组合和排列的关系。元素相同，但顺序不同的两个结果视为不同的排列，例如[1,2,3]和[2,1,3] 。但顺序不同的结果会视为同一组合。那么，我们只需要考虑 中所有**升序**的结果，就自然完成了组合的去重，得到`C(n,k)`。

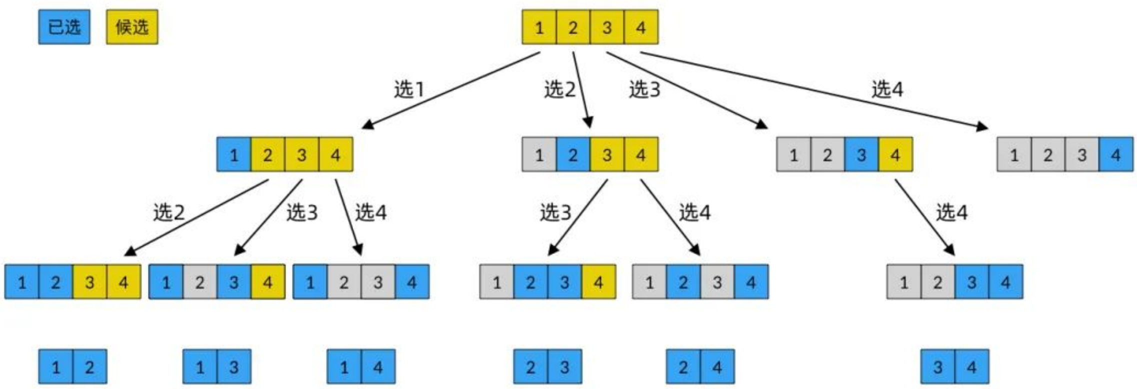
那么，如何让回溯只生成升序的排列呢？这需要稍微动点脑筋，但也不是很难，只需要做到：**每当选择了一个数`x`时，将候选集合中的所有小于`x`的元素删除，不再作为候选元素。**

再仔细想想的话，在排列问题为了维护候选集合而进行的交换操作，这里也不需要了。例如下面的例子，选择元素 6 之后，为了保持结果升序，前面的元素 4、5 也不能要了。不过，我们并不需要关注失效元素，我们只需要关注候选集合的变化情况。我们发现，剩下的候选集合仍然是数组中连续的一段，不会出现排列问题中的“空洞”情况。我们只用一个指针就能表示新的候选集合。



从候选集从候选集合中删除多个元素

下图是决策树，可以看到，候选集合都是连续的。已选集合不连续没有关系，我们可以另开一个数组保存已选元素。



组合问题的决策树

按照这个思路，我们可以写出代码。

```
class Solution{
public:
    vector<vector<int>> combine(vector<int> &nums, int k) {
        deque<int> current;
        vector<vector<int>> res;
        backtrack(k, nums, 0, current, res);
        return res;
```

```cpp
    }

    // current 是已选集合， nums[m..N) 是候选集合
    void backtrack(int k, vector<int> &nums, int m, deque<int> current, vector<vector<int>> res) {
        // 当已选集合达到 k 个元素时，收集结果并停止选择
        if (current.size() == k) {
            res.push_back(current);
            return;
        }
        // 从候选集合中选择
        for (int i = m; i < nums.size(); i++) {
            // 选择数字 nums[i]
            current.push_back(nums[i]);
            // 元素 nums[m..i) 均失效
            backtrack(k, nums, i+1, current, res);
            // 撤销选择
            current.pop_back();
        }
    }

};
```

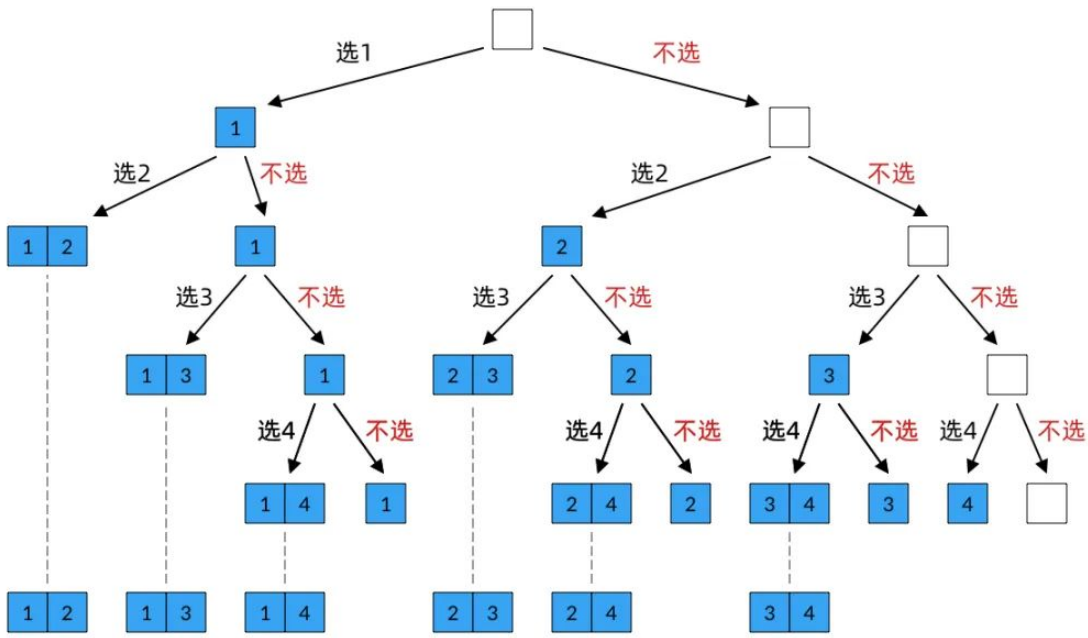
由于已选集合与候选集合并非互补，这里用单独的数组存储已选元素，这一点上与子集问题类似。

组合问题与子集问题的关系

也许是排列 & 组合的 CP 感太重，所以我们在思考组合问题的解法的时候会自然地从排列问题上迁移。其实，组合问题和子集问题有很密切的联系。

由子集问题求解组合问题

组合问题可以看成是子集问题的特殊情况。从n中取k个数的组合，实际上就是求n个元素的所有大小为 k 的子集。也就是说，组合问题的结果是子集问题的一部分。我们可以在子集问题的决策树的基础上，当已选集合大小为k的时候就不再递归，就可以得到组合问题的决策树。



在子集问题决策树基础上得到的组合问题决策树

由组合问题求解子集问题

对于子集问题，大小为n的集合共有2^n 个可能的子集。要得到全部的2^n 个子集，我们可以计算所有n中取1,2,3,...,n 的组合，再把这些组合加起来。根据这个思路，我们可以在组合问题的题解代码上稍加修改得到子集问题的解：

```
class Solution{
public:
    vector<vector<int>> subsets(vector<int> &nums) {
        deque<int> current;
        vector<vector<int>> res;
        backtrack(nums, 0, current, res);
        return res;
    }

    // current 是已选集合， nums[m..N) 是候选集合
    void backtrack(vector<int> &nums, int m, deque<int> current, vector<vector<int>> res) {
        // 收集决策树上每一个结点的结果
        res.push_back(current);
        if (m == nums.size()) {
            // 当候选集合为空时，停止递归
            return;
        }
        // 从候选集合中选择
        for (int i = m; i < nums.size(); i++) {
            // 选择数字 nums[i]
            current.push_back(nums[i]);
            // 元素 nums[m..i) 均失效
            backtrack(nums, i+1, current, res);
            // 撤销选择
            current.pop_back();
        }
    }

};
```

可以看到，每次做决策都会增加一个已选元素。当递归到第k层时，计算的就是大小为`k`的子集。不过，这样写出的子集问题解法没有原解法易懂。

总结

排列组合问题是回溯法中非常实际也非常典型的例题，可以通过做这些题目来体会回溯法的基本技巧。不过它们在 LeetCode 中没有完全对应的例题。文章开头的例题是全排列问题。对于组合问题，LeetCode 只有一个简化版 **77. Combinations**[2]，其中数字固定为 1 到 n 的整数。

排列组合问题展示了在求解回溯法问题时，**候选集合**的概念对理清思路的重要性。实际上，回溯法中的"选择"与"撤销选择"，实际上就是从候选集合中删除元素与添加回元素的操作。而我们在写代码的时候要注意在递归函数上方写注释，明确数组的哪一部分是候选集合。

排列组合问题还存在着一些变种，例如当输入存在重复元素的时候，如何避免结果重复，就需要使用决策树的剪枝方法。

动规

今天要讲的是「**如何定义多个子问题**」。

常规的动态规划问题只需要定义一个子问题即可。然而在某些情况下，把子问题拆成多个会让思路更清晰。如果你没用过这个技巧的话，不妨跟着下面的例题来学习学习。

本篇文章的内容包括：

- 如何拆分动态规划的子问题
- 「最长波形子数组」问题的解法
- 度假问题的解法
- 多个子问题与二维子问题的转换关系

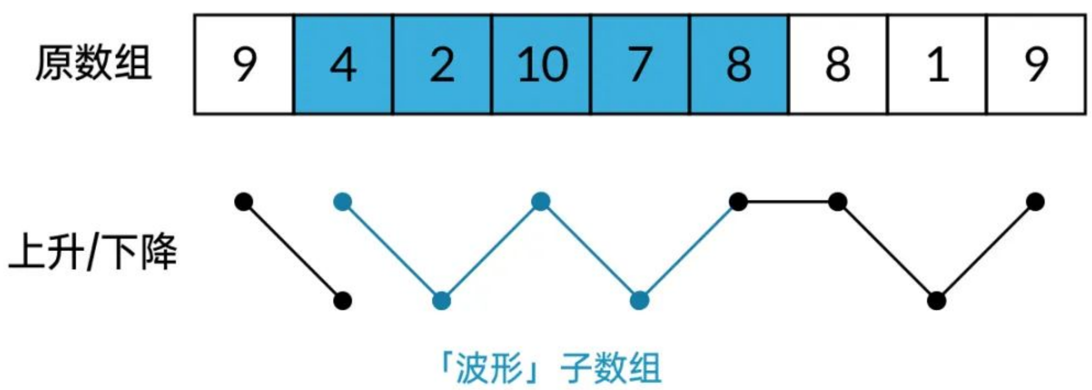
最长波形子数组

我们用「最长波形子数组」的解题过程来展示**定义多个子问题**在解题中的作用。

LeetCode 978. Longest Turbulent Subarray 最长波形子数组（Medium）

> 当 `A` 的子数组 `A[i..j]` 满足下列条件之一时，我们称其为**波形子数组**：
>
> 对于 i <= k < j，当 k 为奇数时， `A[k] > A[k+1]`，当 k 为偶数时，`A[k] < A[k+1]`；
>
> 或者：对于 i <= k < j，当 k 为偶数时，`A[k] > A[k+1]` ，当 k 为奇数时， `A[k] < A[k+1]`。
>
> 也就是说，如果比较符号在子数组中的每个相邻元素对之间翻转，则该子数组是波形子数组。
>
> 返回 A 的最长波形子数组的长度。

首先我们要明白「波形子数组」的含义。（吐槽一句，官方把 trubulent 翻译成「湍流」，这翻译是给人看的吗？）我们关注的是数组中相邻元素之间的**大小关系**。如果后一个元素大于前一个元素，则是数组的「上升段」；反之，则是数组的「下降段」。那么，「波形子数组」就是一段交替上升下降的子数组。例如输入 `[9, 4, 2, 10, 7, 8, 8, 1, 9]` 中， `[4, 2, 10, 7, 8]` 是其中最长的一段波形子数组。



「波形子数组」是一段交替上升下降的子数组

使用单个子问题求解

我们先看看使用传统的单个子问题该怎么求解这道题。

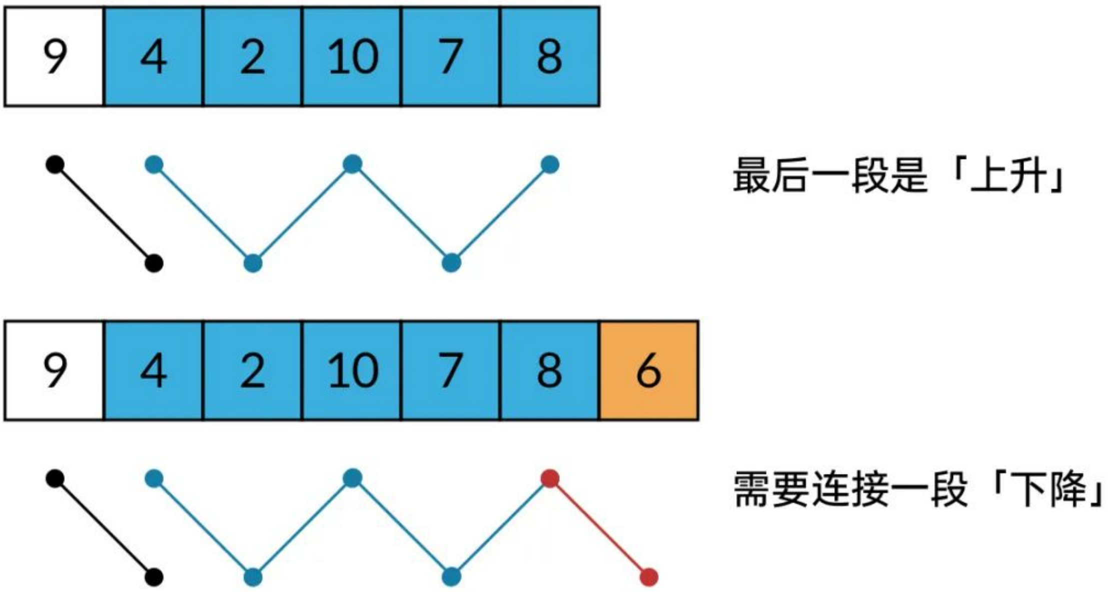
首先，看到题目中的「子数组」字样，我们应当立即想到子数组相关的解题技巧：在定义子问题的时候给子问题加上**位于数组尾部**的限制。

我们可以这样定义子问题：

子问题 表示「数组 `A[0..k)` 中，**位于数组尾部**的最长波形子数组」。

之所以要限制子问题中求的最长波形子数组位于数组尾部，是因为只有数组尾部的波形子数组才可以和新加入的上升/下降段连接起来。

需要注意的是，波形数组的连接是有条件的，需要「上升段」和「下降段」交替出现。如果波形数组的最后一段是「上升」，就需要连接一段「下降」才是合法的波形数组；而如果波形数组的最后一段是「下降」，就需要连接一段「上升」才是合法的波形数组。

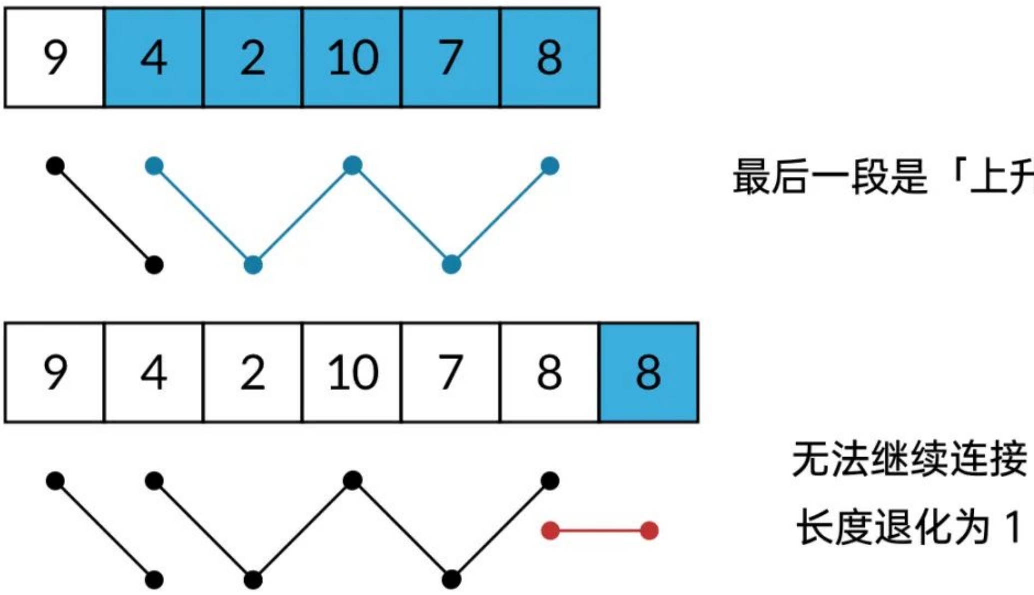


如果波形子数组的最后一段是「上升」，就需要连接一段「下降」

而如果「上升」之后又是一段「上升」，那么整个波形数组不合法。波形子数组的长度减少到 2（包含最后一个上升段的两个元素）。

连续两个「上升」段，无法继续连接，长度退化为 2

当然，如果最后一段既不是上升，也不是下降，而是「水平」段，那这最后一段也是不合法的。波形子数组的长度减少到 1。



新加入的是水平段，无法继续连接，长度退化为 1

那么，我们在写子问题的递推关系时，需要分类讨论。对于子问题：

- 如果`f(k-1)`波形数组的最后一段是「上升」，且 `A[k-1]` 和 `A[k-2]` 之间是「上升」，那么`f(k) = 2`；

- 如果 `f(k-1)` 波形数组的最后一段是「上升」，且 `A[k-1]` 和 `A[k-2]` 之间是「下降」，那么 `f(k) = f(k - 1) + 1`；
- 如果 `f(k-1)` 波形数组的最后一段是「下降」，且 `A[k-1]` 和 `A[k-2]` 之间是「上升」，那么 `f(k) = f(k - 1) + 1`；
- 如果 `f(k-1)` 波形数组的最后一段是「下降」，且 `A[k-1]` 和 `A[k-2]` 之间是「下降」，那么 `f(k) = 2`；
- 如果 `A[k-1]` 和 `A[k-2]` 之间是「水平」，那么 `f(k) = 1`。

什么？一个看似简单的问题竟然要分这么多情况考虑，是不是看得头都大了？

通常来说，如果你发现子问题的递推关系过于复杂，那可能是子问题定义得不是很好。关键的思路来了：**如果对子问题进行拆分，可以减少很多不必要的分类讨论。**

下面，我们尝试拆分子问题，使用多个子问题进行求解。

使用多个子问题求解

既然我们总是要判断波形数组的最后一段是上升还是下降，那我们为何不在子问题定义时就把它们区分开来呢？

我们可以定义**两个**子问题，分别对应最后一段上升和下降的波形子数组：

- 子问题 `f(k)` 表示：数组 `A[0..k)` 中，位于数组尾部，且 **最后一段为「上升」** 的最长波形子数组；
- 子问题 `g(k)` 表示：数组 `A[0..k)` 中，位于数组尾部，且 **最后一段为「下降」** 的最长波形子数组。

这样一来，我们的子问题递推关系也变得清晰了起来：

- 如果 `A[k-1]` 和 `A[k-2]` 之间是「上升」，那么 `f(k) = g(k - 1) + 1`，`g(k) = 1`；
- 如果 `A[k-1]` 和 `A[k-2]` 之间是「下降」，那么 `f(k) = 1`，`g(k) = f(k - 1) + 1`；
- 如果 `A[k-1]` 和 `A[k-2]` 之间是「水平」，那么 `f(k) = 1`，`g(k) = 1`。

这样，我们就可以写出题解代码了。需要注意的是，既然我们定义了多个子问题，就需要在代码中定义多个 DP 数组。我们直接把 DP 数组命名为 `f` 和 `g`，与子问题对应：

```
class Solution{
public:
    int maxTurbulenceSize(vector<int> A) {
        if (A.size() <= 1) {
            return A.size();
        }

        int N = A.size();
        // 定义两个 DP 数组 f, g
        int f[N+1] = {0};
        int g[N+1] = {0};
        f[1] = 1;
        g[1] = 1;

        int res = 1;
        for (int k = 2; k <= N; k++) {
            // 如果 A[k-1] 和 A[k-2] 之间是「上升」的
            if (A[k-2] < A[k-1]) {
                f[k] = g[k-1] + 1;
                g[k] = 1;
            } else if (A[k-2] > A[k-1]) {
```

```
                // 如果 A[k-1] 和 A[k-2] 之间是「下降」的
                f[k] = 1;
                g[k] = f[k-1] + 1;
            } else {
                // 如果 A[k-1] 和 A[k-2] 之间是「水平」的
                f[k] = 1;
                g[k] = 1;
            }
            res = max(res, f[k]);
            res = max(res, g[k]);
        }
        return res;
    }
};
```

多个子问题的本质

让我们从 DP 数组的角度来理解动态规划中「定义多个子问题」究竟意味着什么。

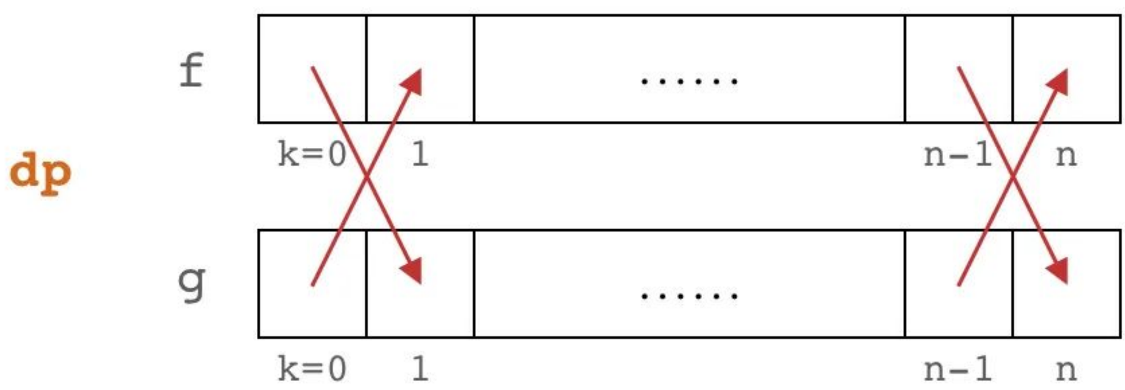
请思考一个问题：在「最长波形子数组」问题中，DP 数组是一维的还是二维的?

从子问题的定义来看的话，子问题只有一个参数k，看起来应该是一维的。不过和普通的一维动态规划问题的不同之处在于，因为有两个子问题 `f(k)` 和 `g(k)`，所以 DP 数组有两个，其中每个是一维的。

我们可以画出 DP 数组的形状来直观地理解。设数组的长度为n，则k的取值范围是 `[0,n]`。DP 数组是两个长度为n+1的数组，如下图所示。

将 DP 数组看成两个一维的数组

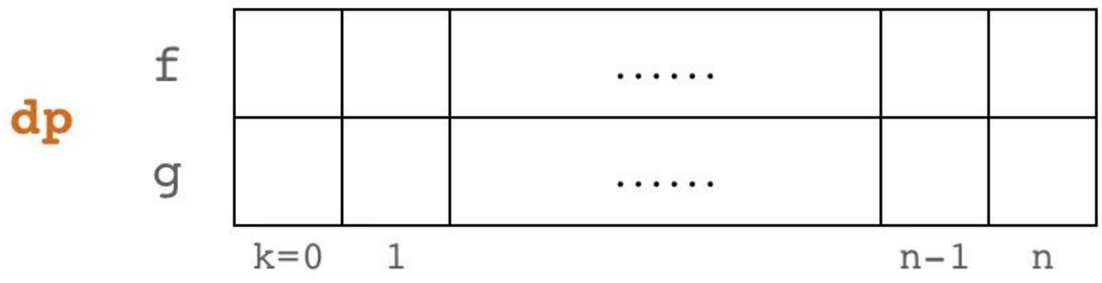
接下来，我们在 DP 数组中画出子问题的依赖关系。 `f(k)` 只依赖于 `g(k - 1 )` ， `g(k)` 只依赖于 `f(k - 1)`，那么可以画出子问题的依赖关系为：



DP 数组中子问题的依赖关系

可以看出，两个子问题互相依赖，整体的依赖顺序是从左往右的。

另一方面，我们也可以把 DP 数组看成二维数组。把两个长度为n+1的数组拼在一起，就得到了一个`2 x (n + 1)`的二维数组。

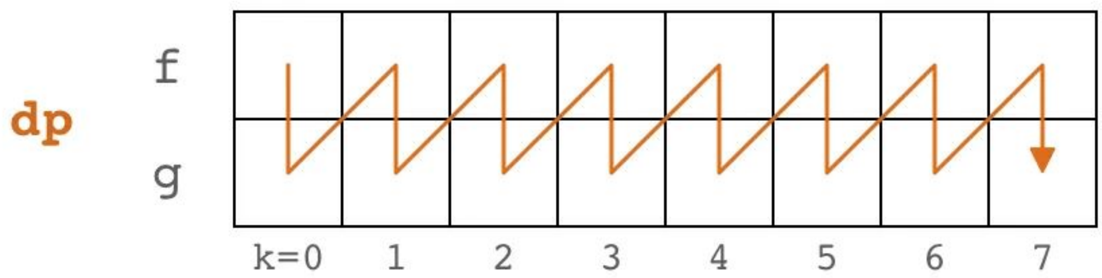


将 DP 数组看成二维数组

但是，这样的一个 DP 数组和常规的二维动态规划中的 DP 数组不太一样：

第一，DP 数组其中一维的长度为 2，是个常数。计算空间复杂度的话，这个二维 DP 数组的空间复杂度是`O(2n) = O(n)`，仍然是一维数组的复杂度级别。

第二，一般的二维动态规划问题（如最长公共子序列、编辑距离这些经典题目），DP 数组的计算顺序既可以是从上往下，也可以是从左往右。而这个 DP 数组根据依赖顺序，计算顺序只能是从左往右，不能先计算第一行（`f`）再计算第二行（`g`）。



DP 数组的计算顺序

综上，我们可以看出，有多个子问题的动态规划，其维度实际上介于一维和二维之间。本题只定义了两个（常数个）子问题，而如果子问题的数量扩展到了`m`个，DP 数组的空间复杂度就到达了`O(m*n)`，变成了一个真正的二维动态规划问题。

另一道例题：度假问题

让我们再看一道典型的拆分子问题的动态规划题目，来理解定义多个子问题的技巧。这道题不是来自 LeetCode，而是来自另一个算法网站 AtCoder：**AtCoder DP-C. Vacation**

> 题目链接：https://atcoder.jp/contests/dp/tasks/dp_c
>
> Taro 的暑假明天开始，他决定现在制定好暑假的计划。
>
> 假期共持续N天。Taro 可以选择在第i天（`1 ≤ i ≤ N`）做以下三件事之一：
>
> - A：游泳。获得a_i点快乐指数。
> - B：捉虫。获得b_i点快乐指数。
> - C：写作业。获得c_i点快乐指数。
>
> 由于 Taro 做一件事情很容易无聊，所以他不能连续两天做同一件事情。
>
> 输入包括N以及数组 a、b、c 的内容。

> 请计算 Taro 总共能获得的最大快乐指数。

这道题目该怎么拆分子问题呢？我们注意到一个关键的题目条件：**Taro 不能连续两天做同一件事情**。也就是说：

- 如果 Taro 今天做的是事情 A，那么他明天可以做事情 B 和 C；
- 如果 Taro 今天做的是事情 B，那么他明天可以做事情 A 和 C；
- 如果 Taro 今天做的是事情 C，那么他明天可以做事情 A 和 B。

这样的话，我们可以根据 Taro 今天做的是哪件事，定义出三个子问题：

- 子问题 `f1(k)` 表示 Taro 在第 k 天做事情 A 的情况下，前 k 天能获得的最大快乐指数；
- 子问题 `f2(k)` 表示 Taro 在第 k 天做事情 B 的情况下，前 k 天能获得的最大快乐指数；
- 子问题 `f3(k)` 表示 Taro 在第 k 天做事情 C 的情况下，前 k 天能获得的最大快乐指数。

然后我们可以写出子问题间的递推关系：

`f1(k) = max{f2(k-1), f3(k-1)} + ak`

`f2(k) = max{f1(k-1), f3(k-1)} + bk`

`f3(k) = max{f1(k-1), f3(k-1)} + ck`

递推关系为什么是这样的呢？以 `f1(k)` 的公式为例：

`f1(k)` 表示 Taro 在第 k 天做事情 A 的情况下，前 k 天能获得的最大快乐指数。既然 Taro 在第 k 天做了事情 A，那么他在第 k-1 天就不能做事情 A，只能做事情 B 或 C，对应 `f2(k-1)` 和 `f3(k-1)`。也就是说，`f1(k)` 是根据 `f2(k-1)` 和 `f3(k-1)` 求出来的。

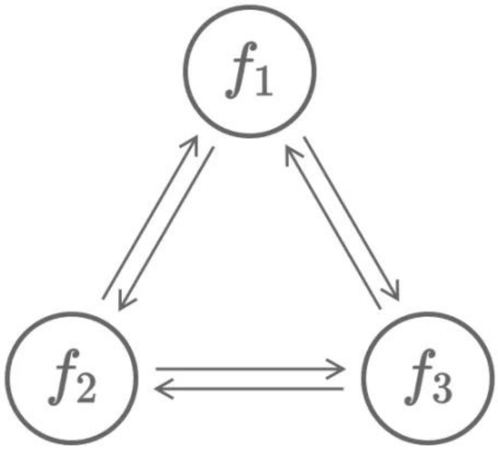
`f2(k)` 和 `f3(k)` 的公式同理可得。

有了这个递推关系，我们就可以写出题解代码：

```
class solution{
public:
    int vacation(vector<int> a, vector<int> b, vector<int> c) {
        int n = a.size();
        int f1[n+1] = {0};
        int f2[n+1] = {0};
        int f3[n+1] = {0};

        for (int k = 1; k <= n; k++) {
            f1[k] = a[k-1] + max(f2[k-1], f3[k-1]);
            f2[k] = b[k-1] + max(f1[k-1], f3[k-1]);
            f3[k] = c[k-1] + max(f1[k-1], f2[k-1]);
        }
        return max(f1[n], max(f2[n], f3[n]));
    }
};
```

可以看到，题解代码还是非常简洁的。在代码中，`f1`、`f2` 和 `f3` 呈现出一种相互依赖、交替计算的关系。

我们可以用这样一张图来描述这三个子问题之间的关系：



三个子问题之间的关系

图中的箭头表示子问题间的**依赖关系**。例如`f1`到`f2`有一条边，表示`f2(k)`依赖于`f1(k-1)`。而`f1(k)`不依赖于`f1(k-1)`，所以`f1`没有到自己的边。

眼尖的同学可能已经看出，这张图实际上展示的是一个状态机。状态机中有`f1`、`f2`、`f3`三种状态。如果状态机在第 `k-1` 天位于状态`f1`，那么第 `k` 天的状态无法维持在`f1`，只能跳到`f2` 或`f3` 。这对应了「Taro 不能连续两天做同一件事情」的题目条件。

实际上，「状态机」是动态规划中的一种技巧，大名鼎鼎的股票买卖问题就是属于「状态机 DP」。感兴趣的同学可以了解下 股票问题和状态机 DP。

总结

本文用两道例题展示了动态规划问题中拆解子问题、定义多个子问题的技巧。两道题目虽然分别定义了 2 个、3 个子问题，但是子问题的拆分方式和计算顺序都是非常相似的。把两道题目放在一起对比的话，可以很快理解动态规划定义多个子问题的套路。