刷题实录，进大厂，刷题是最基本的。刷题数量不在多，而在于对每道题刷过后，反思总结。

- 作者：鱼哥
- 版本：v1.0版本

  文案源自网络!

动态规划专题

动态规划（英语：Dynamic programming，简称 DP）是一种在数学、管理科学、计算机科学、经济学和生物信息学中使用的，通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。

动态规划常常适用于有重叠子问题和最优子结构性质的问题，动态规划方法所耗时间往往远少于朴素解法。

动态规划背后的基本思想非常简单。大致上，若要解一个给定问题，我们需要解其不同部分（即子问题），再根据子问题的解以得出原问题的解。动态规划往往用于优化递归问题，例如斐波那契数列，如果运用递归的方式来求解会重复计算很多相同的子问题，利用动态规划的思想可以减少计算量。

通常许多子问题非常相似，为此动态规划法试图仅仅解决每个子问题一次，具有天然剪枝的功能，从而减少计算量：一旦某个给定子问题的解已经算出，则将其记忆化存储，以便下次需要同一个子问题解之时直接查表。这种做法在重复子问题的数目关于输入的规模呈指数增长时特别有用。

1 、岛屿的最大面积

解题思路：

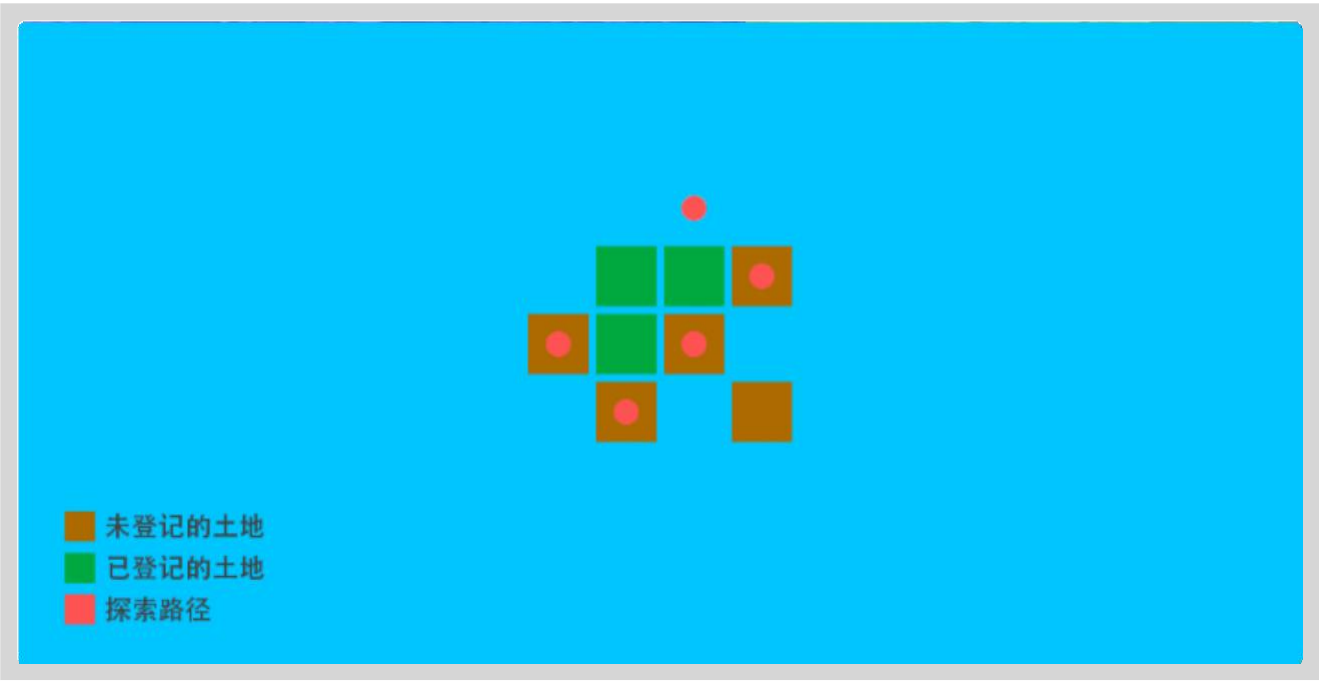
给定一个包含了一些 00 和11 的非空二维数组grid，一个岛屿是由四个方向(水平或垂直) 的11 (代表土地) 构成的组合。你可以假设二维矩阵的四个边缘都被水包围着。 由于每个岛屿皆被水包围，所以，仅需要确保每一次寻找到新岛屿时，所测量到的岛屿面积为该岛屿的最大面积，最后返回所测所有岛屿中的最大面积即可。

由于并不知道如何才可以测量出岛屿的面积，所以为了测量整个岛屿的面积，只能采取一步步探索的方式：

当登陆某个岛屿后，以此时所处位置为行动中心，随后分别向东、南、西、北四个方向前进。如果向某一方向前进后其为水或登记的地方则停止探索，而当步入新地点时，则继续以当前所处位置为行动中 心，

随后再一次向东、南、西、北四个方向前进，以此类推。

此方法过程如下：



最终代码如下：

```
public int maxAreaOfIsland(int[][] grid) {
        int ans = 0;
        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[i].length; j++) { ans =
                Math.max(ans, area(grid, i, j));
            }
        }

    private int area(int[][] grid, int i, int j) {
        if (i < 0 || j < 0 || i == grid.length || j == grid[i].length || grid[i][j] == 0) return 0;

        grid[i][j] = 0;

        return area(grid, i - 1, j)

        + area(grid, i + 1, j) + area(grid, i, j - 1) + area(grid, i, j + 1) + 1;
```

2、一和零

解题思路：

这道题和经典的背包问题很类似，不同的是在背包问题中，我们只有一种容量，而在这道题中，我们有 0 和1 两种容量。每个物品（字符串）需要分别占用 0 和1 的若干容量，并且所有物品的价值均为 1。因此我们可以使用二维的动态规划。

我们用dp(i, j) 表示使用i 个 0 和j 个 1，最多能拼出的字符串数目，那么状态转移方程为：

dp(i, j) = max(1 + dp(i − cost_zero[k], j − cost_one[k]))

if i >= cost_zero[k] and j >= cost_one[k]

其中k 表示第k 个字符串，cost_zero[k] 和cost_one[k] 表示该字符串中 0 和 1 的个数。我们依次枚举这些字符串，并根据状态转移方程更新所有的dp(i, j)。注意由于每个字符串只能使用一次（即有限背包），因此在更新dp(i, j) 时，i 和j 都需要从大到小进行枚举。

最终的答案即为所有dp(i, j) 中的最大值。

最终代码实现：

```
public class Solution {
    public int findMaxForm(String[] strs, int m, int n) { int[][] dp =
        new int[m + 1][n + 1];
        for (String s: strs) {
            int[] count = countzeroesones(s);
            for (int zeroes = m; zeroes >= count[0]; zeroes--) for (int
                ones = n; ones >= count[1]; ones--)
                    dp[zeroes][ones] = Math.max(1 + dp[zeroes - count[0]][ones - count[1]], dp[
        }
        return dp[m][n];
    }
    public int[] countzeroesones(String s) { int[] c =
        new int[2];
        for (int i = 0; i < s.length(); i++)
```

3、矩阵区域和

思路分析：

类比一维求任意区间段的元素和，开一个记忆矩阵，记忆左上角所有元素的和：

最终代码实现：

```
class Solution {
    public int[][] matrixBlockSum(int[][] mat, int K) { int m =
```

```
            s[0][0] = mat[0][0];
            for(int i = 1; i < m; ++i) s[i][0] = s[i – 1][0] + mat[i][0];
            for(int j = 1; j < n; ++j) s[0][j] = s[0][j – 1] + mat[0][j]; for(int i = 1; i < m;
            ++i){
                for(int j = 1; j < n; ++j){
                    s[i][j] = mat[i][j] + s[i][j – 1] + s[i – 1][j] – s[i – 1][j – 1];
                }
            }
            int[][] r = new int[m][n];                  //维恩图，容斥原理，交并补（自己画个图就明白了）
            for(int i = 0; i < m; ++i){
                for(int j = 0; j < n; ++j){
                    int x1 = Math.max(0, i – K), x2 = Math.min(m – 1, i + K);
                    int y1 = Math.max(0, j – K), y2 = Math.min(n – 1, j + K); r[i][j] =
                    s[x2][y2]
        }
    }
```

4、买卖股票的最佳时机含手续费

思路分析：

我们维护两个变量\mathrm{cash}cash 和\mathrm{hold}hold，前者表示当我们不持有股票时的最大利润，后者表示当我们持有股票时的最大利润。

在第 ii 天时，我们需要根据第i − 1i−1 天的状态来更新\mathrm{cash}cash 和\mathrm{hold}hold 的值。

对于\mathrm{cash}cash，我们可以保持不变，或者将手上的股票卖出，状态转移方程为：

cash = max(cash, hold + prices[i] − fee)

对于\mathrm{hold}hold，我们可以保持不变，或者买入这一天的股票，状态转移方程为：

hold = max(hold, cash − prices[i])

在计算这两个状态转移方程时，我们可以不使用临时变量来存储第i − 1i−1 天\mathrm{cash}cash 和\mathrm{hold}hold 的值，而是可以先计算\mathrm{cash}cash 再计算\mathrm{hold}hold，原因是在同一天卖出再买入（亏了一笔手续费）一定不会比不进行任何操作好。

最终代码实现：

```
class Solution {
    public int maxProfit(int[] prices, int fee) { int cash =
        0, hold = -prices[0];
        for (int i = 1; i < prices.length; i++) {
            cash = Math.max(cash, hold + prices[i] - fee); hold =
            Math.max(hold, cash - prices[i]);
        }
```

5、叶值的最小代价生成树

思路分析：

我们发现数组中的数可以划分为两部分，一半是左子树，一半是右子树，根节点就是左边最大和右边最大的乘积。而左右子树里面的值就是当数组中的数为左子树的叶子节点时的情况，右边一样。直到数组中的数只有2个时，答案就是左边右边相乘。

因此我们可以这么看，如果2个数后面又加了一个数，那么我们可以以01为一个节点再和2划分,也可以

0 一个节点和12划分。

树的左右两边至少有1个叶子结点。如果有四个数，有0 123, 01 23, 012 3,同时3个数又有之前的情况。

因此我可以这样找状态，i代表起始点,j代表结束位置。如果我想知道4个数的答案，我就把上面划分的情况算出来，每一个情况还要加上左边和右边的最大值的乘积，作为根节点。于是乎我就是要穷举所有状

态。

最终代码实现：

```
class Solution {
    public int mctFromLeafValues(int[] arr) { int len
        = arr.length;
        int[][] dp = new int[len][len];
        int[][] maxVal = new int[len][len]; for (int i
        = 0; i < len; i++) {
            for (int j = 0; j < len; j++) { int max =
                0;
                for (int k = i; k <= j; k++) if (max < arr[k]) max = arr[k]; maxVal[i][j] =
                max;
            }
        }
        for (int i = 0; i < len; i++) for (int j = 0; j < len; j++) dp[i][j] = Integer.MAX_VALU for (int i = 0; i < len;
        i++) dp[i][i] = 0;
        for (int i = 1; i < len; i++) { // 长度
            for (int j = 0; j < len - i; j++) { // 起始点
                for (int k = j; k < j + i; k++) { // 中间点
```

6、最小路径和

思路分析：

由于路径的方向只能是向下或向右，因此网格的第一行的每个元素只能从左上角元素开始向右移动到达，网格的第一列的每个元素只能从左上角元素开始向下移动到达，此时的路径是唯一的，因此每个元素对应的最小路径和即为对应的路径上的数字总和。

对于不在第一行和第一列的元素，可以从其上方相邻元素向下移动一步到达，或者从其左方相邻元素向右移动一步到达，元素对应的最小路径和等于其上方相邻元素与其左方相邻元素两者对应的最小路径和中的最小值加上当前元素的值。由于每个元素对应的最小路径和与其相邻元素对应的最小路径和有关，因此可以使用动态规划求解。

创建二维数组dp，与原始网格的大小相同，dp[i][j] 表示从左上角出发到(i,j)(i,j) 位置的最小路径和。显然，dp[0][0]=grid[0][0]。对于dp中的其余元素，通过以下状态转移方程计算元素值。

当i>0i>0 且j=0j=0 时，dp[i][0]=dp[i−1][0]+grid[i][0]。

当i=0i=0 且j>0j>0 时，dp[0][j]=dp[0][j−1]+grid[0][j]。

当 i>0i>0 且 j>0j>0 时，dp[i][j]=min(dp[i−1][j],dp[i][j−1])+grid[i][j]。

最后得到dp[m−1][n−1]的值即为从网格左上角到网格右下角的最小路径和。

最终代码实现：

```
class Solution {
    public int minPathSum(int[][] grid) {
        if (grid == null || grid.length == 0 || grid[0].length == 0) { return 0;
        }
        int rows = grid.length, columns = grid[0].length; int[][] dp
        = new int[rows][columns];
        dp[0][0] = grid[0][0];
        for (int i = 1; i < rows; i++) {
            dp[i][0] = dp[i - 1][0] + grid[i][0];
        }
        for (int j = 1; j < columns; j++) {
            dp[0][j] = dp[0][j - 1] + grid[0][j];
        }
        for (int i = 1; i < rows; i++) {
            for (int j = 1; j < columns; j++) {
                dp[i][j] = Math.min(dp[i - 1][j], dp[i][j - 1]) + grid[i][j];
            }
```

双指针，指的是在遍历对象的过程中，不是普通的使用单个指针进行访问，而是使用两个相同方向（快慢指针）或者相反方向（对撞指针）的指针进行扫描，从而达到相应的目的。

换言之，双指针法充分使用了数组有序这一特征，从而在某些情况下能够简化一些运算。 在LeetCode题库中，关于双指针的问题还是挺多的。

1 、盛最多水的容器

思路分析：

- 算法流程：

设置双指针i,j分别位于容器壁两端，根据规则移动指针（后续说明），并且更新面积最大值 res，直到 i=j 时返回res。

- 指针移动规则与证明：

每次选定围成水槽两板高度 h[i],h[j] 中的短板，向中间收窄 11 格。以下证明：

设每一状态下水槽面积为S(i,j),(0<=i<j<n)，由于水槽的实际高度由两板中的短板决定，则可得面积公式S(i,j)=min(h[i],h[j])×(j−i)。

在每一个状态下，无论长板或短板收窄 11 格，都会导致水槽底边宽度−1−1：

若向内移动短板，水槽的短板min(h[i],h[j]) 可能变大，因此水槽面积S(i,j)可能增大。

若向内移动长板，水槽的短板min(h[i],h[j]) 不变或变小，下个水槽的面积一定小于当前水槽面积。

因此，向内收窄短板可以获取面积最大值。换个角度理解：

若不指定移动规则，所有移动出现的S(i,j)的状态数为C(n,2)，即暴力枚举出所有状态。

在状态S(i,j)下向内移动短板至S(i+1,j)（假设h[i]<h[j]），则相当于消去了}S(i,j−1),S(i,j−2),...,S(i,i+1)状态集合。而所有消去状态的面积一定<=S(i,j)：

短板高度：相比S(i,j)相同或更短（<=h[i]）； 底边宽度：相比S(i,j)更短。

因此所有消去的状态的面积都<S(i,j)。通俗的讲，我们每次向内移动短板，所有的消去状态都不会导致丢失面积最大值。

最终代码实现：

```
class Solution {

    public int maxArea(int[] height) {
```

```
            int i = 0, j = height.length - 1, res = 0;
            while(i < j){
                res = height[i] < height[j] ?
                    Math.max(res, (j - i) * height[i++]):
                    Math.max(res, (j - i) * height[j--]);
            }
        }
    }
```

2、寻找重复数

思路分析：

我们定义cnt[i]表示nums[]数组中小于等于i的数有多少个，假设我们重复的数是target，那么[1,target−1] 里的所有数满足cnt[i]≤i，[target,n] 里的所有数满足cnt[i]>i，具有单调性。

以示例 1 为例，我们列出每个数字的cnt值：

nums	**1**	**2**	**3**	**4**
cnt	1	3	4	5

示例中重复的整数是2，我们可以看到[1,1]中的数满足cnt[i]≤i，[2,4]中的数满足cnt[i]>i 。

如果知道cnt[]数组随数字i逐渐增大具有单调性（即target前cnt[i]≤i，target后cnt[i]>i），那么我们就可以直接利用二分查找来找到重复的数。

但这个性质一定是正确的吗？考虑nums[] 数组一共有n+1个位置，我们填入的数字都在[1,n]间，有且只有一个数重复放了两次以上。对于所有测试用例，考虑以下两种情况：

如果测试用例的数组中target出现了两次，其余的数各出现了一次，这个时候肯定满足上文提及的性质，因为小于target 的数i满足cnt[i]=i，大于等于target的数j满足cnt[j]=j+1。

如果测试用例的数组中target出现了三次及以上，那么必然有一些数不在nums[] 数组中了，这个时候相当于我们用target去替换了这些数，我们考虑替换的时候对cnt[] 数组的影响。如果替换的数i小于target，那么[i,target−1]的cnt值均减一，其他不变，满足条件。如果替换的数j大于等于target，那么[target,j−1]的
cnt值均加一，其他不变，亦满足条件。

因此我们生成的数组一定具有上述性质的。

最终代码实现：

```
class Solution {
    public int findDuplicate(int[] nums) { int n =
        nums.length;
        int l = 1, r = n − 1, ans = −1; while (l
        <= r) {
            int mid = (l + r) >> 1; int cnt =
            0;
            for (int i = 0; i < n; ++i) { if
                (nums[i] <= mid) {
                    cnt++;
                }
            }
            if (cnt <= mid) { l =

    }
}
```

3 、返回倒数第 k 个节点

思路分析：

设置快和慢两个指针，初始化时快指针比慢指针多走k−1步，然后两个指针每次都走一步，当快指针到达终点时，慢指针正好处在倒数第k的位置

最终代码实现：

```
public class Solution0202 {

    public int kthToLast(ListNode head, int k) {
        ListNode fast = head;
        for (int i = 1; i < k; i++) { fast =
            fast.next;
        }
        ListNode slow = head;
        while(fast.next != null) { fast =
            fast.next;
            slow = slow.next;
}
```

4、区间列表的交集

思路分析：

- 给定的两个区间列表都是排好序的
- 用两个指针，分别考察A、B数组的子区间
- 根据子区间的左右边界，求出一个交集区间
- 移动指针直至遍历完A、B数组，得到由交集区间组成的数

组图解：

最终代码实现：

```
const intervalIntersection = (A, B) => { const
  res = [];
  let i = 0; let
  j = 0;
  while (i < A.length && j < B.length) {
    const start = Math.max(A[i][0], B[j][0]);
    const end = Math.min(A[i][1], B[j][1]); if (start
```

```
            i++;
        } else {j++;
        }
    }
    return res;
};
```

5、移除元素

思路分析：

既然问题要求我们就地删除给定值的所有元素，我们就必须用O(1)的额外空间来处理它。如何解决？我们可以保留两个指针i和j，其中i是慢指针，j是快指针。

算法：

当nums[j]与给定的值相等时，递增j以跳过该元素。只要nums[j]不等于val，我们就复制 nums[j]到nums[i]并同时递增两个索引。重复这一过程，直到j到达数组的末尾，该数组的新长度为i。

最终代码实现：

```
public int removeElement(int[] nums, int val) { int i = 0;
    for (int j = 0; j < nums.length; j++) { if
        (nums[j] != val) {
            nums[i] = nums[j];
            i++;
        }
    }
    return i;
```

二分查找专题

二分查找也称折半查找（Binary Search），它是一种效率较高的查找方法，前提是数据结构必须先排好序，可以在数据规模的对数时间复杂度内完成查找。但是，二分查找要求线性表具有有随机访问的特点
（例如数组），也要求线性表能够根据中间元素的特点推测它两侧元素的性质，以达到缩减问题规模的效果。

二分查找问题也是面试中经常考到的问题，虽然它的思想很简单，但写好二分查找算法并不是一件容易的事情。

1 、有序矩阵中第K小的元素

思路分析：

- 左上角元素是下限，右下角元素是上限，就有了一个值域，第k 小的元素在这个值域中

- 我们对值域进行二分查找，逼近值域中的目标值，第k 小的元素
- 算出中间值，并求出矩阵里小于等于这个中间值的有几个，count 个
- count 和k 比较，如果比k 小，说明中间值小了，调整值域范围，否则，说明中间值大了，调整值域范围，一步步锁定目标值

为什么对值二分

- 二分查找可以根据索引二分，也可以根据数值二分，有序数组中，索引的大小可以反映值的大小，对索引二分就行
- 但这里不是有序的一维数组，索引不能体现值谁大谁小，无法通过二分索引逼近目标值
- 时间复杂度是 O(log(maxVal−minVal))O(log(maxVal−minVal))

统计矩阵中不大于中间值的元素个数

- 我们可以先让它和一行的最右元素进行比较
- 如果大于等于它，就大于等于它左边所有数和它自己，个数累加进去，考察下一行
- 否则，留在当前行，继续和左边的一个比较
- 时间复杂度是 O(n)O(n)，扫描走了一条折线

最终代码实现：

```
const countInMatrix = (matrix, midVal) => {
  const n = matrix.length;                    // 这题是方阵 n行n列
  let count = 0;
  let row = 0;                                // 第一行
  let col = n - 1;                            // 最后一列
  while (row < n && col >= 0) {
    if (midVal >= matrix[row][col]) { // 大于等于当前行的最右
      count += col + 1;                       // 不大于它的数增加col + 1个
      row++;                                  // 比较下一行
    } else {                                  // 干不过当前行的最右元素
      col--;                                  // 留在当前行，比较左边一个
    }
  }
  return count;


const kthSmallest = (matrix, k) => {
  const n = matrix.length; let

  low = matrix[0][0];
```

```
    while (low <= high) {
        let midVal = low + ((high - low) >>> 1);              // 获取中间值
        let count = countInMatrix(matrix, midVal); // 矩阵中小于等于它的个数
        if (count < k) {
            low = midVal + 1;
        } else {
            high = midVal - 1;
        }
    }
```

2、找出给定方程的正整数解

思路分析：

- x不变时，y增加f增加。y不变时，x增加f增加。
- x从1到1000时，对y用二分，确定f的值。
- 如果f(x,1) > z时，不可能有合适的（x，y）值，算法结束。
- 如果f(x,1000) < z时，x到下一个值。
- 在f(x,1) <= z <= f(x,1000)时，对y在(1,1000)中二分。

最终代码实现：

```
    vector<vector<int>> findSolution(CustomFunction& customfunction, int z)
        { vector<vector<int>> ans;
        int x = 1; int
        y = 1;
        while(x <= 1000){
            if(customfunction.f(x,1) > z) { break;
            }
            if(customfunction.f(x,1000) < z){
                ++x;
                continue;
            }
            int xx=1,yy=1000; int
```

```
                int mv = customfunction.f(x,m);
                if(mv == z){
                    ans.push_back(vector<int>{x,m});
                }
                //yy-xx==1时，m==xx
                if(m == xx){
                    break;
                }
                if(mv < z){
                    xx = m;
                }else{
                    yy = m;
    }
```

3、方阵中战斗力最弱的 K 行

思路分析：

我们计算出方阵中每一行的战斗力（即每一行的元素之和），再对它们进行排序即可。较优的排序方式有两种：

- 待排序的列表中的每个元素包括行号和该行的战斗力。在排序时，我们可以直接按照战斗力为第一关键字，行号为第二关键字进行排序，需要用到的数据都存储在列表内；

- 待排序的列表中的每个元素只包括行号。在排序时，我们需要自己定义比较函数，例如C++ 和 Python 中的lambda 函数，将存储在外部的战斗力数据进行比较。

在排序完成后，我们返回结果的前k 个元素。

最终代码实现：

```
class Solution
{ public:
    vector<int> kWeakestRows(vector<vector<int>>& mat, int k) { int n =
        mat.size();
        vector<pair<int, int>> power; for
        (int i = 0; i < n; ++i) {
            int sum = accumulate(mat[i].begin(), mat[i].end(), 0);
            power.emplace_back(sum, i);
        }
        sort(power.begin(), power.end());
        vector<int> ans;
        for (int i = 0; i < k; ++i) {
```

4、两个数组的交集

思路分析：

迭代并检查第一个数组nums1 中的每个值是否也存在于nums2 中。如果存在，则将值添加到输出。这种方法的时间复杂度为O(n×m)，其中n和m分别为数组nums1 和nums2 的长度。

为了在线性时间内解决这个问题，我们使用集合set 这一数据结构，该结构可以提供平均时间复杂度为O(1)的 in/contains 操作（用于测试某一元素是否为该集合的成员）。

本解法先将两个数组都转换为集合，然后迭代较小的集合，检查其中的每个元素是否同样存在于较大的集合中。平均情况下，这种方法的时间复杂度为O(n+m)。

最终代码实现：

```
class Solution {
    public int[] set_intersection(HashSet<Integer> set1, HashSet<Integer> set2) { int [] output
        = new int[set1.size()];
```

```
        int idx = 0;
        for (Integer s : set1)

        return Arrays.copyOf(output, idx);
    }

    public int[] intersection(int[] nums1, int[] nums2) {
        HashSet<Integer> set1 = new HashSet<Integer>(); for

        (Integer n : nums1) set1.add(n);

        HashSet<Integer> set2 = new HashSet<Integer>();

        if (set1.size() < set2.size()) return set_intersection(set1, set2);
        else return set_intersection(set2, set1);
    }
}
```

5、寻找重复数

思路分析：

定义cnt[i]表示nums[]数组中小于等于i的数有多少个，假设我们重复的数是target，那么[1,target−1]里的所有数满足cnt[i]≤i，[target,n] 里的所有数满足cnt[i]>i，具有单调性。

以示例 1 为例，我们列出每个数字的cnt值：

nums	**1**	**2**	**3**	**4**
cnt	1	3	4	5

示例中重复的整数是2，我们可以看到[1,1]中的数满足cnt[i]≤i，[2,4]中的数满足cnt[i]>i 。

如果知道cnt[] 数组随数字i逐渐增大具有单调性（即target前cnt[i]≤i，target 后cnt[i]>i），那么我们就可以直接利用二分查找来找到重复的数。

但这个性质一定是正确的吗？考虑nums[]数组一共有n+1个位置，我们填入的数字都在[1,n] 间，有且只有一个数重复放了两次以上。对于所有测试用例，考虑以下两种情况：

如果测试用例的数组中target出现了两次，其余的数各出现了一次，这个时候肯定满足上文提及的性质，因为小于target 的数i满足cnt[i]=i，大于等于target的数j满足cnt[j]=j+1。

如果测试用例的数组中target出现了三次及以上，那么必然有一些数不在nums[]数组中了，这个时候相当于我们用target去替换了这些数，我们考虑替换的时候对cnt[]数组的影响。如果替换的数i小于target，那

么[i,target−1]的cnt值均减一，其他不变，满足条件。如果替换的数j大于等于target，那么[target,j−1]的cnt值均加一，其他不变，亦满足条件。

因此我们生成的数组一定具有上述性质的。

最终代码实现：

```
class Solution {
    public int findDuplicate(int[] nums) { int n =
        nums.length;
        int l = 1, r = n − 1, ans = −1; while (l
        <= r) {
            int mid = (l + r) >> 1; int cnt =
            0;
            for (int i = 0; i < n; ++i) { if
                (nums[i] <= mid) {
                    cnt++;
                }
            }
            if (cnt <= mid) { l =

    }
}
```

贪心算法专题

贪心算法（又称贪婪算法）是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，他所做出的是在某种意义上的局部最优解。

贪心算法不是对所有问题都能得到整体最优解，关键是贪心策略的选择，选择的贪心策略必须具备无后效性，即某个状态以前的过程不会影响以后的状态，只与当前状态有关。

1 、根据身高重建队列

思路分析：

1. 排序：

- 按高度降序排列。
- 在同一高度的人中，按k 值的升序排列。

2. 逐个地把它们放在输出队列中，索引等于它们的k 值。

3. 返回输出

队列最终代码实

现：

```
class Solution {
        public int[][] reconstructQueue(int[][] people)
       { Arrays.sort(people, new Comparator<int[]>() {
      @Override
      public int compare(int[] o1, int[] o2) {
        // if the heights are equal, compare k-values

    List<int[]> output = new LinkedList<>();
    for(int[] p : people){
```

```
        }

        int n = people.length;
        return output.toArray(new int[n][2]);
    }
}
```

2、避免重复字母的最小删除成本

思路分析：

把字符串按照连续相同字符分成互不相交的子串，每个子串里面都是最长的连续的相同字符。 因此我们要有最小删除成本，只需要保留每个子串里cost最大的那个字符即可。

最终代码实现：

```
class Solution
{ public:
    int minCost(string s, vector<int>& cost) { int n =
        cost.size();
        vector<int> pre(n);
        pre[0] = 1;
        for(int i=1;i<n;i++){
            if(s[i] == s[i-1]) pre[i] = pre[i-1] + 1; else pre[i] =
            1;
        }
        int all,maxn,ans = 0;
        for(int i=n-1;i>=0;i-=pre[i]){ all =
            0,maxn = 0;
            for(int j=i-pre[i]+1;j<=i;j++){ all +=
```

3、用户分组

思路分析：

使用HashMap 作为辅助字典，记录未组满的用户组。其中用户组大小groupSizes[i]作为key，未组满用户组作为值。

对于用户x，其用户组大小groupSizes[x]。如果有一个大小相同的用户组group 还没有组满，则贪心选择该组，将用户x 加入用户组group。

最终代码实现：

```
/**
 * 用户分组
 *
 * <p> 贪心算法
 */
class Solution {
    public List<List<Integer>> groupThePeople(int[] groupSizes) {

      int N = groupSizes.length;

      List<List<Integer>> ans = new ArrayList<>();
      // 辅助字典：用户组大小 -> 未满用户组
      Map<Integer,List<Integer>> helper = new HashMap<>(); for
      (int i = 0; i < N; i++) {
        // 用户组大小为1
        if(groupSizes[i]==1) {
          ans.add(Arrays.asList(i));
          continue;
        }

        // 用户组
        List<Integer> group =
            helper.getOrDefault(groupSizes[i], new ArrayList<>());
        // 用户入组
```

```
        group.add(i);


        // 用户组新建
        if(group.size()==1) {
          helper.put(groupSizes[i], group);
          ans.add(group);
        }
        // 用户组已满
        else if(group.size()==groupSizes[i])
          { helper.remove(groupSizes[i]);
        }
}
```

```
    return ans;
        }
}
```

4、买卖股票的最佳时机 II

思路分析：

股票买卖策略：

- 单独交易日： 设今天价格p1、明天价格p2，则今天买入、明天卖出可赚取金额 p2−p1（负值代表亏损）。
- 连续上涨交易日： 设此上涨交易日股票价格分别为 p1, p2,..., pn，则第一天买最后一天卖收益最大，即pn−p1p；等价于每天都买卖，即 pn−p1=(p2−p1)+(p3−p2)+...+(pn−p{n−1})。
- 连续下降交易日： 则不买卖收益最大，即不会亏钱。

算法流程：

遍历整个股票交易日价格列表 price，策略是所有上涨交易日都买卖（赚到所有利润），所有下降交易日都不买卖（永不亏钱）。

- 设tmp 为第 i−1 日买入与第i 日卖出赚取的利润，即tmp = prices[i] − prices[i−1] ；
- 当该天利润为正tmp > 0，则将利润加入总利润 profit；当利润为0或为负，则直接跳过；
- 遍历完成后，返回总利润 profit。

最终代码实现：

```
class Solution {
    public int maxProfit(int[] prices) { int profit
        = 0;
        for (int i = 1; i < prices.length; i++) { int tmp =
            prices[i] – prices[i – 1]; if (tmp > 0) profit
            += tmp;
        }
```

5、划分字母区间

思路分析：

策略就是不断地选择从最左边起最小的区间。可以从第一个字母开始分析，假设第一个字母是'a'，那么第一个区间一定包含最后一次出现的'a'。但第一个出现的'a' 和最后一个出现的'a' 之间可能还有其他字母，这些字母会让区间变大。举个例子，在"abccaddbeffe" 字符串中，第一个最小的区间是"abccaddb"。 通过以上的分析，我们可以得出一个算法：对于遇到的每一个字母，去找这个字母最后一次出现的位置，用来更新当前的最小区间。

算法

定义数组last[char] 来表示字符char 最后一次出现的下标。定义anchor 和 j 来表示当前区间的首尾。如果遇到的字符最后一次出现的位置下标大于j，就让j=last[c] 来拓展当前的区间。当遍历到了当前区间的末尾时(即i==j)，把当前区间加入答案，同时将start 设为i+1 去找下一个区间。

最终代码实现：

```
class Solution {

    public List<Integer> partitionLabels(String S) { int[] last

        = new int[26];

            for (int i = 0; i < S.length(); ++i)

        int j = 0, anchor = 0;

        List<Integer> ans = new ArrayList(); for (int i

        = 0; i < S.length(); ++i) {

            j = Math.max(j, last[S.charAt(i) - 'a']); if (i == j) {
```

```
                    anchor = i + 1;
                }
            }
        }
    }
```

链表专题

链表（Linked List）是一种常见的基础数据结构，是一种线性表，但是并不会按线性的顺序存储数据，而是在每一个节点里存到下一个节点的指针（Pointer）。

由于不必须按顺序存储，链表在插入的时候可以达到 O(1)O(1) 的复杂度，比另一种线性表—— 顺序表快得多，但是查找一个节点或者访问特定编号的节点则需要 O(n)O(n) 的时间，而顺序表相应的时间复杂度分别是 O(log\ n)O(log n) 和 O(1)O(1)。

使用链表结构可以克服数组链表需要预先知道数据大小的缺点，链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。但是链表失去了数组随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大。

在计算机科学中，链表作为一种基础的数据结构可以用来生成其它类型的数据结构。链表通常由一连串节点组成，每个节点包含任意的实例数据（data fields）和一或两个用来指向上一个/或下一个节点的位置的链接（links）。链表最明显的好处就是，常规数组排列关联项目的方式可能不同于这些数据项目在

记忆体或磁盘上顺序，数据的访问往往要在不同的排列顺序中转换。而链表是一种自我指示数据类型，因为它包含指向另一个相同类型的数据的指针（链接）。

链表允许插入和移除表上任意位置上的节点，但是不允许随机存取。链表有很多种不同的类型：单向链表，双向链表以及循环链表。

链表通常可以衍生出循环链表，静态链表，双链表等。对于链表使用，需要注意头结点的使用。

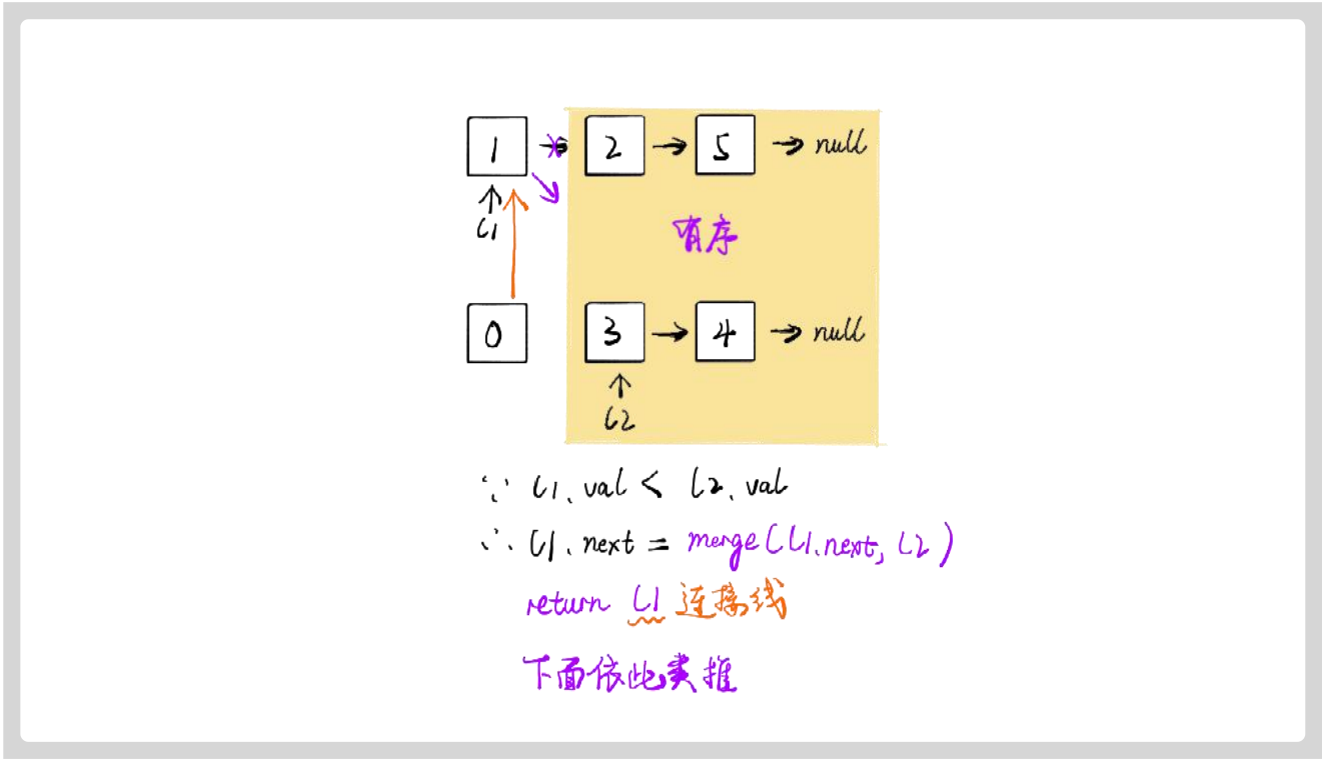
1、合并两个有序链表

思路分析：

这道题可以使用递归实现，新链表也不需要构造新节点，我们下面列举递归三个要素终止条件：两条链表分别名为L1 和L2，当L1 为空或L2 为空时结束返回值：每一层调用都返回排序好的链表头递归内容：

如果L1 的 val 值更小，则将L1.next 与排序好的链表头相接，L2 同理O(m+n)O(m+n)，mm 为L1的长度，nn 为L2 的长度

图解：



最终代码实现：

```
class Solution {

    public ListNode mergeTwoLists(ListNode L1, ListNode L2) { if(L1

        == null) {

            return L2;

        }
```

```
            return L1;
        }

        if(L1.val < L2.val) {
            L1.next = mergeTwoLists(L1.next, L2);

            return L1;

        } else {
            L2.next = mergeTwoLists(L1, L2.next);
    }
}
```

2、复杂链表的复制

思路分析：

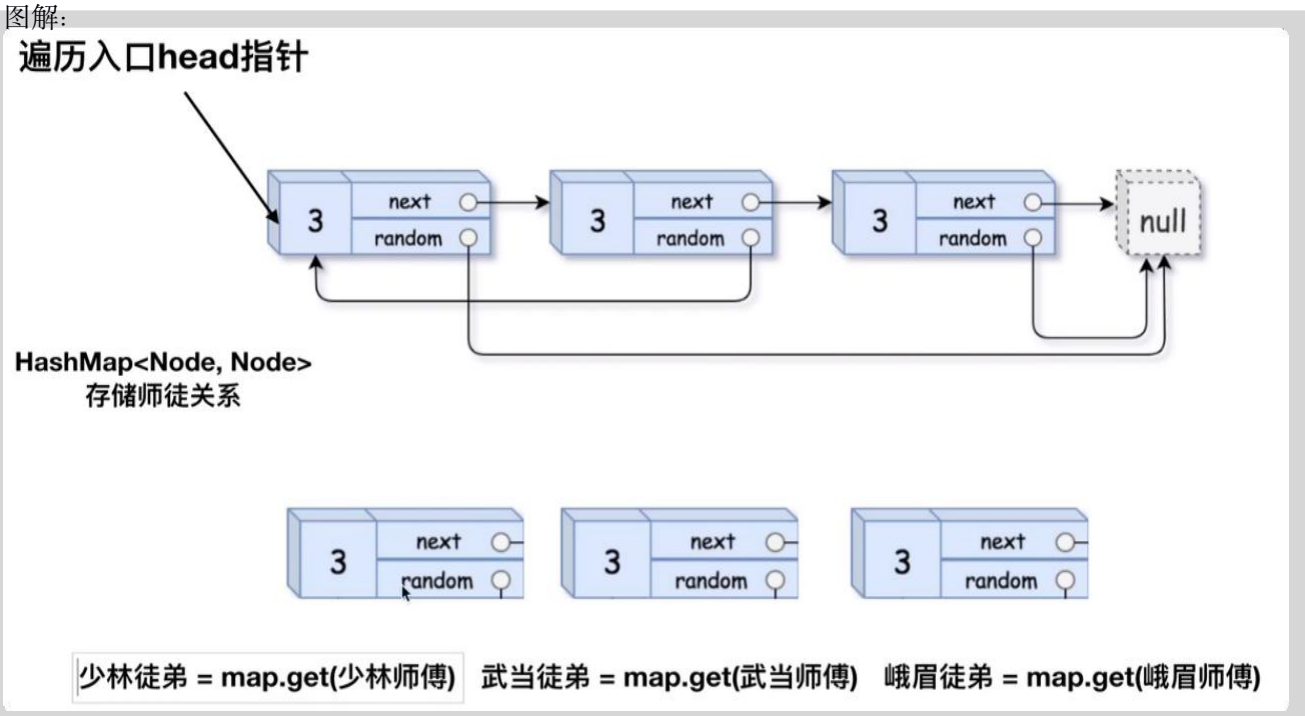
1.创建

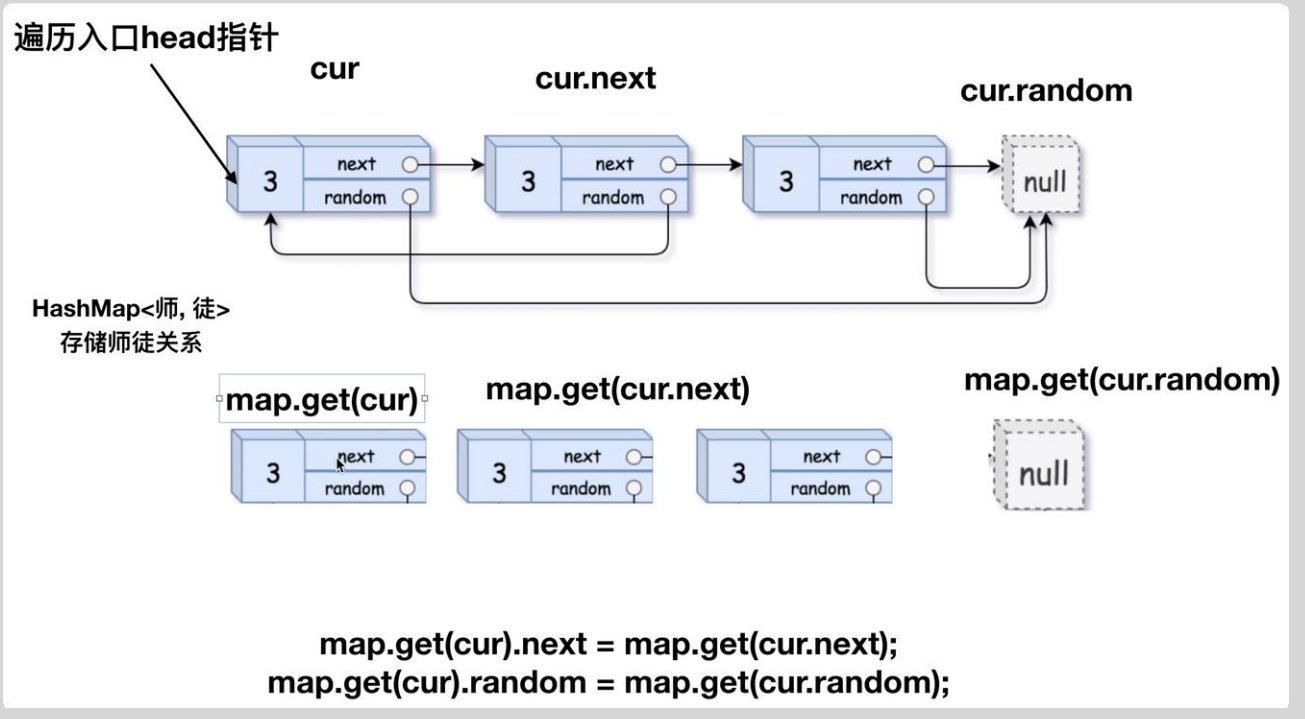
HashMap 2.复

制结点值

3.复制指向（next,random）

图解：





最终代码实现：

```
/*
// Definition for a Node. class
Node {
    int val;
    Node next;
    Node random;

    public Node(int val)
        { this.val = val;
        this.next = null;
        this.random = null;
    }
}
*/
class Solution { //HashMap实现
    public Node copyRandomList(Node head) {
        HashMap<Node,Node> map = new HashMap<>(); //创建HashMap集
        合Node cur=head;
        //复制结点值
        while(cur!=null){
            //存储put:<key,value1>
            map.put(cur,new Node(cur.val)); //顺序遍历，存储老结点和新结点(先存储新创建的结点值)
            cur=cur.next;
        }
        //复制结点指向
        cur = head;
        while(cur!=null){
```

```
                //得到get:<key>.value2,3
                map.get(cur).next = map.get(cur.next); //新结点next指向同旧结点的next指向
                map.get(cur).random = map.get(cur.random); //新结点random指向同旧结点的random指向cur
                = cur.next;
            }

            //返回复制的链表
            return map.get(head);
        }
    }
```

3、返回倒数第 k 个节点

思路分析：

设置快和慢两个指针，初始化时快指针比慢指针多走k−1步，然后两个指针每次都走一步，当快指针到达终点时，慢指针正好处在倒数第k的位置

最终代码实现：

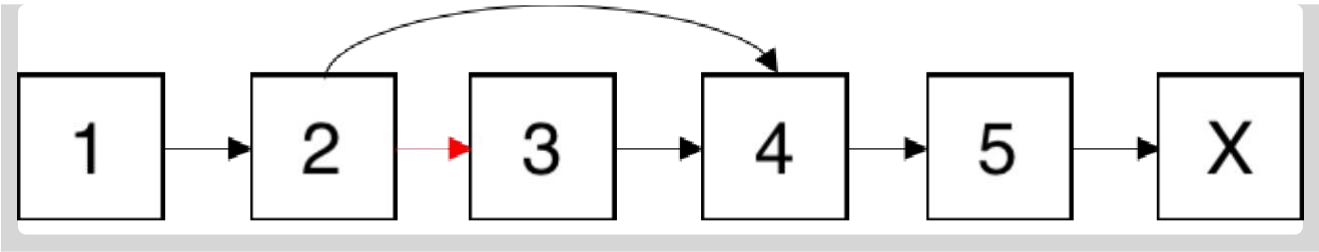
```
public class Solution {

    public int kthToLast(ListNode head, int k) {
        ListNode fast = head;
        for (int i = 1; i < k; i++) { fast =
            fast.next;
        }
        ListNode slow = head;
        while(fast.next != null) { fast =
            fast.next;
            slow = slow.next;
}
```

4、删除链表中的节点

思路分析：

从链表里删除一个节点node 的最常见方法是修改之前节点的next 指针，使其指向之后的节点。



因为，我们无法访问我们想要删除的节点之前的节点，我们始终不能修改该节点的next 指针。相反，我们必须将想要删除的节点的值替换为它后面节点中的值，然后删除它之后的节点。

因为我们知道要删除的节点不是列表的末尾，所以我们可以保证这种方法是可行的。

最终代码实现：

```
public void deleteNode(ListNode node)

    { node.val = node.next.val;

    node.next = node.next.next;
}
```

5、排序链表

思路分析：

- 数组进行归并排序时，需要一个额外的数组记录归并结果，因此数组归并的空间复杂度是O(n)+O(logn)，而对于链表的话，直接交换引用即可，不用额外的空间保存，所以只需要O(logn)的递归空间复杂度即可
- 使用快慢指针把链表分成两半，要断链，同时断了之后还要有两个链表的头节点，所以用快慢指针断链的时候需要一个结点pre记录slow的前驱结点，然后 pre->next==nullptr,那么两个链表即为[head,pre]和[slow,nullptr)
- 合并分开的链表的时候，方法就是21题中的合并排序链表的递归的方法

最终代码实现：

```
class Solution

{ public:

    ListNode* sortList(ListNode* head)
    {

        if(head==nullptr || head->next==nullptr) return

            head;
```

```
            //记录slow的前驱，针对链表有偶数个结点的情况
            pre=slow;
            slow=slow->next;
            fast=fast->next->next;
        }
        pre->next=nullptr;
        //明确递归函数的返回结果是什么，sortList的返回结果就是已经排序号的链表的头节点；而merge的返回结
        return merge(sortList(head),sortList(slow));
    }
    ListNode* merge(ListNode* l1,ListNode* l2)
    {
        if(l1==nullptr)
            return l2;
        if(l2==nullptr)
            return l1;
        if(l1->val<=l2->val)
        {
            l1->next=merge(l1->next,l2);
};
```

6、奇偶链表

思路分析：

将奇节点放在一个链表里，偶链表放在另一个链表里。然后把偶链表接在奇链表的尾部。

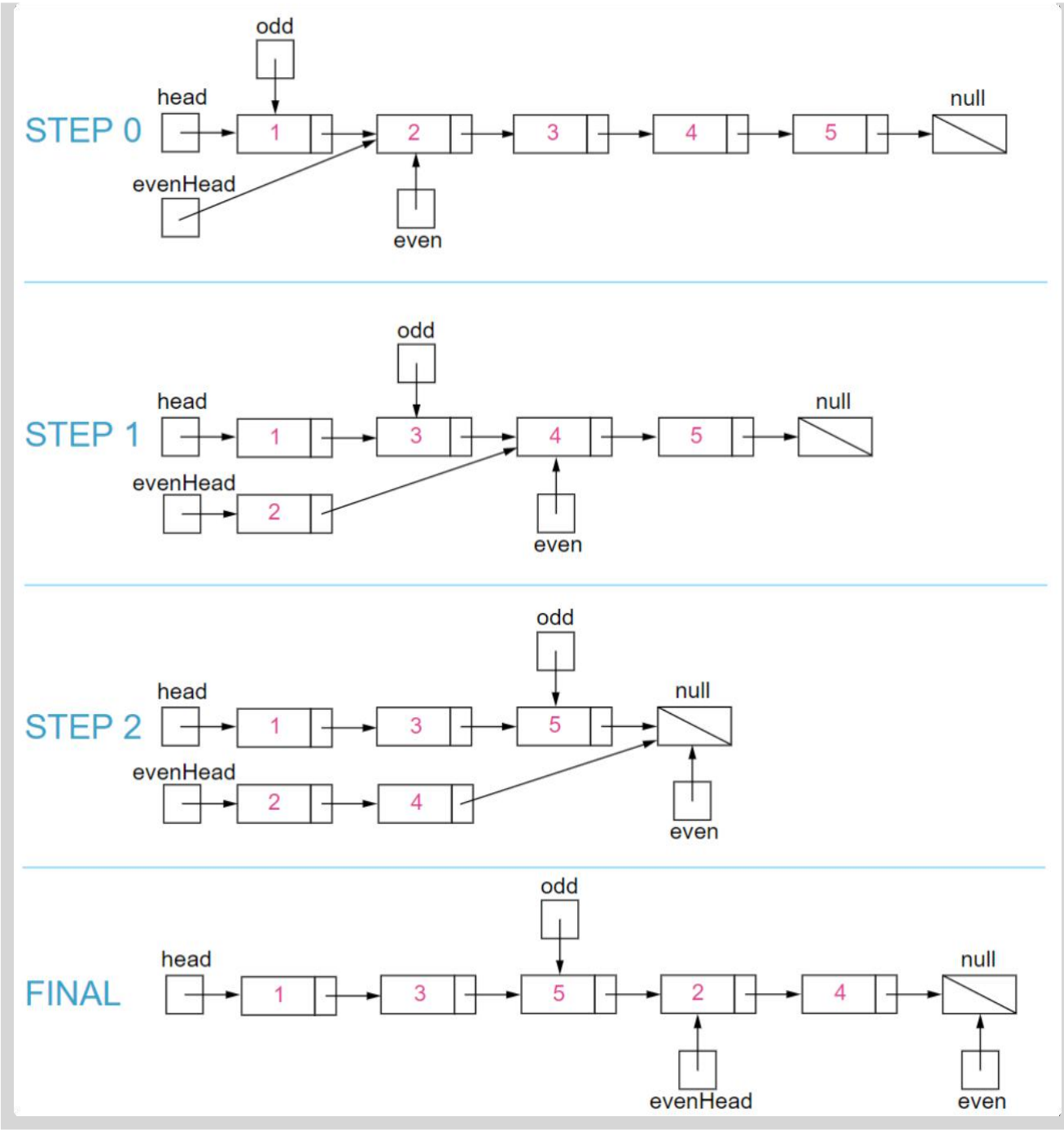
算法

这个解法非常符合直觉思路也很简单。但是要写一个精确且没有 bug 的代码还是需要进行一番思索的。

一个 LinkedList 需要一个头指针和一个尾指针来支持双端操作。我们用变量 head 和odd 保存奇链表的头和尾指针。 evenHead 和even 保存偶链表的头和尾指针。算法会遍历原链表一次并把奇节点放到奇链表里去、偶节点放到偶链表里去。遍历整个链表我们至少需要一个指针作为迭代器。这里odd 指针和 even 指针不仅仅是尾指针，也可以扮演原链表迭代器的角色。

解决链表问题最好的办法是在脑中或者纸上把链表画出来。

图解：



最终代码实现：

```
public class Solution {

    public ListNode oddEvenList(ListNode head) { if

        (head == null) return null;

        ListNode odd = head, even = head.next, evenHead = even;

        while (even != null && even.next != null) { odd.next

            = even.next;

            odd = odd.next;

            even.next = odd.next;

            even = even.next;
```

```
    }
}
```

广度优先搜索专题

宽度优先搜索算法（又称广度优先搜索）是最简便的图的搜索算法之一，这一算法也是很多重要的图的算法的原型。Dijkstra单源最短路径算法和Prim最小生成树算法都采用了和宽度优先搜索类似的思想。

其别名又叫BFS，属于一种盲目搜寻法，目的是系统地展开并检查图中的所有节点，以找寻结果。换句话说，它并不考虑结果的可能位置，彻底地搜索整张图，直到找到结果为止。

1 、二叉树的层次遍历 II

思路分析：

树的层次遍历可以使用广度优先搜索实现。从根节点开始搜索，每次遍历同一层的全部节点，使用一个列表存储该层的节点值。

如果要求从上到下输出每一层的节点值，做法是很直观的，在遍历完一层节点之后，将存储该层节点值的列表添加到结果列表的尾部。这道题要求从下到上输出每一层的节点值，只要对上述操作稍作修改即可：在遍历完一层节点之后，将存储该层节点值的列表添加到结果列表的头部。

为了降低在结果列表的头部添加一层节点值的列表的时间复杂度，结果列表可以使用链表的结构，在链表头部添加一层节点值的列表的时间复杂度是O(1)。在Java 中，由于我们需要返回的 List 是一个接口，这里可以使用链表实现；而C++ 或Python 中，我们需要返回一个vector 或 list，它不方便在头部插入元素（会增加时间开销），所以我们可以先用尾部插入的方法得到从上到下的层次遍历列表，然后再进行反转。

最终代码实现：

```
class Solution {

    public List<List<Integer>> levelOrderBottom(TreeNode root) {

        List<List<Integer>> levelOrder = new LinkedList<List<Integer>>(); if (root ==

        null) {

            return levelOrder;

        }

        Queue<TreeNode> queue = new LinkedList<TreeNode>();
```

```
                List<Integer> level = new ArrayList<Integer>();
                int size = queue.size();
                for (int i = 0; i < size; i++) {
                    TreeNode node = queue.poll();
                    level.add(node.val);
                    TreeNode left = node.left, right = node.right; if (left !=
                    null) {
                        queue.offer(left);
                    }
                    if (right != null) {
                        queue.offer(right);
                    }
        }
}
```

2 、跳跃游戏 III

思路分析：

我们可以使用广度优先搜索的方法得到从start 开始能够到达的所有位置，如果其中某个位置对应的元素值为 0，那么就返回True。

具体地，我们初始时将start 加入队列。在每一次的搜索过程中，我们取出队首的节点u，它可以到达的位置为u + arr[u] 和 u − arr[u]。如果某个位置落在数组的下标范围[0, len(arr)) 内，并且没有被搜索过， 则将该位置加入队尾。只要我们搜索到一个对应元素值为 0 的位置，我们就返回True。在搜索结束后， 如果仍然没有找到符合要求的位置，我们就返回False。

最终代码实现：

```
class Solution {

public:

    bool canReach(vector<int>& arr, int start) { if

        (arr[start] == 0) {


        int n = arr.size();

        vector<bool> used(n);

        queue<int> q;
```

```
            used[start] = true;

            while (!q.empty()) {
                int u = q.front();
                q.pop();
                if (u + arr[u] < n && !used[u + arr[u]]) { if (arr[u +
                    arr[u]] == 0) {
                        return true;
                    }
                    q.push(u + arr[u]);
                    used[u + arr[u]] = true;
                }
                if (u - arr[u] >= 0 && !used[u - arr[u]]) { if (arr[u -
                    arr[u]] == 0) {
                        return true;
                    }
        }
};
```

3、从上到下打印二叉树

思路分析：

1. 特例处理： 当树的根节点为空，则直接返回空列表[] ；

2. 初始化： 打印结果列表res = [] ， 包含根节点的队列queue = [root] ；

3. BFS 循环： 当队列queue 为空时跳出；

- 出队： 队首元素出队，记为node；
- 打印： 将node.val 添加至列表tmp 尾部；
- 添加子节点： 若node 的左（右）子节点不为空，则将左（右）子节点加入队列queue ；

4. 返回值： 返回打印结果列表res 即

可。最终代码实现：

```
class Solution {
```

```
        if(root == null) return new int[0];
        Queue<TreeNode> queue = new LinkedList<>(){{ add(root); }};
        ArrayList<Integer> ans = new ArrayList<>();
        while(!queue.isEmpty()) {
            TreeNode node = queue.poll();
            ans.add(node.val);
            if(node.left != null) queue.add(node.left);
            if(node.right != null) queue.add(node.right);
        }
    }
}
```

4、扫雷游戏

思路分析：

给定初始二维数组和起点，返回修改后的二维数组。

若起点处是雷，即‘M’，直接将其修改为'X'，游戏结束； 若起点处是空，即‘E’，则从起点开始向 8 邻域中的空地搜索，直到到达邻接　　的空地停止。 和二叉树从根结点开始搜索，直到达到叶子节点停止，是几乎一样的，所以会写二叉树的BFS/DFS，那么这题也就写出来了。

最终代码实现：

```
class Solution {
    // 定义 8 个方向
    int[] dx = {–1, 1, 0, 0, –1, 1, –1, 1};

    public char[][] updateBoard(char[][] board, int[] click) {
        // 1. 若起点是雷，游戏结束，直接修改 board 并返回。
        int x = click[0], y = click[1]; if
        (board[x][y] == 'M') {
            board[x][y] = 'X';

        // 2. 若起点是空地，则将起点入队，从起点开始向 8 邻域的空地进行宽度优先搜索。
        int m = board.length, n = board[0].length;
        boolean[][] visited = new boolean[m][n];
        visited[x][y] = true;
        Queue<int[]> queue = new LinkedList<>();
        queue.offer(new int[] {x, y});
        while (!queue.isEmpty()) {
```

```
                int cnt = 0;
                for (int k = 0; k < 8; k++) { int
                    newX = i + dx[k];
                       int newY = j + dy[k];
                    if (newX < 0 || newX >= board.length || newY < 0 || newY >= board[0].length) {
                        continue;
                    }
                    if (board[newX][newY] == 'M')
                        { cnt++;
                    }
                }
                // 若空地 (i, j) 周围有雷，则将该位置修改为雷数；否则将该位置更新为 'B'，并将其 8 邻域中的
                if (cnt > 0) {
                    board[i][j] = (char)(cnt + '0');
                } else {
                    board[i][j] = 'B';
                    for (int k = 0; k < 8; k++) { int
                        newX = i + dx[k];
                           int newY = j + dy[k];
                        if (newX < 0 || newX >= board.length || newY < 0 || newY >= board[0].length
                            || board[newX][newY] != 'E' || visited[newX][newY]) {
                            continue;
                        }
                        visited[newX][newY] = true;
                        queue.offer(new int[] {newX, newY});
                    }
                }
            }
            return board;
        }
    }
```

5、找树左下角的值

思路分析：

通常BFS遍历都是从上到下，从左到右。然而根据题目意思，是要取到最下面，最左边的元素。故只需要对BFS遍历稍作改进即可。具体思路为从上到下保持不变，但水平遍历方向改为从右到左即可。如此一来，先上后下，先右后左，此策略走下去，最后一个元素必然是最下方最左边的元素，最后返回该节点node.val 即 可

最终代码实现：

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def findBottomLeftValue(self, root: TreeNode) -> int:
        queue = [root]
        while queue:
            node = queue.pop(0)
            if node.right: # 先右后左
                queue.append(node.right)
            if
```

深度优先搜索专题

深度优先搜索算法（英语：Depth-First-Search，DFS）是一种用于遍历或搜索树或图的算法。沿着树的深度遍历树的节点，尽可能深的搜索树的分支。当节点v的所在边都已被探寻过，搜索将回溯到发现节点v 的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。属于盲目搜索。

深度优先搜索是图论中的经典算法，利用深度优先搜索算法可以产生目标图的相应拓扑排序表，利用拓扑排序表可以方便的解决很多相关的图论问题，如最大路径问题等等。

1、检查平衡性

思路分析：

1. 根据平衡性的定义，计算每个节点的左右子树高度。

2. 如果某个节点的左右子树高度差大于1，则不平衡，返回false，算法结束；否则说明以当前节点为根的子树是平衡的，继续检查当前节点的左右子树的平衡性。

最终代码实现：

```
/**
 * Definition for a binary tree node.
 *      int val;
 *      TreeNode left;
 *      TreeNode right;
 *      TreeNode(int x) { val = x; }
```

```
 * }
 */
class Solution {
    public boolean isBalanced(TreeNode root) {
        // 根结点为null，说明是棵空树，认为是平衡的
        if (root == null)
            { return true;
        }
        int leftTreeDepth = treeDepth(root.left); // 左子树高度 int
        rightTreeDepth = treeDepth(root.right); // 右子树高度 if
        (Math.abs(leftTreeDepth – rightTreeDepth) > 1) {

    }
    public int treeDepth(TreeNode root) {
        // 递归出口，空树的高度为0
        if (root == null) {
            return 0;
        }
        // 当前二叉树的高度 = max(左子树高度,右子树高度) + 1
}
```

2、求根到叶子节点数字之和

思路分析：

首先依然是判断root,没有的话输出0。然后设置一个变量self,sum=0，来保存所有数字之和。解法与上面广度优先搜索类似，只是广度优先搜索是尽可能的广度搜索，而深度优先搜索是一直往下。

接着就是建立递归函数dfs了，分别有两个变量node,sum,前一个是节点，后一个是节点值的和。

首先是判断该节点是否有左子节点，有的话前一个加入左子节点node.left,后一个加入字符串化了的该节点的值，以及字符串化了的该节点左子节点的值sum+str(node.left.val)，然后递归调用函数dfs，如果一直有左子节点就一直这么递归，直到叶子节点。判断右子节点类似就不重复了。

那么到叶子节点，自然就是没有左子节点与右子节点了，就把前面一直字符串相加的sum，变成整数然后加给self.sum，函数就结束了。

通过dfs(root,str(root.val))开始函数，然后最后return self.sum。

最终代码实现：

```
class Solution:
    def sumNumbers(self, root: TreeNode) -> int: if not
        root: return 0
        self.sum = 0
        def dfs(node,sum): if
            node.left:
                dfs(node.left,sum+str(node.left.val)) if
            node.right:
                dfs(node.right,sum+str(node.right.val)) if not
```

3、二叉树的所有路径

思路分析：

在深度优先搜索遍历二叉树时，我们需要考虑当前的节点以及它的孩子节点。

- 如果当前节点不是叶子节点，则在当前的路径末尾添加该节点，并继续递归遍历该节点的每一个孩子节点。
- 如果当前节点是叶子节点，则在当前路径末尾添加该节点后我们就得到了一条从根节点到叶子节点的路径，将该路径加入到答案即可。

如此，当遍历完整棵二叉树以后我们就得到了所有从根节点到叶子节点的路径。

最终代码实现：

```
class Solution {
    public List<String> binaryTreePaths(TreeNode root)
        { List<String> paths = new ArrayList<String>();
        constructPaths(root, "", paths); return

    public void constructPaths(TreeNode root, String path, List<String> paths) {
```

```
        if (root != null) {
            StringBuffer pathSB = new StringBuffer(path);
            pathSB.append(Integer.toString(root.val));
            if (root.left == null && root.right == null) { // 当前节点是叶子节点
                paths.add(pathSB.toString());  // 把路径加入到答案中
            } else {
                pathSB.append("->"); // 当前节点不是叶子节点，继续递归遍历
                constructPaths(root.left, pathSB.toString(), paths);
    }
}
```

4、路径总和 II

思路分析：

当到达叶子结点，并且数组和等于sum，将数据拷贝一份到res。 因为题目没有给出节点大小是正数的条件，因此不能剪枝。

最终代码实现：

```
var pathSum = function(root, sum)
    { if(!root){
        return [];
    }
    let res = [];
    dfs(root, sum, root.val, [root.val], res ); return res;
};

let dfs = function(node, sum, now, nowarr, res){ if(now ==
    sum && !node.left && !node.right){
        res.push(nowarr.map(it=>it));
        return;
    }
    if(node.left){
        dfs(node.left, sum, now + node.left.val, nowarr.concat([node.left.val]), res);
    }
```

5、在每个树行中找最大值

思路分析：

深度搜索的同时记录一下当前正在查看的节点所在的层数level和当前所找的各层的最大值res，把当前节点的value和res[level]比较一下取较大者即可；

若res[level]为空说明第一次跑到这一层，直接添加

最终代码实现：

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<Integer> largestValues(TreeNode root)
        { List<Integer> res = new ArrayList<>();
        levelOrder(root, res, 0);
        return res;
    }

    public void levelOrder(TreeNode root, List<Integer> res, int level) {//当前root是第level层if(root==null)
        return;
        if(level>=res.size()) {res.add(root.val);} else {
            int max_value = Math.max(res.get(level), root.val);
            res.set(level, max_value);
        }
        levelOrder(root.left,res,level+1);
        levelOrder(root.right,res,level+1);

    }
```

```
}
```

树专题

树是一种抽象数据类型（ADT）或是实现这种抽象数据类型的数据结构，用来模拟具有树状结构性质的数据集合。它是由n(n>0)个有限节点组成一个具有层次关系的集合。

把它叫做「树」是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。

它具有以下的特点：

- 每个节点都只有有限个子节点或无子节点；
- 没有父节点的节点称为根节点；
- 每一个非根节点有且只有一个父节点；
- 除了根节点外，每个子节点可以分为多个不相交的子树；
- 树里面没有环路。

1、平衡二叉树

思路分析：

自底向上递归的做法类似于后序遍历，对于当前遍历到的节点，先递归地判断其左右子树是否平衡，再判断以当前节点为根的子树是否平衡。如果一棵子树是平衡的，则返回其高度（高度一定是非负整数），否则返回−1−1。如果存在一棵子树不平衡，则整个二叉树一定不平衡。

最终代码实现：

```
class Solution {
    public boolean isBalanced(TreeNode root)
        { return height(root) >= 0;

    public int height(TreeNode root) {
        if (root == null)
            { return 0;
        }
        int leftHeight = height(root.left);
        int rightHeight = height(root.right);
        if (leftHeight == -1 || rightHeight == -1 || Math.abs(leftHeight - rightHeight) > 1) { return -1;
        } else {
            return Math.max(leftHeight, rightHeight) + 1;
}
```

2、从根到叶的二进制数之和

思路分析：

1.常规栈实现二叉树的深度遍历 2.需要价格参数记录从根到最深叶子节点的所有值，并转换为二进制 3.在到达叶子结点时，记录在res列表中

最终代码实现：

```
# Definition for a binary tree node.
# class TreeNode:
#         def __init__(self, x):
#                 self.val = x
```

```
#             self.left = None
#             self.right = None

class Solution:
    def sumRootToLeaf(self, root: TreeNode) -> int: if not
        root:

        res = []
        q = []
        q.append([root, '0b']) # 构建二进制的叶子结点字符串
        while q:
            node, tmp = q.pop(0) tmp

            if node.left:
                q.append([node.left, tmp]) if
            node.right:

            if not node.left and not node.right:
                res.append(tmp)
        return sum(int(bin_str, 2)          for bin_str in res)
```

3、删点成林

思路分析：

只需要做一次 DFS，如果当前节点存在于 to_delete，则parent.left = None 或者parent.right = None（取决于要删除的节点是父节点的左孩子还是右孩子），并将其加入ans，最后返回ans。

因此dfs 的时候，扩展一下参数，将 parent 和是左还是右带上即可dfs(node, parent, direction)。

同时，为了减少搜索to_delete 的时间复杂度，将其转换为了set。

最终代码实现：

```
class Solution:
    def delNodes(self, root: TreeNode, to_delete: List[int]) -> List[TreeNode]: if not root:
        return None
```

```
        ans = [] if root.val in mapper else [root] def

        dfs(node, parent, direction):

            if not node: return

            dfs(node.left, node, 'left')

            dfs(node.right, node, 'right') if

            node.val in mapper:

                if node.left: ans.append(node.left) if

                node.right: ans.append(node.right) if

                direction == 'left':

                    parent.left = None
```

4、最深叶节点的最近公共祖先

思路分析：

如果当前节点是最深叶子节点的最近公共祖先，那么它的左右子树的高度一定是相等的，否则高度低的那个子树的叶子节点深度一定比另一个子树的叶子节点的深度小，因此不满足条件。所以只需要dfs遍历找到左右子树高度相等的根节点即出答案。

最终代码实现：

```
/**
 * Definition for a binary tree node.

 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public TreeNode lcaDeepestLeaves(TreeNode root)

        { if(root==null)

        int rd=depth(root.right);
```

```
            if(ld==rd){
                return root;
            }
            else if(ld>rd){
                return lcaDeepestLeaves(root.left);
            }
            else {
                return lcaDeepestLeaves(root.right);
            }
        }
        public int depth(TreeNode
            node){ if(node==null)
                return 0:
}
```

5、翻转二叉树

思路分析：

这是一个非常经典的树的问题，这个问题很适合用递归方法来解决。

反转一颗空树结果还是一颗空树。对于一颗根为r，左子树为mboxright，右子树为mboxleft 的树来说，它的反转树是一颗根为r，左子树为mboxright的反转树，右子树为mboxleft的反转树的树。

最终代码实现：

```
public TreeNode invertTree(TreeNode root) { if
    (root == null) {
        return null;
    }
    TreeNode right = invertTree(root.right);
    TreeNode left = invertTree(root.left);
    root.left = right;
    root.right = left;
```

字符串专题

字符串是由零个或多个字符组成的有限序列。一般记为s=a1a2...an。它是编程语言中表示文本的数据类型。

字符串与数组有很多相似之处，比如使用名称[下标] 来得到一个字符。然而，字符串有其鲜明的特点，即结构相对简单，但规模可能是庞大的。

以生物中的DNA 序列为例，假设一个 DNA 序列为"GCCGTAATATCG...", 在人体中，该序列的长度可能会达到n×10^8。

然而，构成序列的基本碱基种类只有 "A", "T", "G", "C" 4 种。

这里我们可将 "A", "T", "G", "C" 看作一个字符集，由字符集中的一些字符组合而成的 DNA 序列可以看作一个字符串。

在编程语言中，字符串往往由特定字符集内有限的字符组合而成，根据其特点，对字符串的操作可以归结为以下几类：

- 字符串的比较、连接操作（不同编程语言实现方式有所不同）；
- 涉及子串的操作，比如前缀，后缀等；
- 字符串间的匹配操作，如 KMP 算法、BM 算法等。

1 、上升下降字符串

思路分析：

可以开一个长度为 26 的数组表示 26 个桶，每个桶里存放一种字母。先用O(|s|)的时间扫描一遍字符串（其中|s|代表字符串的长度），统计每个字母出现的次数。然后我们只要不停地扫描这里的「桶序列」——先从小到大扫，再从大到小扫，每次发现一个桶当中计数值不为 0 的时候，就把这个桶对应的字母添加到结果字符串的最后方，然后对计数值减一。

具体地，开一个长度为 26 的数组h[]，作为用来计数的「桶」。haveChar 的功能是在每次循环开始执行之前判断是否还有未使用的字符。appendChar 的功能是检测当前位置的桶是否计数值为 0，如果不为 0 则修改目标串和计数值。

最终代码实现：

```
class Solution:
    def sortString(self, s: str) -> str: h = [0] *
        26
        for ch in s:
            def appendChar(ret, p):
                if h[p] > 0:
                    h[p] -= 1
```

```
        def haveChar():
            return any(h[i] > 0 for i in range(26))
        ret = list()
        while True:
            if not haveChar():
                break
            for i in range(26):
                appendChar(ret, i) for i
            .           (26)
```

2、生成每种字符都是奇数个的字符串

思路分析：

1. 奇数直接用一个字符*n

2. 偶数转换成奇

数+1 最终代码实现：

```
class Solution:
    def generateTheString(self, n: int) -> str: return 'a'*n
        if n%2==1 else 'a'*(n-1)+'b'
```

3、最优除法

思路分析：

这题的暴力方法是将列表分成leftleft 和rightright 两部分并分别对它们调用函数。我们从startstart 到endend 遍历 ii 并得到left=(start,i)left=(start,i) 和right=(i+1,end)right=(i+1,end)。

leftleft 和rightright 分别返回它们对应部分的最大和最小值和它们对应的字符串。

最小值可以通过左边部分的最小值除以右边部分的最大值得到，也就是minVal=left.min/right.maxminVal=left.min/right.max。

类似的，最大值可以通过左边部分的最大值除以右边部分的最小值得到，也就是maxVal=left.max/right.minmaxVal=left.max/right.min。

现在，怎么添加括号呢？由于出发运算是从左到右的，也就是最左边的除法默认先执行，所以我们不需要给左边部分添加括号，但我们需要给右边部分添加括号。

比方假设左边部分是"2"，右边部分是"3/4"，那么结果字符串"2/(3/4)" 对应的是左边部分+"/"+"("+右边部分+")"。

还有一点，如果右边部分只有一个数字，我们也不需要添加括号。

也就是说，如果左边部分是"2" 且右边部分是"3" （只包含单个数字），那么答案应该是"2/3" 而不是"2/(3)"。

最终代码实现：

```
public class Solution {

    public String optimalDivision(int[] nums) {

        T t = optimal(nums, 0, nums.length – 1, "");

        return t.max_str;

    }

    class T {

        float max_val, min_val;

        String min_str, max_str;

    }

    public T optimal(int[] nums, int start, int end, String res) { T t = new T();

        if (start == end) {

            t.max_val = nums[start];

            t.min_val = nums[start];

            t.min_str = "" + nums[start];

            t.max_str = "" + nums[start];

            return t;

        }

        t.min_val = Float.MAX_VALUE;

        t.max_val = Float.MIN_VALUE;

        t.min_str = t.max_str = "";

        for (int i = start; i < end; i++) {
```

```
T left = optimal(nums, start, i, "");

T right = optimal(nums, i + 1, end, "");

if (t.min_val > left.min_val / right.max_val) { t.min_val =

    left.min_val / right.max_val;

    t.min_str = left.min_str + "/" + (i + 1 != end ? "(" : "") + right.max_str + (i

}

if (t.max_val < left.max_val / right.min_val) { t.max_val =

    left.max_val / right.min_val;

    t.max_str = left.max_str + "/" + (i + 1 != end ? "(" : "") + right.min_str + (i
```

```
                }
            }

        }
    }
```

4、旅行终点站

思路分析：

关键就是要从出发城市from去找到它到达的下一个城市，如果发现它没有可以到达的城市，那么它就是终点，否则就将from移动到下一个城市。

题目中已经说过保证线路图会形成一条不存在循环的线路，所以从哪一个城市出发最终结果都一样，那么将String from = paths.get(0).get(0);。

如何来查找某个城市能到达的下一个城市，可以使用一个Map<String, String> map，键为出发城市，值为到达城市。构造这个map使用Map<String, String> prepare(List<List> paths)。

于是，发现它没有可以到达的城市就是!map.containsKey(from)，此时return from。 否则就将from移动到下一个城市。from = map.get(from);。

while循环一定会结束，因为题目说了，一定有终点。

最终代码实现：

```
public class destCity {
    public String destCity(List<List<String>> paths)
        { Map<String, String> map = prepare(paths);
        String from = paths.get(0).get(0);
        while(true){
            if(!map.containsKey(from))


    private Map<String, String> prepare(List<List<String>> paths){
        Map<String, String> map = new HashMap<>();
        for(List<String> path : paths)
            map.put(path.get(0), path.get(1)); return
}
```

5、变位词组

思路分析：

用字典来构造一个key，value的结构。遍历strs将每个元素排序，按照排序后的结果进行添加入对应的key,value里面。

最后用字典构造列表进行返回，达到题目要求的结构

最终代码实现：

```
class Solution(object):
    def groupAnagrams(self, strs): """
        :type strs: List[str]
        :rtype: List[List[str]] """
        temp_dict = {}
        for i in strs:
            s = list(i)
            s.sort()
            temp = ''.join(s)
            if temp in temp_dict.keys():
                temp_dict[temp].append(i)
                continue
```

– END –