# 常用方法

## Map

```
Map<String, Integer> counts = new HashMap();
// 如果不存在，则返回0
counts.getOrDefault(cur, 0);
```

## String

```
String domain = "900 google.mail.com";
// 分隔符 切分
String[] cpinfo = domain.split("\\s+");
// 特殊符号远点 切分
String[] frags = cpinfo[1].split("\\.");

String s = "aaabb";
int k = 3;
StringBuilder sb = new StringBuilder(s);
// 遍历字符串的每一个字符，并在遍历的时候操作
for(int i=0; i<s.length(); i++) {
    char c = s.charAt();
    sb.delete(i - k + 1, i + 1);
    i = i - k;
}

//子串
int begin = 0;
int end = s.length;
s.substring(begin, end);
s.substring(begin);
```

## Array

```
// 初始化 数组
int[] keyValue = {1, 2};
// 初始化 数组 int默认0,boolean默认false
int[] array = new int[5];
boolean[] booleanArray = new boolean[3]
int[] people;
// 排序 默认数组元素 进行升序排序
Arrays.sort(people);
```

## List

```
List<int[]> list = new ArrayList<>();
// 根据集合元素 进行 升序排序 （1,1,2,3）
Collections.sort(list, Comparator.comparingInt(item -> item[1]));

LinkedList<Integer> track = new LinkedList<>();
// 移除最后一个
track.removeLast();
// 数组转List,List初始化HashSet
Set<String> good = new HashSet<String>(Arrays.asList(words));
```

## Set

```
Set<String> set = new HashSet();
// 取集合第一个元素
set.iterator().next().length()
```

## int

```
int ret, maxlen;
// 两者取最大的
ret = Math.max(ret, maxlen);
```

## Stack

```
// 栈 压栈出栈操作
Stack<Integer> counts = new Stack<>();
counts.push(1);
counts.pop();

// 单调栈初始化
Deque<Integer> stack = new LinkedList<Integer>();
// 获取栈顶元素，也就是最后压进的一个,不同于pop(),仅查看不出栈
stack.peek();
```

# 其他

```
// 整形取值，最好用Long，因为Integer会有范围，部分测试用例可能不能通过
Long.MIN_VALUE;
Long.MAX_VALUE;
```

# 常用英文

```
// 遍历
traverse();
// 行数
int rows;
// 列数
int columns;
// 大小
long max,min;
long upper,lower;
```

# 常见问题

- 题目是给开发工程师看的，对题目文字要敏感，比如包含于 表示 主语一般是主集合
- 遍历的时候，容易弄错left,right
- 递归的时候，容易copy带来低级错误
- **数组下标，不易理解时，先按实际的下标设值，取值是-1即可**
- 字符类算法题，一般希望使用char[] 字符数组解决，而不是使用String的一系列函数
- 数组从0开始，所以一般计算时，比较多的操作是**顺移**，length-1再操作

# 递归

- 既然打算使用递归，一定是把结束条件放在开始的地方

# 动态规划

非常重要，找到**转移方程**

# 预先练习题型集合

- 最长回文子串

```
输入：s = "babad"
输出："bab"
解释："aba" 同样是符合题意的答案。
```

中心扩展动态规划

状态转移链：

边界情况（1个， 2个相同的）--> 中心扩展计算最长-->直至遍历完所有潜在，记录起始记录最后取到答案

```
public String longestPalindrome(String s) {
        if (s == null || s.length() < 1) {
```

```
            return "";
        }
        int start = 0, end = 0;
        for (int i = 0; i < s.length(); i++) {
            int len1 = expandAroundCenter(s, i, i);
            int len2 = expandAroundCenter(s, i, i + 1);
            int len = Math.max(len1, len2);
            if (len > end - start) {
                start = i - (len - 1) / 2;
                end = i + len / 2;
            }
        }
        return s.substring(start, end + 1);
    }
public int expandAroundCenter(String s, int left, int right) {
    while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right))
{
        --left;
        ++right;
    }
    return right - left - 1;
}
```

- 岛屿数量https://leetcode-cn.com/problems/number-of-islands/

类似的岛屿面积、周长（更合适的方式是转换一下），DFS遍历图也是一样的

DFS + 遍历结束条件的设定（**沉岛思想**），dfs更新相邻的1为0，这样一次dfs只累计一次，正好符合所要

数量

```
void dfs(char[][] grid, int r, int c) {
        int nr = grid.length;
        int nc = grid[0].length;

        if (r < 0 || c < 0 || r >= nr || c >= nc || grid[r][c] == '0') {
            return;
        }

        grid[r][c] = '0';
        dfs(grid, r - 1, c);
        dfs(grid, r + 1, c);
        dfs(grid, r, c - 1);
        dfs(grid, r, c + 1);
    }

    public int numIslands(char[][] grid) {
        if (grid == null || grid.length == 0) {
            return 0;
        }

        int nr = grid.length;
        int nc = grid[0].length;
        int num_islands = 0;
        for (int r = 0; r < nr; ++r) {
```

```
            for (int c = 0; c < nc; ++c) {
                if (grid[r][c] == '1') {
                    ++num_islands;
                    dfs(grid, r, c);
                }
            }
        }

        return num_islands;
    }
```

面积

```
public int maxAreaOfIsland(int[][] grid) {
        int res = 0;
        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[i].length; j++) {
                if (grid[i][j] == 1) {
                    res = Math.max(res, dfs(i, j, grid));
                }
            }
        }
        return res;
    }
    // 每次调用的时候默认num为1，进入后判断如果不是岛屿，则直接返回0，就可以避免预防错误的情况。
    // 每次找到岛屿，则直接把找到的岛屿改成0，这是传说中的沉岛思想，就是遇到岛屿就把他和周围的全部沉默。
    // ps：如果能用沉岛思想，那么自然可以用朋友圈思想。有兴趣的朋友可以去尝试。
    private int dfs(int i, int j, int[][] grid) {
        if (i < 0 || j < 0 || i >= grid.length || j >= grid[i].length || grid[i][j] == 0) {
            return 0;
        }
        grid[i][j] = 0;
        int num = 1;
        num += dfs(i + 1, j, grid);
        num += dfs(i - 1, j, grid);
        num += dfs(i, j + 1, grid);
        num += dfs(i, j - 1, grid);
        return num;
    }
```

周长，总周长 = 4 * 土地个数 - 2 * 接壤边的条数

```
public int islandPerimeter(int[][] grid) {
    int totolCount = 0;
    int sideCount = 0;
    int width = grid.length;
    int hight = grid[0].length;
    for(int x = 0; x < width; x++){
        for(int y =0; y < hight; y++){
            if(grid[x][y] == 1){
                totolCount ++;
                if(x < width -1){
                    if(grid[x + 1][y] == 1){
```

```
                        sideCount++;
                    }
                }
                if(y < hight -1){
                    if(grid[x][y+1] == 1){
                        sideCount++;
                    }
                }
            }
        }
    }
    return totolCount * 4 - sideCount * 2;
}
```

- 面积最大正方形 https://leetcode-cn.com/problems/maximal-square/

与统计全为 1 的正方形子矩阵相似

动态规划，转义方程很重要，后者利用遍历二维数组，把满足条件的已累计到格子中很妙

个数会统计之后，面积不就是++改为Math.max，然后平方吗?

**转移方程**关键

`matrix[i][j]` 表示以 `(i, j)` 为右下角的正方形的个数，其实也表示以 `(i, j)` 为右下角的最大正方形边长

```
matrix[i][j] = min(matrix[i - 1][j - 1],min(matrix[i - 1][j],matrix[i][j - 1]))
+ 1;
```

- 单词的压缩编码

  存储后缀

> 如果单词 `Y` 不在任何别的单词 `X` 的后缀中出现，那么 `Y` 一定是编码字符串的一部分

```
public int minimumLengthEncoding(String[] words) {
    Set<String> good = new HashSet<String>(Arrays.asList(words));
    for (String word: words) {
        for (int k = 1; k < word.length(); ++k) {
            good.remove(word.substring(k));
        }
    }

    int ans = 0;
    for (String word: good) {
        ans += word.length() + 1;
    }
    return ans;
}
```

- 每日温度

数组输入，数组输出，比对大小，下一次得到上一次的值，即适合用栈，否则就是大循环

```
public int[] dailyTemperatures(int[] T) {
    int length = T.length;
    int[] ans = new int[length];
    Deque<Integer> stack = new LinkedList<Integer>();
    for (int i = 0; i < length; i++) {
        int temperature = T[i];
        while (!stack.isEmpty() && temperature > T[stack.peek()]) {
            int prevIndex = stack.pop();
            ans[prevIndex] = i - prevIndex;
        }
        stack.push(i);
    }
    return ans;
}
```

- **跳跃游戏II** https://leetcode-cn.com/problems/jump-game-ii/

典型的贪心算法，通过局部最优解得到全局最优解

# 典型算法

DFS树，全排列，N皇后问题

```
List<List<Integer>> res = new LinkedList<>();

/* 主函数，输入一组不重复的数字，返回它们的全排列 */
List<List<Integer>> permute(int[] nums) {
    // 记录「路径」
    LinkedList<Integer> track = new LinkedList<>();
    backtrack(nums, track);
    return res;
}

// 路径：记录在 track 中
// 选择列表：nums 中不存在于 track 的那些元素
// 结束条件：nums 中的元素全都在 track 中出现
void backtrack(int[] nums, LinkedList<Integer> track) {
    // 触发结束条件
    if (track.size() == nums.length) {
        res.add(new LinkedList(track));
        return;
    }

    for (int i = 0; i < nums.length; i++) {
        // 排除不合法的选择
        if (track.contains(nums[i]))
            continue;
        // 做选择
        track.add(nums[i]);
```

```
        // 进入下一层决策树
        backtrack(nums, track);
        // 取消选择
        track.removeLast();
    }
}
```