1.21 有效的数独

判断一个 9x9 的数独是否有效。只需要**根据以下规则**，验证已经填入的数字是否有效即可。

1. 数字 `1-9` 在每一行只能出现一次。
2. 数字 `1-9` 在每一列只能出现一次。
3. 数字 `1-9` 在每一个以粗实线分隔的 `3x3` 宫内只能出现一次。

数独部分空格内已填入了数字，空白格用 `'.'` 表示。

说明:

- 一个有效的数独（部分已被填充）不一定是可解的。
- 只需要根据以上规则，验证已经填入的数字是否有效即可。
- 给定数独序列只包含数字 `1-9` 和字符 `'.'` 。
- 给定数独永远是 `9x9` 形式的。

思路

一次迭代：

- 如何枚举子数独?

> 可以使用 `box_index = (row / 3) * 3 + columns / 3`，其中 `/` 是整数除法。

- 如何确保行 / 列 / 子数独中没有重复项?

> 可以利用 `value -> count` 哈希映射来跟踪所有已经遇到的值。

Solution

```
class Solution {
public:
    bool isValidSudoku(vector<vector<char>>& board) {
        bool flag = true;
        int row[9][10];//第一维为行序号
        int column[9][10];//第一维为列序号
        int sub[9][10]; //第一维为子块序号，定义为“(行/3)*3+列/3”
        for (int i = 0 ; i < 9; i++) {
            for(int j = 0 ; j < 10; j++) {
                row[i][j] = 0;
                column[i][j] = 0;
                sub[i][j] = 0;
            }
        }
```

```
        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {

                if(board[i][j] != '.') {
                    int true_num = board[i][j] - '0';
                    row[i][true_num]++;
                    column[j][true_num]++;
                    sub[(i/3)*3 + j/3][true_num]++;
                }
            }
        }

        for(int i = 0; i < 9 ; i++) {
            for(int j = 0; j < 10 ; j++) {
                if(row[i][j] >= 2 || column[i][j] >= 2 || sub[i][j] >= 2) {
                    flag =  false;
                }
            }
        }
        return flag;
    }
};
```

1.21 报数

报数序列是一个整数序列，按照其中的整数的顺序进行报数，得到下一个数。其前五项如下：

```
1.     1
2.     11
3.     21
4.     1211
5.     111221
```

`1` 被读作 `"one 1"` (`"一个一"`), 即 `11`。 `11` 被读作 `"two 1s"` (`"两个一"`), 即 `21`。 `21` 被读作 `"one 2"`, `"one 1"` (`"一个二"`, `"一个一"`), 即 `1211`。

给定一个正整数 *n*（1 ≤ *n* ≤ 30），输出报数序列的第 *n* 项。

注意：整数顺序将表示为一个字符串。

Solution

```
class Solution {
public:
```

```
    string countAndSay(int n) {
        string a="1",temp="";
        for(int i=1;i<n;i++)
        {
            int t=0;
            while(t<a.size())
            {
                int value=1;
                while(a[t]==a[t+1])//&&t+1<a.size()
                {
                    value++;
                    t++;
                }
                temp=temp+to_string(value)+a[t];//char可以直接，不用to_string
                t++;
            }
            a=temp;
            temp="";
        }
        return a;
    }
};
```

1.22 复数乘法

给定两个表示复数的字符串。

返回表示它们乘积的字符串。注意，根据定义 i2 = -1 。

示例 1:

输入: "1+1i", "1+1i" 输出: "0+2i" 解释: (1 + i) * (1 + i) = 1 + i2 + 2 * i = 2i ，你需要将它转换为 0+2i 的形式。 示例 2:

输入: "1+-1i", "1+-1i" 输出: "0+-2i" 解释: (1 - i) * (1 - i) = 1 + i2 - 2 * i = -2i ，你需要将它转换为 0+-2i 的形式。 注意:

输入字符串不包含额外的空格。 输入字符串将以 a+bi 的形式给出，其中整数 a 和 b 的范围均在 [-100, 100] 之间。输出也应当符合这种形式。

思路

两个复数的乘法可以依下述方法完成：(a+ib)×(x+iy)=ax+i2by+i(bx+ay)=ax−by+i(bx+ay)
(a+ib)×(x+iy)=ax+i 2 by+i(bx+ay)=ax−by+i(bx+ay)

我们简单地根据 '+' 和 'i' 符号分割给定的复杂字符串的实部和虚部。我们把 a 和 b两个字符串的实部分别存储为 x[0]和 y[0]，虚部分别用 x[1] 和 y[1] 存储。然后，将提取的部分转换为整数后，根据需要将实部和虚部相乘。然后，我们再次以所需的格式形成返回字符串，并返回结果。

Solution

```
public class Solution {

    public String complexNumberMultiply(String a, String b) {
        String x[] = a.split("\\+|i");
        String y[] = b.split("\\+|i");
        int a_real = Integer.parseInt(x[0]);
        int a_img = Integer.parseInt(x[1]);
        int b_real = Integer.parseInt(y[0]);
        int b_img = Integer.parseInt(y[1]);
        return (a_real * b_real - a_img * b_img) + "+" + (a_real * b_img +
a_img * b_real) + "i";

    }
}
```

1.22 石子游戏

亚历克斯和李用几堆石子在做游戏。偶数堆石子**排成一行**，每堆都有正整数颗石子 `piles[i]` 。

游戏以谁手中的石子最多来决出胜负。石子的总数是奇数，所以没有平局。

亚历克斯和李轮流进行，亚历克斯先开始。 每回合，玩家从行的开始或结束处取走整堆石头。 这种情况一直持续到没有更多的石子堆为止，此时手中石子最多的玩家获胜。

假设亚历克斯和李都发挥出最佳水平，当亚历克斯赢得比赛时返回 `true` ，当李赢得比赛时返回 `false` 。

示例：

```
输入：[5,3,4,5]
输出：true
解释：
亚历克斯先开始，只能拿前 5 颗或后 5 颗石子 。
假设他取了前 5 颗，这一行就变成了 [3,4,5] 。
如果李拿走前 3 颗，那么剩下的是 [4,5]，亚历克斯拿走后 5 颗赢得 10 分。
如果李拿走后 5 颗，那么剩下的是 [3,4]，亚历克斯拿走后 4 颗赢得 9 分。
这表明，取前 5 颗石子对亚历克斯来说是一个胜利的举动，所以我们返回 true 。
```

提示：

1. `2 <= piles.length <= 500`
2. `piles.length` 是偶数。
3. `1 <= piles[i] <= 500`
4. `sum(piles)` 是奇数。

思路

方法一：动态规划

思路

让我们改变游戏规则，使得每当李得分时，都会从亚历克斯的分数中扣除。

令 `dp(i, j)` 为亚历克斯可以获得的最大分数，其中剩下的堆中的石子数是 `piles[i], piles[i+1], ..., piles[j]`。这在比分游戏中很自然：我们想知道游戏中每个位置的值。

我们可以根据 `dp(i + 1, j)` 和 `dp(i, j-1)` 来制定 `dp(i, j)` 的递归，我们可以使用动态编程以不重复这个递归中的工作。（该方法可以输出正确的答案，因为状态形成一个DAG（有向无环图）。）

算法

当剩下的堆的石子数是 `piles[i], piles[i+1], ..., piles[j]` 时，轮到的玩家最多有 2 种行为。

可以通过比较 `j-i` 和 `N modulo 2` 来找出轮到的人。

如果玩家是亚历克斯，那么她将取走 `piles[i]` 或 `piles[j]` 颗石子，增加她的分数。之后，总分为 `piles[i] + dp(i+1, j)` 或 `piles[j] + dp(i, j-1)`；我们想要其中的最大可能得分。

如果玩家是李，那么他将取走 `piles[i]` 或 `piles[j]` 颗石子，减少亚历克斯这一数量的分数。之后，总分为 `-piles[i] + dp(i+1, j)` 或 `-piles[j] + dp(i, j-1)`；我们想要其中的*最小*可能得分。

Solution

```
class Solution {
public:
    bool stoneGame(vector<int>& piles) {
        int N = piles.size();

        // dp[i+1][j+1] = the value of the game [piles[i], ..., piles[j]]
        int dp[N+2][N+2];
        memset(dp, 0, sizeof(dp));

        for (int size = 1; size <= N; ++size)
            for (int i = 0, j = size - 1; j < N; ++i, ++j) {
                int parity = (j + i + N) % 2;  // j - i - N; but +x = -x (mod 2)
                if (parity == 1)
                    dp[i+1][j+1] = max(piles[i] + dp[i+2][j+1], piles[j] + dp[i+1][j]);
                else
```

```
                    dp[i+1][j+1] = min(-piles[i] + dp[i+2][j+1], -piles[j]
+ dp[i+1][j]);
            }

        return dp[1][N] > 0;
    }
};
```

1.22 螺旋矩阵

给定一个正整数 *n*，生成一个包含 1 到 *n*2 所有元素，且元素按顺时针顺序螺旋排列的正方形矩阵。

示例:

```
输入: 3
输出:
[
 [ 1, 2, 3 ],
 [ 8, 9, 4 ],
 [ 7, 6, 5 ]
]
```

思路

类似于蛇形迷宫解题思路，把握好边界条件即可！将填写数字的过程看成一条蛇在螺旋走位。

Solution

```
class Solution {
public:
    vector<vector<int>> generateMatrix(int n) {
        int s = 0;
        int e = n - 1;
        int num = 1;
        vector<vector<int> > res(n, vector<int>(n, 0)); // 必须初始化
        while(s < e) {
            for(int j = s; j <= e; j++) res[s][j] = num++;
            for(int i = s + 1; i <= e; i++) res[i][e] = num++;
            for(int j = e - 1; j >= s; j--) res[e][j] = num++;
            for(int i = e - 1; i > s; i--) res[i][s] = num++;
            ++s;
            --e;
        }
        if(s == e) res[s][s] = num;
        return res;
    }
};
```

1.22 煎饼排序

给定数组 `A`，我们可以对其进行*煎饼翻转*：我们选择一些正整数 `**k** <= A.length`，然后反转 `A` 的前 **k** 个元素的顺序。我们要执行零次或多次煎饼翻转（按顺序一次接一次地进行）以完成对数组 `A` 的排序。

返回能使 `A` 排序的煎饼翻转操作所对应的 k 值序列。任何将数组排序且翻转次数在 `10 * A.length` 范围内的有效答案都将被判断为正确。

示例 1：

```
输入：[3,2,4,1]
输出：[4,2,4,3]
解释：
我们执行 4 次煎饼翻转，k 值分别为 4，2，4，和 3。
初始状态 A = [3, 2, 4, 1]
第一次翻转后 (k=4)：A = [1, 4, 2, 3]
第二次翻转后 (k=2)：A = [4, 1, 2, 3]
第三次翻转后 (k=4)：A = [3, 2, 1, 4]
第四次翻转后 (k=3)：A = [1, 2, 3, 4]，此时已完成排序。
```

示例 2：

```
输入：[1,2,3]
输出：[]
解释：
输入已经排序，因此不需要翻转任何内容。
请注意，其他可能的答案，如[3，3]，也将被接受。
```

提示：

1. `1 <= A.length <= 100`
2. `A[i]` 是 `[1, 2, ..., A.length]` 的排列

思路

方法一：从大到小排序

思路

我们可以将最大的元素（在位置 `i`，下标从 1 开始）进行煎饼翻转 `i` 操作将它移动到序列的最前面，然后再使用煎饼翻转 `A.length` 操作将它移动到序列的最后面。 此时，最大的元素已经移动到正确的位置上了，所以之后我们就不需要再使用 `k` 值大于等于 `A.length` 的煎饼翻转操作了。

我们可以重复这个过程直到序列被排好序为止。 每一步，我们只需要花费两次煎饼翻转操作。

算法

我们从数组 A 中的最大值向最小值依次进行枚举，每一次将枚举的元素放到正确的位置上。

每一步，对于在位置 `i` 的（从大到小枚举的）元素，我们会使用思路中提到的**煎饼翻转组合操作**将它移动到正确的位置上。 值得注意的是，执行一次煎饼翻转操作 `f`，会将位置在 `i, i <= f` 的元素翻转到位置 `f+1 - i` 上。

Solution

```
class Solution {
    public List<Integer> pancakeSort(int[] A) {
        List<Integer> ans = new ArrayList();
        int N = A.length;

        Integer[] B = new Integer[N];
        for (int i = 0; i < N; ++i)
            B[i] = i+1;
        Arrays.sort(B, (i, j) -> A[j-1] - A[i-1]);

        for (int i: B) {
            for (int f: ans)
                if (i <= f)
                    i = f+1 - i;
            ans.add(i);
            ans.add(N--);
        }

        return ans;
    }
}
```

1.25 翻转矩阵后的得分

- 有一个二维矩阵 `A` 其中每个元素的值为 `0` 或 `1` 。

  移动是指选择任一行或列，并转换该行或列中的每一个值：将所有 `0` 都更改为 `1`，将所有 `1` 都更改为 `0`。

  在做出任意次数的移动后，将该矩阵的每一行都按照二进制数来解释，矩阵的得分就是这些数字的总和。

  返回尽可能高的分数。

示例：

```
输入：[[0,0,1,1],[1,0,1,0],[1,1,0,0]]
输出：39
解释：
转换为 [[1,1,1,1],[1,0,0,1],[1,1,1,1]]
0b1111 + 0b1001 + 0b1111 = 15 + 9 + 15 = 39
```

提示：

1. `1 <= A.length <= 20`
2. `1 <= A[0].length <= 20`
3. `A[i][j]` 是 `0` 或 `1`

思路

> 1. 返回尽可能高分这个要求，理解为对同一组数，高位尽可能置1，对不同组的相同位尽可能多的置1。
> 2. 1，判断最高位是否为1，否，移动当前行。
>
>   2，判断每列的的0的个数，如果0较多，移动当前列。
>
>   3，输出最高分数。

Solution

```
class Solution {
public:
    int matrixScore(vector<vector<int>>& A) {
        int row = A.size();
        int cloumn = A[0].size();
        for (int i = 0; i < row; i++)
        {
            if(A[i][0]==0)
            {
                for (int j = 0; j < cloumn; j++)
                {
                    A[i][j] = 1 - A[i][j];
                }
            }
        }
        for (int j = 0; j < cloumn; j++)
        {
            int coutZero = 0, coutOne = 0;
            for (int i = 0; i < row; i++)
            {
                if (A[i][j] == 0)
                {
```

```
                    coutZero++;
                }
                if (A[i][j] == 1)
                {
                    coutOne++;
                }
            }
            if(coutZero>coutOne)
            {
                for (int m = 0; m < row; m++)
                {
                    A[m][j] = 1 - A[m][j];
                }
            }
        }

        int sum = 0;
        for (int i = 0; i < row; i++)
        {
            for (int j = 0; j < cloumn; j++)
            {
                sum += A[i][j] * pow(2, (cloumn - j - 1));
            }
        }
        return sum;
    }
};
```

1.25 二叉树剪枝

给定二叉树根结点 `root` ，此外树的每个结点的值要么是 0，要么是 1。

返回移除了所有不包含 1 的子树的原二叉树。

(节点 X 的子树为 X 本身，以及所有 X 的后代。)

思路

与二叉树有关的问题，首先要想到“递归”，注意递归的终止条件。

关键是从底往上剪,也就是从叶子节点,如果叶子节点为0,就None

Solution

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

//递归即可
class Solution {
public:
    TreeNode* pruneTree(TreeNode* root) {
        if(root == NULL) {
            return NULL;
        }
        if(root->left == NULL && root->right == NULL) {
            if(root->val == 0) {
                return NULL;
            } else {
                return root;
            }
        }
        root->left = pruneTree(root->left);
        root->right = pruneTree(root->right);
        return root->left == NULL && root->right == NULL && root->val == 0
? NULL:root;
    }
};
```

1.27 下一个全排列

实现获取下一个排列的函数，算法需要将给定数字序列重新排列成字典序中下一个更大的排列。

如果不存在下一个更大的排列，则将数字重新排列成最小的排列（即升序排列）。

必须**原地**修改，只允许使用额外常数空间。

以下是一些例子，输入位于左侧列，其相应输出位于右侧列。 `1,2,3` → `1,3,2` `3,2,1` → `1,2,3` `1,1,5` → `1,5,1`

思路

我们需要找到给定数字列表的下一个字典排列，而不是由给定数组形成的数字。

- 方法一：暴力求解
    - 在这种方法中，我们找出由给定数组的元素形成的列表的每个可能的排列，并找出比给定的排列更大的排列。 但是这个方法是一种非常天真的方法，因为它要求我们找出所有可能

的排列 这需要很长时间，实施起来也很复杂。 因此，这种方法根本无法通过。 所以，我们直接采用正确的方法。

- 方法二：一遍扫描
  - **算法**

    首先，我们观察到对于任何给定序列的降序，没有可能的下一个更大的排列。

    例如，以下数组不可能有下一个排列：

    ```
    [9, 5, 4, 3, 1]
    ```

    我们需要从右边找到第一对两个连续的数字 a[i] 和 a[i−1]，它们满足 a[i]>a[i−1]。现在，没有对 a[i−1]右侧的重新排列可以创建更大的排列，因为该子数组由数字按降序组成。因此，我们需要重新排列 a[i−1] 右边的数字，包括它自己。

    现在，什么样的重新排列将产生下一个更大的数字？我们想要创建比当前更大的排列。因此，我们需要将数字 a[i−1]替换为位于其右侧区域的数字中比它更大的数字，例如 a[j]a[j]。

    我们交换数字 a[i−1] 和 a[j]。我们现在在索引 i−1处有正确的数字。 但目前的排列仍然不是我们正在寻找的排列。我们需要通过仅使用 a[i−1]右边的数字来形成最小的排列。 因此，我们需要放置那些按升序排列的数字，以获得最小的排列。

    但是，请记住，在从右侧扫描数字时，我们只是继续递减索引直到我们找到 a[i]和 a[i−1]这对数。其中，a[i]>a[i−1]。因此，a[i−1]右边的所有数字都已按降序排序。此外，交换 a[i−1]和 a[j] 并未改变该顺序。因此，我们只需要反转 a[i−1]之后的数字，以获得下一个最小的字典排列。

Solution

```
public class Solution {
    public void nextPermutation(int[] nums) {
        int i = nums.length - 2;
        while (i >= 0 && nums[i + 1] <= nums[i]) {
            i--;
        }
        if (i >= 0) {
            int j = nums.length - 1;
            while (j >= 0 && nums[j] <= nums[i]) {
                j--;
            }
            swap(nums, i, j);
        }
        reverse(nums, i + 1);
    }

    private void reverse(int[] nums, int start) {
        int i = start, j = nums.length - 1;
        while (i < j) {
            swap(nums, i, j);
```

```
            i++;
            j--;
        }
    }

    private void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

1.27 无重复的全排列

给定一个**没有重复**数字的序列，返回其所有可能的全排列。

示例:

```
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]
]
```

思路

这是很经典的全排列问题，本题的解法很多。因为这里的所有数都是相异的，故笔者采用了交换元素+DFS的方法来求解。 **交换法**的思路是for(i = start to end)，循环中: swap (第start个和第i个)，递归调用(start+1)，swap back

Solution

```
class Solution {
public:
    vector<vector<int> > permute(vector<int> &num) {
        if(num.size() == 0) return res;
        permuteCore(num, 0);
        return res;
    }
private:
    vector<vector<int> > res;
```

```
    void permuteCore(vector<int> &num, int start){
        if(start == num.size()){
            vector<int> v;
            for(vector<int>::iterator i = num.begin(); i < num.end(); ++i){
                v.push_back(*i);
            }
            res.push_back(v);
        }
        for(int i = start; i < num.size(); ++i){
            swap(num, start, i);
            permuteCore(num, start+1);
            swap(num, start, i);
        }
    }
    void swap(vector<int> &num, int left, int right){
        int tmp = num[left];
        num[left] = num[right];
        num[right] = tmp;
    }
};
```

1.27 有重复的全排列

给定一个可包含重复数字的序列，返回所有不重复的全排列。

示例:

```
输入: [1,1,2]
输出:
[
  [1,1,2],
  [1,2,1],
  [2,1,1]
]
```

思路

本题与上一题略有不同，给定的序列中含有重复元素，需要返回这些数所能排列出的所有不同的序列集合。这里我们先考虑一下，它与第二题唯一的不同在于：在DFS函数中，做循环遍历时，如果与当前元素相同的一个元素已经被取用过，则要跳过所有值相同的元素。 举个例子：对于序列<1,1,2,3>。在DFS首遍历时，1 作为首元素被加到list中，并进行后续元素的添加；那么，当DFS跑完第一个分支，遍历到1 (第二个)时，这个1 不再作为首元素添加到list中，因为1 作为首元素的情况已经在第一个分支中考虑过了。为了实现这一剪枝思路，有了如下的解题算法。

1. 先对给定的序列nums进行排序，使得大小相同的元素排在一起。

1. 新建一个used数组，大小与nums相同，用来标记在本次DFS读取中，位置i的元素是否已经被添加到list中了。
2. 根据思路可知，我们选择跳过一个数，当且仅当这个数与前一个数相等，并且前一个数未被添加到list中。 根据以上算法，对题二的代码略做修改，可以得到如下的AC代码。

   (在处理一般性问题时，建议用此算法，毕竟题二只是特殊情况)

本题目还可以借用第一题中的下一个排列的接口来实现，避免了递归，更加简单。

Solution

```
class Solution {
public:
    vector<vector<int> > permuteUnique(vector<int> &num) {
        if(num.size() <= 0) return res;
        sort(num.begin(), num.end());
        res.push_back(num);
        int i = 0, j = 0;
        while(1){
            //Calculate next permutation
            for(i = num.size()-2; i >= 0 && num[i] >= num[i+1]; --i);
            if(i < 0) break;
            for(j = num.size()-1; j > i && num[j] <= num[i]; --j);
            swap(num, i, j);
            j = num.size()-1;
            ++i;
            while(i < j)
                swap(num, i++, j--);
            //push next permutation
            res.push_back(num);
        }
        return res;
    }
private:
    vector<vector<int> > res;
    void swap(vector<int> &num, int left, int right){
        int tmp = num[left];
        num[left] = num[right];
        num[right] = tmp;
    }
};
```

1.27 取特定位置的全排列字符串(第k个排列)

给出集合 `[1,2,3,…,*n*]`，其所有元素共有 *n*! 种排列。

按大小顺序列出所有排列情况，并一一标记，当 *n* = 3 时, 所有排列如下：

1. `"123"`
2. `"132"`
3. `"213"`
4. `"231"`
5. `"312"`
6. `"321"`

给定 *n* 和 *k*，返回第 *k* 个排列。

说明：

- 给定 *n* 的范围是 [1, 9]。
- 给定 *k* 的范围是[1, *n*!]。

示例 1:

```
输入: n = 3, k = 3
输出: "213"
```

示例 2:

```
输入: n = 4, k = 9
输出: "2314"
```

Solution

> 连续k次调用第一个题目中的接口速度太慢

```
class Solution {

    public String getPermutation(int n, int k) {
        /**
        直接用回溯法做的话需要在回溯到第k个排列时终止就不会超时了, 但是效率依旧感人
        可以用数学的方法来解, 因为数字都是从1开始的连续自然数, 排列出现的次序可以推
        算出来, 对于n=4, k=15 找到k=15排列的过程:

        1 + 对2,3,4的全排列 (3!个)
        2 + 对1,3,4的全排列 (3!个)          3, 1 + 对2,4的全排列(2!个)
        3 + 对1,2,4的全排列 (3!个)-------> 3, 2 + 对1,4的全排列(2!个)-------> 
3, 2, 1 + 对4的全排列(1!个)-------> 3214
        4 + 对1,2,3的全排列 (3!个)          3, 4 + 对1,2的全排列(2!个)
3, 2, 4 + 对1的全排列(1!个)

        确定第一位:
            k = 14(从0开始计数)
            index = k / (n-1)! = 2, 说明第15个数的第一位是3
            更新k
            k = k - index*(n-1)! = 2
        确定第二位:
```

```
            k = 2
            index = k / (n-2)! = 1, 说明第15个数的第二位是2
            更新k
            k = k - index*(n-2)! = 0
        确定第三位:
            k = 0
            index = k / (n-3)! = 0, 说明第15个数的第三位是1
            更新k
            k = k - index*(n-3)! = 0
        确定第四位:
            k = 0
            index = k / (n-4)! = 0, 说明第15个数的第四位是4
        最终确定n=4时第15个数为3214
        **/

        StringBuilder sb = new StringBuilder();
        // 候选数字
        List<Integer> candidates = new ArrayList<>();
        // 分母的阶乘数
        int[] factorials = new int[n+1];
        factorials[0] = 1;
        int fact = 1;
        for(int i = 1; i <= n; ++i) {
            candidates.add(i);
            fact *= i;
            factorials[i] = fact;
        }
        k -= 1;
        for(int i = n-1; i >= 0; --i) {
            // 计算候选数字的index
            int index = k / factorials[i];
            sb.append(candidates.remove(index));
            k -= index*factorials[i];
        }
        return sb.toString();
    }
}
```

1.28 所有可能的满二叉树

- *满二叉树*是一类二叉树，其中每个结点恰好有 0 或 2 个子结点。

  返回包含 `N` 个结点的所有可能满二叉树的列表。 答案的每个元素都是一个可能树的根结点。

  答案中每个树的每个 `结点` 都**必须**有 `node.val=0` 。

  你可以按任何顺序返回树的最终列表。

思路

方法：递归

思路与算法

令 FBT(N) 作为所有含 N 个结点的可能的满二叉树的列表。

每个满二叉树 T 含有 3 个或更多结点，在其根结点处有 2 个子结点。这些子结点 `left` 和 `right` 本身就是满二叉树。

因此，对于 N≥3，我们可以设定如下的递归策略：FBT(N)= [对于所有的 x，所有的树的左子结点来自 FBT(x)而右子结点来自 FBT(N−1−x)]。

此外，通过简单的计数参数，没有满二叉树具有正偶数个结点。

最后，我们应该缓存函数 FBT之前的结果，这样我们就不必在递归中重新计算它们。

Solution

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<TreeNode*> allPossibleFBT(int N) {
        vector<vector<TreeNode*>> dp(N+1);
        dp[1].push_back({new TreeNode(0)});

        for(int i = 3; i <= N; i+=2)
            for(int l = 1; l <= i - 2; l+=2){
                int r = i - 1 - l;

                for(auto& ltree: dp[l]){
                    for(auto& rtree: dp[r]){
                        TreeNode* root = new TreeNode(0);
                        root->left = ltree;
                        root->right = rtree;
                        dp[i].push_back(root);
                    }

                }
            }
        return dp[N];
    }
```

```
};
```

1.28 组合

给定两个整数 *n* 和 *k*，返回 1 ... *n* 中所有可能的 *k* 个数的组合。

示例:

```
输入: n = 4, k = 2
输出:
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

思路

这道题让求1到n共n个数字里k个数的组合数的所有情况，还是要用深度优先搜索DFS来解，根据以往的经验，像这种要求出所有结果的集合，一般都是用DFS调用递归来解。那么我们建立一个保存最终结果的大集合res，还要定义一个保存每一个组合的小集合out，每次放一个数到out里，如果out里数个数到了k个，则把out保存到最终结果中，否则在下一层中继续调用递归。

Solution

```
class Solution {
public:
    vector<vector<int>> combine(int n, int k) {
        vector<vector<int>> res;
        vector<int> out;
        helper(n, k, 1, out, res);
        return res;
    }
    void helper(int n, int k, int level, vector<int>& out,
vector<vector<int>>& res) {
        if (out.size() == k) {res.push_back(out); return;}
        for (int i = level; i <= n; ++i) {
            out.push_back(i);
            helper(n, k, i + 1, out, res);
            out.pop_back();
        }
    }
};
```

1.30 为运算表达式设计优先级

- 给定一个含有数字和运算符的字符串，为表达式添加括号，改变其运算优先级以求出不同的结果。你需要给出所有可能的组合的结果。有效的运算符号包含 `+`, `-` 以及 `*` 。

  示例 1:

  ```
  输入: "2-1-1"
  输出: [0, 2]
  解释:
  ((2-1)-1) = 0
  (2-(1-1)) = 2
  ```

  示例 2:

  ```
  输入: "2*3-4*5"
  输出: [-34, -14, -10, -10, 10]
  解释:
  (2*(3-(4*5))) = -34
  ((2*3)-(4*5)) = -14
  ((2*(3-4))*5) = -10
  (2*((3-4)*5)) = -10
  (((2*3)-4)*5) = 10
  ```

思路

这道题是典型的利用递归实现分而治之的题目。

基本思想是：若要求解一个字符串加上不同括号产生的所有结果，先对输入的字符串进行for循环遍历，若遇到'+'、'-'、'*'这三个运算符，则记录当前的运算符。然后分别对运算符左边和右边的子字符串递归，求解子字符串加上不同括号产生的所有结果。即是下面的两条语句：

lResult = diffWaysToCompute(input.substr(0, i)); rResult = diffWaysToCompute(input.substr(i + 1, len - i - 1));

最后，在计算完子字符串的结果后，根据之前记录的运算符，算出最终结果。

这个递归方法的结束条件是：当传入的字符串中不含任何运算符时，将该字符串转化为数字，然后将该数字push入Vector容器数组并返回。我用了一个bool变量flag来判断字符串中是否含运算符。

因为一个字符串加上不同括号会产生不止一个结果，所以diffWaysToCompute函数返回的是Vector容器数组，该Vector容器数组包含了该字符串可以产生的所有结果。所以利用左右子字符串产生的结果来计算最终结果时，是利用了两个嵌套的for循环

Solution

```
class Solution {
public:
    vector<int> diffWaysToCompute(string input) {
```

```
        vector<int> lResult;
        vector<int> rResult;
        vector<int> result;
        int len = input.size();
        bool flag = false;
        char op;

        if (len == 0) return result;

        for (int i = 0; i < len; i++) {
            switch (input[i]) {
                case '+':
                case '-':
                case '*':
                    flag = true;
                    op = input[i];
                    lResult = diffWaysToCompute(input.substr(0, i));
                    rResult = diffWaysToCompute(input.substr(i + 1, len - i
- 1));
                    for (int j = 0; j < lResult.size(); j++) {
                        for (int k = 0; k < rResult.size(); k++) {
                            switch (op) {
                            case '+':
                                result.push_back(lResult[j] + rResult[k]);
                                break;
                            case '-':
                                result.push_back(lResult[j] - rResult[k]);
                                break;
                            case '*':
                                result.push_back(lResult[j] * rResult[k]);
                                break;
                            default:
                                break;
                            }
                        }
                    }
                    break;
                default:
                    break;
            }
        }

        if (flag == false) {
            result.push_back(atoi(input.data()));
        }

        return result;
    }
```

```
};
```

1.30 划分字母区间

- 字符串 `S` 由小写字母组成。我们要把这个字符串划分为尽可能多的片段，同一个字母只会出现在其中的一个片段。返回一个表示每个字符串片段的长度的列表。

  示例 1:

  ```
  输入: S = "ababcbacadefegdehijhklij"
  输出: [9,7,8]
  解释:
  划分结果为 "ababcbaca", "defegde", "hijhklij"。
  每个字母最多出现在一个片段中。
  像 "ababcbacadefegde", "hijhklij" 的划分是错误的，因为划分的片段数较少。
  ```

  注意:

  1. `S` 的长度在 `[1, 500]` 之间。
  2. `S` 只包含小写字母 `'a'` 到 `'z'` 。

思路

> 遍历字符串 找到和起点相同的最后一个字母 查看此区间里的字母最后的index是否超出区间 超出则更新区间 直至找到最大的index 则index - i + 1就是所求区间长度 使用cache来存储每个字母的最后出现位置

Solution

```java
class Solution {
    public List<Integer> partitionLabels(String S) {
        if (S == null || S.length() == 0) {
            return null;
        }

        List<Integer> res = new ArrayList<>();
        int index, i, len = S.length();
        int[] cache = new int[26];
        for (i = 0; i < len; i++) {
            cache[S.charAt(i) - 'a'] = i;
        }
        i = 0;
        while (i < len) {
            index = cache[S.charAt(i) - 'a'];
            for (int j = i + 1; j < index && j < len; j++) {
```

```
                if (cache[S.charAt(j) - 'a'] > index) {
                    index = cache[S.charAt(j) - 'a'];
                }
            }
            res.add(index - i + 1);
            i = index + 1;
        }
        return res;
    }
}
```

1.31 查找和替换模式

- 你有一个单词列表 `words` 和一个模式 `pattern`，你想知道 `words` 中的哪些单词与模式匹配。

  如果存在字母的排列 `p` ，使得将模式中的每个字母 `x` 替换为 `p(x)` 之后，我们就得到了所需的单词，那么单词与模式是匹配的。

  （回想一下，字母的排列是从字母到字母的双射：每个字母映射到另一个字母，没有两个字母映射到同一个字母。）

  返回 `words` 中与给定模式匹配的单词列表。

  你可以按任何顺序返回答案。

  示例：

  ```
  输入：words = ["abc","deq","mee","aqq","dkd","ccc"], pattern = "abb"
  输出：["mee","aqq"]
  解释：
  "mee" 与模式匹配，因为存在排列 {a -> m, b -> e, ...}。
  "ccc" 与模式不匹配，因为 {a -> c, b -> c, ...} 不是排列。
  因为 a 和 b 映射到同一个字母。
  ```

  提示：

  - `1 <= words.length <= 50`
  - `1 <= pattern.length = words[i].length <= 20`

思路

构造一个hashmap，遍历每一个字符串，将该字符串中的每一个字符都和pattern中的每一个字符进行一一对应，例如，第一个"abc"，则遍历之后，a对应pattern中的a；第二个"b"遍历后对应pattern中的第二个"b"；但是第三个"c"和pattern中的"b"不对应，所以该字符不能压入list中。所以hashmap以pattern中的每个字符为key，以字符串中的每个字符为value，进行遍历。但是这里需要特别注意的地方是凡是pattern中不一样的字符，所对应的字符串中的每一个字符必须不一样，哪怕对应到字符串中的字符没有在hashmap中出现过，这需要单独判断！！！

Solution

```
class Solution {
        /**
     * 查找和替换模式
     * @param words
     * @param pattern
     * @return
     */
    public List<String> findAndReplacePattern(String[] words, String
pattern) {
        List<String> result = new ArrayList<>();
        char[] patternChars = pattern.toCharArray();
        for (String word : words) {
            char[] wordChars = word.toCharArray();
            //是否匹配
            boolean match = true;
            //用于存储映射关系的Map
            Map<Character, Character> map = new HashMap<>();
            for (int i = 0; i < patternChars.length; i++) {
                char p = patternChars[i];
                char w = wordChars[i];
                if (map.containsKey(p)) {//模式中的字母是否已经映射已经被映射
                    char value = map.get(p);
                    if (value != w) {
                        match = false;
                        break;
                    }
                } else {
                    if(map.containsValue(w)){//判断单词中的字母是否被映射，防止
模式多个字母都映射单词中的同一个字母
                        match = false;
                        break;
                    }else {
                        map.put(p, w);
                    }
                }
            }
            if (match) {
                result.add(word);
            }
        }
        return result;
    }
}
```

1.21 顶端迭代器

给定一个迭代器类的接口，接口包含两个方法： `next()` 和 `hasNext()` 。设计并实现一个支持 `peek()` 操作的顶端迭代器 -- 其本质就是把原本应由 `next()` 方法返回的元素 `peek()` 出来。

示例:

```
假设迭代器被初始化为列表 [1,2,3]。

调用 next() 返回 1，得到列表中的第一个元素。
现在调用 peek() 返回 2，下一个元素。在此之后调用 next() 仍然返回 2。
最后一次调用 next() 返回 3，末尾元素。在此之后调用 hasNext() 应该返回 false。
```

Solution

```
// Java Iterator interface reference:
// https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html
class PeekingIterator implements Iterator<Integer> {

    private Iterator<Integer> mIterator;
    private Integer next;

    public PeekingIterator(Iterator<Integer> iterator) {
        // initialize any member here.
        this.mIterator = iterator;
    }

    // Returns the next element in the iteration without advancing the
iterator.
    public Integer peek() {
        if (next == null && mIterator.hasNext()) {
            next = mIterator.next();
        }
        return next;
    }

    // hasNext() and next() should behave the same as in the Iterator
interface.
    // Override them if needed.
    @Override
    public Integer next() {
        if (next == null) {
            return mIterator.next();
        } else {
            Integer temp = next;
            next = null;
```

```
                return temp;
            }
        }

        @Override
        public boolean hasNext() {
            return next != null || mIterator.hasNext();
        }
    }
```

2.1 在二叉树中分配硬币

- 给定一个有 `N` 个结点的二叉树的根结点 `root`，树中的每个结点上都对应有 `node.val` 枚硬币，并且总共有 `N` 枚硬币。

  在一次移动中，我们可以选择两个相邻的结点，然后将一枚硬币从其中一个结点移动到另一个结点。(移动可以是从父结点到子结点，或者从子结点移动到父结点。)。

  返回使每个结点上只有一枚硬币所需的移动次数。

思路

方法一：深度优先遍历

思路

如果树的叶子仅包含 0 枚金币（与它所需相比，它的 `过载量` 为 -1），那么我们需要从它的父亲节点移动一枚金币到这个叶子节点上。如果说，一个叶子节点包含 4 枚金币（它的 `过载量` 为 3），那么我们需要将这个叶子节点中的 3 枚金币移动到别的地方去。总的来说，对于一个叶子节点，需要移动到它中或需要从它移动到它的父亲中的金币数量为 `过载量 = Math.abs(num_coins - 1)`。然后，在接下来的计算中，我们就再也不需要考虑这些已经考虑过的叶子节点了。

算法

我们可以用上述的方法来逐步构建我们的最终答案。定义 `dfs(node)` 为这个节点所在的子树中金币的 *过载量*，也就是这个子树中金币的数量减去这个子树中节点的数量。接着，我们可以计算出这个节点与它的子节点之间需要移动金币的数量为 `abs(dfs(node.left)) + abs(dfs(node.right))`，这个节点金币的过载量为 `node.val + dfs(node.left) + dfs(node.right) - 1`。

Solution

```
class Solution {
    int ans;
    public int distributeCoins(TreeNode root) {
        ans = 0;
        dfs(root);
        return ans;
```

```
    }

    public int dfs(TreeNode node) {
        if (node == null) return 0;
        int L = dfs(node.left);
        int R = dfs(node.right);
        ans += Math.abs(L) + Math.abs(R);
        return node.val + L + R - 1;
    }
}
```

2.1 煎饼排序

给定数组 `A`，我们可以对其进行*煎饼翻转*：我们选择一些正整数 `k <= A.length`，然后反转 `A` 的前 **k** 个元素的顺序。我们要执行零次或多次煎饼翻转（按顺序一次接一次地进行）以完成对数组 `A` 的排序。

返回能使 `A` 排序的煎饼翻转操作所对应的 k 值序列。任何将数组排序且翻转次数在 `10 * A.length` 范围内的有效答案都将被判断为正确。

示例 1：

```
输入：[3,2,4,1]
输出：[4,2,4,3]
解释：
我们执行 4 次煎饼翻转，k 值分别为 4，2，4，和 3。
初始状态 A = [3, 2, 4, 1]
第一次翻转后 (k=4)：A = [1, 4, 2, 3]
第二次翻转后 (k=2)：A = [4, 1, 2, 3]
第三次翻转后 (k=4)：A = [3, 2, 1, 4]
第四次翻转后 (k=3)：A = [1, 2, 3, 4]，此时已完成排序。
```

示例 2：

```
输入：[1,2,3]
输出：[]
解释：
输入已经排序，因此不需要翻转任何内容。
请注意，其他可能的答案，如[3，3]，也将被接受。
```

提示：

1. `1 <= A.length <= 100`
2. `A[i]` 是 `[1, 2, ..., A.length]` 的排列

思路

方法一：从大到小排序

思路

我们可以将最大的元素（在位置 `i`，下标从 1 开始）进行煎饼翻转 `i` 操作将它移动到序列的最前面，然后再使用煎饼翻转 `A.length` 操作将它移动到序列的最后面。 此时，最大的元素已经移动到正确的位置上了，所以之后我们就不需要再使用 `k` 值大于等于 `A.length` 的煎饼翻转操作了。

我们可以重复这个过程直到序列被排好序为止。 每一步，我们只需要花费两次煎饼翻转操作。

算法

我们从数组 A 中的最大值向最小值依次进行枚举，每一次将枚举的元素放到正确的位置上。

每一步，对于在位置 `i` 的（从大到小枚举的）元素，我们会使用思路中提到的**煎饼翻转组合操作**将它移动到正确的位置上。 值得注意的是，执行一次煎饼翻转操作 `f`，会将位置在 `i, i <= f` 的元素翻转到位置 `f+1 - i` 上。

Solution

```
class Solution {
    public List<Integer> pancakeSort(int[] A) {
        List<Integer> ans = new ArrayList();
        int N = A.length;

        Integer[] B = new Integer[N];
        for (int i = 0; i < N; ++i)
            B[i] = i+1;
        Arrays.sort(B, (i, j) -> A[j-1] - A[i-1]);

        for (int i: B) {
            for (int f: ans)
                if (i <= f)
                    i = f+1 - i;
            ans.add(i);
            ans.add(N--);
        }

        return ans;
    }
}
```

P237 删除链表中的节点

请编写一个函数，使其可以删除某个链表中给定的（非末尾）节点，你将只被给定要求被删除的节点。

现有一个链表 -- head = [4,5,1,9]，它可以表示为:

实例：

```
输入：head = [4,5,1,9], node = 5
输出：[4,1,9]

输入：head = [4,5,1,9], node = 1
输出：[4,5,9]
```

思路

- 较为简单

Solution

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    void deleteNode(ListNode* node) {
        node->val = node->next->val;
        node->next = node->next->next;
    }
};
```

P1021 删除最外层的括号

有效括号字符串为空 ("")、"(" + A + ")" 或 A + B，其中 A 和 B 都是有效的括号字符串，+ 代表字符串的连接。例如，""，"()"，"(())()" 和 "(()(()))" 都是有效的括号字符串。

如果有效字符串 S 非空，且不存在将其拆分为 S = A+B 的方法，我们称其为原语（primitive），其中 A 和 B 都是非空有效括号字符串。

给出一个非空有效字符串 S，考虑将其进行原语化分解，使得：S = P_1 + P_2 + ... + P_k，其中 P_i 是有效括号字符串原语。

对 S 进行原语化分解，删除分解中每个原语字符串的最外层括号，返回 S 。

示例:

```
输入："(()())(())"
输出："()()()"
```

```
解释：
输入字符串为 "(()())(())"，原语化分解得到 "(()())" + "(())"，
删除每个部分中的最外层括号后得到 "()()" + "()" = "()()()"。

输入："(()())(())(()(()))"
输出："()()()()(())"
解释：
输入字符串为 "(()())(())(()(()))"，原语化分解得到 "(()())" + "(())" + "(()
(()))"，
删除每隔部分中的最外层括号后得到 "()()" + "()" + "()(())" = "()()()()(())"。

输入："()()"
输出：""
解释：
输入字符串为 "()()"，原语化分解得到 "()" + "()"，
删除每个部分中的最外层括号后得到 "" + "" = ""。
```

思路

- 最外面的左括号跳过，L初始为1；当右括号数量等于左括号数量的位置就是最外面的右括号

Solution

```
class Solution {
public:
    string removeOuterParentheses(string S) {
        int L=1;int R=0;
        string ans;
        for(int i=1;i<S.size();i++){
            if(S[i]=='(')L++;
            else R++;
            if(R!=L)ans.push_back(S[i]);
            else {
                i++;L=1;R=0;
            }
        }
        return ans;
    }
};
```

P832 翻转图像

给定一个二进制矩阵 A，我们想先水平翻转图像，然后反转图像并返回结果。

水平翻转图片就是将图片的每一行都进行翻转，即逆序。例如，水平翻转 [1, 1, 0] 的结果是 [0, 1, 1]。

反转图片的意思是图片中的 0 全部被 1 替换， 1 全部被 0 替换。例如，反转 [0, 1, 1] 的结果是 [1, 0, 0]。

实例：

```
输入：[[1,1,0],[1,0,1],[0,0,0]]
输出：[[1,0,0],[0,1,0],[1,1,1]]
解释：首先翻转每一行：[[0,1,1],[1,0,1],[0,0,0]]；
     然后反转图片：[[1,0,0],[0,1,0],[1,1,1]]

输入：[[1,1,0,0],[1,0,0,1],[0,1,1,1],[1,0,1,0]]
输出：[[1,1,0,0],[0,1,1,0],[0,0,0,1],[1,0,1,0]]
解释：首先翻转每一行：[[0,0,1,1],[1,0,0,1],[1,1,1,0],[0,1,0,1]]；
     然后反转图片：[[1,1,0,0],[0,1,1,0],[0,0,0,1],[1,0,1,0]]
```

思路

- 暴力求解，直接翻转即可

Solution

```
class Solution
{
public:
    vector<vector<int>> flipAndInvertImage(vector<vector<int>>& A)
    {
        for(auto &vec:A)
        {
            reverse(vec.begin(),vec.end());
            for(auto &elm:vec)
            {
                if(elm==0)
                {
                    elm=1;
                }
                else
                {
                    elm=0;
                }
            }
        }
        return A;
    }
};
```

P617 合并二叉树

给定两个二叉树，想象当你将它们中的一个覆盖到另一个上时，两个二叉树的一些节点便会重叠。

你需要将他们合并为一个新的二叉树。合并的规则是如果两个节点重叠，那么将他们的值相加作为节点合并后的新值，否则不为 NULL 的节点将直接作为新二叉树的节点。

示例:

```
输入:
        Tree 1                     Tree 2
          1                         2
         / \                       / \
        3   2                     1   3
       /                           \   \
      5                             4   7
输出:
合并后的树:
             3
            / \
           4   5
          / \   \
         5   4   7
```

思路

- 递归终止条件：其中一节点为空则返回另一节点，否则递归合并左右子树

Solution

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* mergeTrees(TreeNode* t1, TreeNode* t2) {
        if(t2==NULL)
            return t1;
        if(t1==NULL)
            return t2;
        TreeNode* res=new TreeNode(t1->val+t2->val);
        res->left=mergeTrees(t1->left,t2->left);
        res->right=mergeTrees(t1->right,t2->right);
        return res;
    }
};
```

P461 汉明距离

两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。

给出两个整数 x 和 y，计算它们之间的汉明距离。

注意： 0 ≤ x, y < 2^31.

实例：

```
输入: x = 1, y = 4

输出: 2

解释:
1   (0 0 0 1)
4   (0 1 0 0)
```

思路

- 先将两个数进行异或^（相同为0不同为1）运算
- 再将异或完的数 计算汉明重量，也就是计算有多少个不同的数， 计算共多少个1

Solution

```
class Solution {
    public:
        int hammingDistance(int x, int y) {
        int count = 0;
        y = x ^ y;
        while(y != 0){
            y &= y -1;
            count++;
        }
        return count;
    }
};
```

P657 机器人能否返回原点

在二维平面上，有一个机器人从原点 (0, 0) 开始。给出它的移动顺序，判断这个机器人在完成移动后是否在 (0, 0) 处结束。

移动顺序由字符串表示。字符 move[i] 表示其第 i 次移动。机器人的有效动作有 R（右），L（左），U（上）和 D（下）。如果机器人在完成所有动作后返回原点，则返回 true。否则，返回 false。

注意：机器人“面朝”的方向无关紧要。 “R” 将始终使机器人向右移动一次，“L” 将始终向左移动等。此外，假设每次移动机器人的移动幅度相同。

示例:

```
输入：“UD”
输出：true
解释：机器人向上移动一次，然后向下移动一次。所有动作都具有相同的幅度，因此它最终回到它开始的原点。因此，我们返回 true。

输入：“LL”
输出：false
解释：机器人向左移动两次。它最终位于原点的左侧，距原点有两次 “移动” 的距离。我们返回 false，因为它在移动结束时没有返回原点。
```

思路

- 统计各个字符的数量，‘R’和‘L’对比，‘U’和‘D’对比

Solution

```
class Solution {
public:
    bool judgeCircle(string moves) {
         int R_nums = 0;
    int L_nums = 0;
    int U_nums = 0;
    int D_nums = 0;

    for (int i=0; i<moves.length(); i++)
    {
        if (moves[i] == 'R')    R_nums++;
        if (moves[i] == 'L')    L_nums++;
        if (moves[i] == 'U')    U_nums++;
        if (moves[i] == 'D')    D_nums++;
    }

    if ((R_nums == L_nums)  && (U_nums == D_nums))
        return true;
    else
        return false;
    }
};
```

P226 翻转二叉树

翻转一棵二叉树。

实例：

```
         4
   /   \
  2     7
 / \   / \
1   3 6   9

     4
   /   \
  7     2
 / \   / \
9   6 3   1
```

思路

- 递归翻转

Solution

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        TreeNode* temp;
        if(root==NULL){return NULL;}
        temp=root->left;
        root->left=root->right;
        root->right=temp;
        invertTree(root->left);
        invertTree(root->right);
        return  root;
    }
};
```

P1051 高度检查器

学校在拍年度纪念照时，一般要求学生按照 非递减 的高度顺序排列。

请你返回至少有多少个学生没有站在正确位置数量。该人数指的是：能让所有学生以 非递减 高度排列的必要移动人数。

示例:

```
输入：[1,1,4,2,1,3]
输出：3
解释：
高度为 4、3 和最后一个 1 的学生，没有站在正确的位置。
```

思路

- 笨方法，拷贝一份数组，将原数组排序，与拷贝数组比较，有几个不一样的元素

Solution

```
class Solution {
public:
        int heightChecker(vector<int>& heights) {
                int ans = 0;
                int len = heights.size();
                vector<int> copy(len);
                for (int i = 0; i < len; i++)
                        copy[i] = heights[i];
                sort(heights.begin(), heights.end());
                for (int i = 0; i < len; i++) {
                        if (heights[i] != copy[i])
                                ans++;
                }
                return ans;
        }
};
```

P977 有序数组的平方

给定一个按非递减顺序排序的整数数组 `A`，返回每个数字的平方组成的新数组，要求也按非递减顺序排序。

实例：

```
输入：[-4,-1,0,3,10]
输出：[0,1,9,16,100]

输入：[-7,-3,2,3,11]
输出：[4,9,9,49,121]
```

思路

- 平方再排序为常规做法，不过忽略了有序条件

由于数组有序，较大平方必然位于两侧，故而采用双指针法，一头一尾，遇到较大平方则插入并移动指针

Solution

```
class Solution {
public:
    vector<int> sortedSquares(vector<int>& A) {
        int length = A.size();
        vector<int> ans(length);
        int i = 0, j = length - 1,k = length - 1;
        while (k >= 0) {
            int a = A[i] * A[i];
            int b = A[j] * A[j];
            if (a > b) {
                ans[k] = a;
                i++;
            }
            else {
                ans[k] = b;
                j--;
            }
            k--;
        }
        return ans;
    }
};
```

P104 二叉树的最大深度

给定一个二叉树，找出其最大深度。

二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

说明: 叶子节点是指没有子节点的节点。

示例:

```
给定二叉树 [3,9,20,null,null,15,7]
    3
   / \
  9  20
    /  \
   15   7

返回它的最大深度 3
```

思路

- 递归累加求最大深度

Solution

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    int maxDepth(TreeNode* root) {
        // 当根结点为空
        if(root == NULL) {
            return 0;
        }
        // 当左右节点均为空（叶子节点），返回1
        if((root->left == NULL) && (root->right == NULL)) {
            return 1;
        }
        // 递归：每个根节点的最大深度=左右节点最大深度的最大值  +  1 ，这里在累加
        return max(maxDepth(root->left), maxDepth(root->right)) + 1;
    }
};
```

P942 增减字符串匹配

给定只含 "I"（增大）或 "D"（减小）的字符串 S ，令 N = S.length。

返回 [0, 1, ..., N] 的任意排列 A 使得对于所有 i = 0, ..., N-1，都有：

- 如果 S[i] == "I"，那么 A[i] < A[i+1]
- 如果 S[i] == "D"，那么 A[i] > A[i+1]

实例：

```
输出："IDID"
输出：[0,4,1,3,2]

输出："III"
输出：[0,1,2,3]

输出："DDI"
输出：[3,2,0,1]
```

思路

- 双指针，大小指针分别指向big=S.size()和small=0
- 若字符为‘I’,则压入small,同时small加1；若字符为‘D’,则压入big,同时big加1
- 最后特殊处理最后一个字符

Solution

```
class Solution {
public:
    vector<int> diStringMatch(string S) {
        vector<int> res;
        if(S.empty())
            return res;
        int small=0,big=S.size();
        for(int i=0;i<S.size();i++)
        {
            if(S[i]=='I')
            {
                res.push_back(small);
                small++;
            }
            if(S[i]=='D')
            {
                res.push_back(big);
                big--;
            }
        }
        //最后一个字符需要特殊处理
        if(S[S.size()-1]=='I')
            res.push_back(small);
        else
            res.push_back(big);
        return res;
    }
};
```

P589 N叉树的前序遍历

给定一个 N 叉树，返回其节点值的*前序遍历*。

例如，给定一个 `3叉树` ：

1

/\\

324

/\

56

返回其前序遍历: `[1,3,5,6,2,4]`。

示例:

```
无
```

思路

- 递归求解

Solution

```
/*
// Definition for a Node.
class Node {
public:
    int val;
    vector<Node*> children;

    Node() {}

    Node(int _val, vector<Node*> _children) {
        val = _val;
        children = _children;
    }
};
*/
class Solution
{
public:
    vector<int> preorder(Node *root)
    {
        vector<int> V;
        if (!root)
            return V;
        V.push_back(root->val);
        for (auto node : root->children)//node是root的所有孩子结点
        {
            for (auto num : preorder(node))//所有的孩子结点再递归生成vector，而num是这个孩子结点vector中的每个元素
            {
                V.push_back(num);
            }
        }
        return V;
    }
};
```

P728 自除数

- 自除数 是指可以被它包含的每一位数除尽的数。

  例如，128 是一个自除数，因为 128 % 1 == 0，128 % 2 == 0，128 % 8 == 0。

  还有，自除数不允许包含 0 。

  给定上边界和下边界数字，输出一个列表，列表的元素是边界（含边界）内所有的自除数。

实例：

```
输入：
上边界left = 1, 下边界right = 22
输出： [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 15, 22]
```

思路

- 挨个遍历，暴力判断

Solution

```
class Solution {
public:
    vector<int> selfDividingNumbers(int left, int right){
        vector<int> res; //结果数组
        for(int i=left; i<=right; i++){
            int num = i;
            int flag = 0; // 是否是自除数标志变量，0不是，1是
            while(num){
                int digit = num % 10;
                if( digit != 0 && i%digit == 0 ){
                    flag = 1;
                    num /= 10;
                }else{
                    flag = 0;
                    break;
                }
            }
            if(flag){
                res.push_back(i);
            }
        }
        return res;
    }
};
```

P589 N叉树的后序遍历

给定一个 N 叉树，返回其节点值的*后序遍历*。

例如，给定一个 `3叉树` :

1

/\\

324

/\

56

返回其前序遍历: `[5,6,3,2,4,1]`。

示例:

```
无
```

思路

- 递归

Solution

```
class Solution {
public:
    vector<int> ret;
    void post(Node* root){
        if(root==NULL) return;
        for(int i = 0;i<root->children.size();++i){
            post(root->children[i]);
        }
        ret.push_back(root->val);
    }
    vector<int> postorder(Node* root) {
        post(root);
        return ret;
    }
};
```

P700 二叉搜索树中的搜索

- 给定二叉搜索树（BST）的根节点和一个值。 你需要在BST中找到节点值等于给定值的节点。 返回以该节点为根的子树。 如果节点不存在，则返回 NULL。

实例：

给定二叉搜索树：

```
        4
       / \
      2   7
     / \
    1   3
```

和值：2

```
      2
     / \
    1   3
```

思路

- 递归

Solution

```
class Solution
{
public:
    TreeNode *searchBST(TreeNode *root, int val)
    {
        if (!root)
        {
            return root;
        }
        if (val > root->val)
        {
            return searchBST(root->right, val);
        }
        else if (val < root->val)
        {
            return searchBST(root->left, val);
        }
        else
            return root;
    }
};
```

P292 Nim游戏

你和你的朋友，两个人一起玩 Nim 游戏：桌子上有一堆石头，每次你们轮流拿掉 1 - 3 块石头。 拿掉最后一块石头的人就是获胜者。你作为先手。

你们是聪明人，每一步都是最优解。 编写一个函数，来判断你是否可以在给定石头数量的情况下赢得游戏。

示例:

```
输入： 4
输出： false
解释： 如果堆中有 4 块石头，那么你永远不会赢得比赛；
      因为无论你拿走 1 块、2 块 还是 3 块石头，最后一块石头总是会被你的朋友拿走。
```

思路

- 只要轮到你的石头数量不是4的倍数，意味着 `n%4!=0` 时就胜利

Solution

```
class Solution {
public:
    bool canWinNim(int n) {
        return n%4 !=0;
    }
};
```

P852 山脉数组的峰顶索引

- 我们把符合下列属性的数组 A 称作山脉：
  - A.length >= 3
  - 存在 0 < i < A.length - 1 使得A[0] < A[1] < ... A[i-1] < A[i] > A[i+1] > ... > A[A.length - 1]
- 给定一个确定为山脉的数组，返回任何满足 A[0] < A[1] < ... A[i-1] < A[i] > A[i+1] > ... > A[A.length - 1] 的 i 的值。

实例：

```
输入：[0,1,0]
输出：1

输入：[0,2,1,0]
输出：1
```

思路

- 遍历判断即可

Solution

```
class Solution {
```

```
public:
    int peakIndexInMountainArray(vector<int>& A) {
        int ans=0;
        int len=A.size();
        for(int i=1;i<len;i++)
        {
            int j=0,k=0;
            for(j=i+1;j<len;j++)
            {
                if(A[i]<A[j])
                {break;}
            }

            for(k=i+1;k>=0;k--)
            {
                if(A[k]>A[i])
                {break;}
            }
            if(j==len&&k==-1)
            {
                ans=i;
                break;
            }
        }
        return ans;
    }
};
```

P476 数字的补数

给定一个正整数，输出它的补数。补数是对该数的二进制表示取反。

注意:

1. 给定的整数保证在32位带符号整数的范围内。
2. 你可以假定二进制数不包含前导零位。

示例:

```
输入：5
输出：2
解释：5的二进制表示为101（没有前导零位），其补数为010。所以你需要输出2。

输入：1
输出：0
解释：1的二进制表示为1（没有前导零位），其补数为0。所以你需要输出0。
```

思路

- 5的二进制是：0101，7的二进制是： 0111，它们的抑或为：0010，去掉前导零位即为取反。再来一个例子，假设a为1110 0101，b为1111 1111，a^b = 0001 1010是a的取反。也就是说二进制位数与num相同，且全为1的数tmp与num的抑或即为所求。

Solution

```
class Solution {
public:
    int findComplement(int num) {
        int tmp = 1;
        while (tmp < num)
        {
            tmp <<= 1;
            tmp += 1;
        }
        return (tmp^num);
    }
};
```

P1025 除数博弈

爱丽丝和鲍勃一起玩游戏，他们轮流行动。爱丽丝先手开局。

最初，黑板上有一个数字 N 。在每个玩家的回合，玩家需要执行以下操作：

- 选出任一 x，满足 0 < x < N 且 N % x == 0 。
- 用 N - x 替换黑板上的数字 N 。
- 如果玩家无法执行这些操作，就会输掉游戏。

只有在爱丽丝在游戏中取得胜利时才返回 True，否则返回 false。假设两个玩家都以最佳状态参与游戏。

实例：

```
输入：2
输出：true
解释：爱丽丝选择 1，鲍勃无法进行操作。

输入：3
输出：false
解释：爱丽丝选择 1，鲍勃也选择 1，然后爱丽丝无法进行操作。
```

思路

- 直接分析
- 动态规划

Solution

```
class Solution {
public:
    bool divisorGame(int N) {
        int count = 0;
        int num = N;
        while(num != 1) {
            int x = 1;
            while(num % x != 0) {
                x++;
            }
            num -= x;
            count++;
        }
        if(count % 2 == 0) {
            return false;
        } else {
            return true;
        }
    }
};
```

P344 反转字符串

编写一个函数，其作用是将输入的字符串反转过来。输入字符串以字符数组 char[] 的形式给出。

不要给另外的数组分配额外的空间，你必须原地修改输入数组、使用 O(1) 的额外空间解决这一问题。

你可以假设数组中的所有字符都是 ASCII 码表中的可打印字符。

示例:

```
输入：["h","e","l","l","o"]
输出：["o","l","l","e","h"]

输入：["H","a","n","n","a","h"]
输出：["h","a","n","n","a","H"]
```

思路

- 原地修改和O(1)空间是这道题的难点
- 双指针，交换头尾两个指针所指的两个位置的值，指针向中间移动一个位置，重复以上操作直到两个指针交错

Solution

```
class Solution {
public:
    void reverseString(vector<char>& s) {
        int i = 0;
        int j = s.size() - 1;
        while(i<j)
        {
            swap(s[i],s[j]);
            ++ i;
            -- j;
        }
    }
};
```

P500 键盘行

给定一个单词列表，只返回可以使用在键盘同一行的字母打印出来的单词。键盘如下图所示。

实例：

```
输入: ["Hello", "Alaska", "Dad", "Peace"]
输出: ["Alaska", "Dad"]
```

思路

- 建立map映射即可
- flag = 1表示在第一行； flag = 2第二行； flag = 3第三行； flag在123之间改变，表明不在同一行，结束当前单词； vector<string> findWords(vector<string>& words) { set<char> dic1 = {'Q','W','E','R','T','Y','U','I','O','P'}; set<char> dic2 = {'A','S','D','F','G','H','J','K','L'}; set<char> dic3 = {'Z','X','C','V','B','N','M'};

Solution

```
class Solution {
public:
    vector<string> findWords(vector<string>& words)
    {
        string q{"qwertyuiop"};
        string a{"asdfghjkl"};
        string z{"zxcvbnm"};
        vector<string> ans;

        for(int i=0;i<words.size();i++)
        {
            int d=0,b=0,c=0;
            for(int j=0;j<words[i].size();j++)
            {
```

```
                if(q.find(tolower(words[i][j]))!=string::npos) b++;
                if(a.find(tolower(words[i][j]))!=string::npos) c++;
                if(z.find(tolower(words[i][j]))!=string::npos) d++;
            }
            if(b==words[i].size()||c==words[i].size()||d==words[i].size())
                ans.push_back(words[i]);
        }
        return ans;
    }
};
```

P557 反转字符串中的单词 III

给定一个字符串，你需要反转字符串中每个单词的字符顺序，同时仍保留空格和单词的初始顺序。

示例:

```
输入："Let's take LeetCode contest"
输出："s'teL ekat edoCteeL tsetnoc"
```

思路

- 思路很简单，按照空格切分，然后逆转每个单词 但是有个值得注意的地方，直接使用原字符串的话，首末单词判定的时候会出现特殊情况 比如，如果按照单词后的空格作为切分标志，最后一个单词是特殊情况，末尾没有空格 如果按照单词前面的空格作为切分标志，首单词前没有空格 所以，这里思路很简单，先给原字符串末尾加一个空格，这样就能够统一处理所有单词 按照单词尾部的空格为标志，逆转所有单词 完事儿之后还要记得把字符串末尾的空格删掉 代码还能写得更简洁，把自增自减运算符放在运算里面，然后去掉大括号 但是这样会导致可读性变差 所以代码就这样了，简单易懂

Solution

```
class Solution {
public:
    string reverseWords(string s) {
        int pos=0;
        int left=0,right=0;
        s.push_back(' ');
        int len=s.length();
        while(pos<len){
            if(s[pos]==' '){
                right=pos-1;
                while(left<right){
                    swap(s[left], s[right]);
                    ++left;
                    --right;
                }
```

```
                left=pos+1;
            }
            ++pos;
        }
        s.erase(s.end()-1);
        return s;
    }
};
```

P944 删列造序

给定由 N 个小写字母字符串组成的数组 A，其中每个字符串长度相等。

删除 操作的定义是：选出一组要删掉的列，删去 A 中对应列中的所有字符，形式上，第 n 列为 [A[0][n], A[1][n], ..., A[A.length-1][n]]）。

比如，有 A = ["abcdef", "uvwxyz"],

要删掉的列为 `{0, 2, 3}`，删除后 `A` 为 `["bef", "vyz"]`， `A` 的列分别为 `["b","v"], ["e","y"], ["f","z"]`。

你需要选出一组要删掉的列 `D`，对 `A` 执行删除操作，使 `A` 中剩余的每一列都是 **非降序** 排列的，然后请你返回 `D.length` 的最小可能值。

实例：

```
输入：["cba", "daf", "ghi"]
输出：1
解释：
当选择 D = {1}，删除后 A 的列为：["c","d","g"] 和 ["a","f","i"]，均为非降序排列。
若选择 D = {}，那么 A 的列 ["b","a","h"] 就不是非降序排列了。

输入：["a", "b"]
输出：0
解释：D = {}

输入：["zyx", "wvu", "tsr"]
输出：3
解释：D = {0, 1, 2}
```

思路

- 题目很绕，其实很简单，字符串一行行摆好，一列列的比较，上一行字母不能大于下一行字母，否则这一列就是需要删除的，记录所需i删除的列数。

Solution

```
class Solution {
public:
```

```
    int minDeletionSize(vector<string>& A) {
        int l = A[0].length(), n = A.size(), count = 0;
        for(int i=0;i<l;i++){
            for(int j=0;j<n-1;j++){
                if(A[j][i]>A[j+1][i]){
                    count++;
                    break;
                }
            }
        }
        return count;
    }
};
```

P559 N叉树的最大深度

给定一个 N 叉树，找到其最大深度。

最大深度是指从根节点到最远叶子节点的最长路径上的节点总数。

示例:

```
无
```

思路

- 递归

Solution

```
/*
// Definition for a Node.
class Node {
public:
    int val;
    vector<Node*> children;

    Node() {}

    Node(int _val, vector<Node*> _children) {
        val = _val;
        children = _children;
    }
};
*/
class Solution {
public:
    int maxDepth(Node* root) {
```

```
    if (!root) return 0;
    int m = 0;
    for (Node* it : root->children)
        m = max(m, maxDepth(it));
    return ++m;
        }
};
```

P561 数组拆分I

给定长度为 2n 的数组, 你的任务是将这些数分成 n 对, 例如 (a1, b1), (a2, b2), ..., (an, bn) ， 使得从1到 n 的 min(ai, bi) 总和最大。

实例：

```
输入: [1,4,3,2]

输出: 4
解释: n 等于 2, 最大总和为 4 = min(1, 2) + min(3, 4).
```

思路

- 先从小到大排序，把奇数位的数字加起来就好了

Solution

```
class Solution {
public:
    int arrayPairSum(vector<int>& nums) {
        int ans = 0;
        sort(nums.begin(), nums.end()); //将数组从小到大排列
        for(int i = 0; i < nums.size(); i = i +2) //将每对的第一位数相加
            ans = ans + nums[i];
        return ans;
    }
};
```

P908 最小差值I

给定一个整数数组 A，对于每个整数 A[i]，我们可以选择任意 x 满足 -K <= x <= K，并将 x 加到 A[i] 中。

在此过程之后，我们得到一些数组 B。

返回 B 的最大值和 B 的最小值之间可能存在的最小差值。

示例:

```
输入：A = [1], K = 0
输出：0
解释：B = [1]

输入：A = [0,10], K = 2
输出：6
解释：B = [2,8]

输入：A = [1,3,6], K = 3
输出：0
解释：B = [3,3,3] 或 B = [4,4,4]
```

思路

- 分别求出B中“最大值的最小值”和“最小值的最大值”

Solution

```
class Solution {
public:
    int smallestRangeI(vector<int>& A, int K) {
        if(A.size() == 0) return 0;
        int maxn = INT_MIN,minn = INT_MAX;
        for(int n : A){
            maxn = max(maxn , n);
            minn = min(minn , n);
        }
        maxn -= K;
        minn += K;
        return (maxn - minn < 0 ? 0 : maxn - minn);
    }
};
```

P108 将有序数组转换为二叉搜索树

将一个按照升序排列的有序数组，转换为一棵高度平衡二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1。

实例：

给定有序数组：[-10,-3,0,5,9],

一个可能的答案是：[0,-3,9,-10,null,5]，它可以表示下面这个高度平衡二叉搜索树：

```
      0
     / \
   -3   9
   /   /
 -10  5
```

思路

- 二分法递归

Solution

```
class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        if(nums.empty()) return nullptr;
        return helper(nums,0,nums.size()-1);
    }

    TreeNode* helper(vector<int>& nums, int left, int right){
        if(left > right)
            return nullptr;
        int mid = (left+ right) /2;
        TreeNode *root = new TreeNode(nums[mid]);
        root->left = helper(nums, left,mid -1);
        root->right = helper(nums,mid+1, right);
        return root;
    }
};
```

P965 单值二叉树

如果二叉树每个节点都具有相同的值，那么该二叉树就是*单值*二叉树。

只有给定的树是单值二叉树时，才返回 `true`；否则返回 `false`。

示例:

```
输入：[1,1,1,1,1,null,1]
输出：true

输入：[2,2,2,5,2]
输出：false
```

思路

- 递归

Solution

```
class Solution
{
public:
    bool isUnivalTree(TreeNode *root)
    {
        if (!root || (!root->left && !root->right)) //如果当前结点为空或者结点的左右子树都为空（没有人可以与其比较），则为true
        {
            return true;
        }
        if (!root->left && root->right) //如果节点的左子树为空，右子树不为空，则先判断其与右子树根节点的值是否相等，再递归判断右子树
        {
            return (root->val == root->right->val) && isUnivalTree(root->right);
        }
        else if (root->left && !root->right) //同理，如果节点的右子树为空，左子树不为空，则先判断其与左子树根节点的值是否相等，再递归判断左子树
        {
            return (root->val == root->left->val) && isUnivalTree(root->left);
        }
        if (root->val == root->left->val && root->val == root->right->val) //如果该节点左子树右子树都不为空且三者值相等，则递归判断其左右子树
        {
            return isUnivalTree(root->left) && isUnivalTree(root->right);
        }
        return false; //不符合以上情况的则返回false
    }
};
```

P349 两个数组的交集

给定两个数组，编写一个函数来计算它们的交集。

实例：

```
输入：nums1 = [1,2,2,1], nums2 = [2,2]
输出：[2]

输入：nums1 = [4,9,5], nums2 = [9,4,9,8,4]
输出：[9,4]
```

思路

- 简单，本题借助了一个bool 数组来完成解题,但是该方法有一个弊端就是你不能确定所给出的最大数据最大是多大所以需要解题人对大概的数据有一个预估

Solution

```
class Solution {
public:
    vector<int> ans;
    vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
        bool vis[9999] = {0};
        for(int i=0;i<nums1.size();i++)vis[nums1[i]] = 1;
        for(int i=0;i<nums2.size();i++)
        if(vis[nums2[i]] == 1){ans.push_back(nums2[i]);vis[nums2[i]] = 0;}
        return ans;
    }
};
```

P171 Excel表序列号

给定一个Excel表格中的列名称，返回其相应的列序号。

例如，

```
A -> 1
B -> 2
C -> 3
...
Z -> 26
AA -> 27
AB -> 28
...
```

示例:

```
输入：“A”
输出：1

输入：“AB”
输出：28
```

思路

- 该题目的意思其实就是26进制转十进制，其中1~26分别用A~Z表示，其实就是个进制转化问题

Solution

```
class Solution {
public:
    int titleToNumber(string s) {
        int sum=0;//结果
        int n=s.size()-1;//阶数
        for(int i=0;i<s.size();i++)
        {
            int temp=pow(26,n)*(s[i]-'A'+1);
            sum+=temp;
            n--;
        }
        return sum;
    }
};
```

P999 车的可用捕获量

在一个 8 x 8 的棋盘上，有一个白色车（rook）。也可能有空方块，白色的象（bishop）和黑色的卒（pawn）。它们分别以字符 “R”，“.”，“B” 和 “p” 给出。大写字符表示白棋，小写字符表示黑棋。

车按国际象棋中的规则移动：它选择四个基本方向中的一个（北，东，西和南），然后朝那个方向移动，直到它选择停止、到达棋盘的边缘或移动到同一方格来捕获该方格上颜色相反的卒。另外，车不能与其他友方（白色）象进入同一个方格。

返回车能够在一次移动中捕获到的卒的数量。

实例：

思路

- 东南西北移动

Solution

```
class Solution {
public:
    int numRookCaptures(vector<vector<char>>& board) {
        //找出R的位置
        int x,y;
        for(int i=0;i<board.size();i++)
            for(int j=0;j<board[i].size();j++)
                if(board[i][j]=='R')
                {
                    x=i;y=j;
                }
        int sum=0;
```

```
            //向北遍历
            for(int i=x;i<8;i++)
            {
                if(board[i][y]=='B')break;
                if(board[i][y]=='p'){
                    sum++;break;
                }
            }
            //向南遍历
            for(int i=x;i>=0;i--)
            {
                if(board[i][y]=='B')break;
                if(board[i][y]=='p'){
                    sum++;break;
                }
            }
            //向东遍历
            for(int j=y;j<8;j++)
            {
               if(board[x][j]=='B')break;
                if(board[x][j]=='p')
                {
                    sum++;break;
                }
            }
            //向西遍历
            for(int j=y;j>=0;j--)
            {
               if(board[x][j]=='B')break;
                if(board[x][j]=='p')
                {
                    sum++;break;
                }
            }
             return sum;
        }
    };
```

P922 按奇偶排序数组 II

给定一个非负整数数组 A， A 中一半整数是奇数，一半整数是偶数。

对数组进行排序，以便当 A[i] 为奇数时，i 也是奇数；当 A[i] 为偶数时， i 也是偶数。

你可以返回任何满足上述条件的数组作为答案。

示例:

```
输入：[4,2,5,7]
输出：[4,5,2,7]
解释：[4,7,2,5]，[2,5,4,7]，[2,7,4,5] 也会被接受。
```

思路

- 首尾双指针

Solution

```
class Solution {
public:
    vector<int> sortArrayByParityII(vector<int>& A) {
        int pre1=1,pre2=0,aLen=A.size();
        while( true ) {
            while(  pre2<aLen &&  !(A[pre2] & 1) ) pre2+=2;//找到 不是偶数但又是偶数索引
            while(  pre1<aLen && A[pre1] & 1  ) pre1+=2;//找到 不是奇数但 是奇数索引
            if( pre1 < aLen || pre2<aLen )
                swap(A[pre1],A[pre2]);
            else
                break;
        }
        return A;
    }
};
```

P821 字符的最短距离

给定一个字符串 `S` 和一个字符 `C`。返回一个代表字符串 `S` 中每个字符到字符串 `S` 中的字符 `C` 的最短距离的数组。

实例：

```
输入：S = "loveleetcode", C = 'e'
输出：[3, 2, 1, 0, 1, 0, 0, 1, 2, 2, 1, 0]
```

思路

- 简单

Solution

```
class Solution {
public:
```

```
    vector<int> shortestToChar(string S, char C) {
        vector<int> distance;
        for(int i = 0; i < S.length(); i++) {
            if(S[i] == C) {
                distance.push_back(0);
                continue;
            }
            vector<int> pos;
            for(int j = 0; j < S.length(); j++) {
                if(S[j] == C) {
                    pos.push_back(abs(i-j));
                }
            }
            int minValue = *min_element(pos.begin(),pos.end());
            distance.push_back(minValue);
        }
        return distance;
    }
};
```

P258 各位相加

给定一个非负整数 `num`，反复将各个位上的数字相加，直到结果为一位数。

示例:

```
输入: 38
输出: 2
解释: 各位相加的过程为: 3 + 8 = 11, 1 + 1 = 2。 由于 2 是一位数，所以返回 2。
```

思路

- 双重while循环即可

Solution

```
class Solution {
public:
    int addDigits(int num) {
        int result = 10;
        while(result >= 10) {
            result = 0;
            while(num != 0) {
                int x = num % 10;
                result += x;
                num = num / 10;
            }
            num = result;
```

```
        }
        returen result;
    }
};
```

P206 反转链表

反转一个单链表。

实例：

```
输入：1->2->3->4->5->NULL
输出：5->4->3->2->1->NULL
```

思路

- 递归

Solution

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if(!head){
            return nullptr;
        }
        return reverse(head, head, head->next);
    }

    ListNode* reverse(ListNode* head, ListNode* first, ListNode* target){
        if(!target){
            return head;
        }
        first->next = target->next;
        ListNode* temp = target->next;
        target->next = head;
        return reverse(target, first, temp);
    }
};
```

P811 子域名访问计数

一个网站域名，如"discuss.leetcode.com"，包含了多个子域名。作为顶级域名，常用的有"com"，下一级则有"leetcode.com"，最低的一级为"discuss.leetcode.com"。当我们访问域名"discuss.leetcode.com"时，也同时访问了其父域名"leetcode.com"以及顶级域名 "com"。

给定一个带访问次数和域名的组合，要求分别计算每个域名被访问的次数。其格式为访问次数+空格+地址，例如："9001 discuss.leetcode.com"。

接下来会给出一组访问次数和域名组合的列表cpdomains 。要求解析出所有域名的访问次数，输出格式和输入格式相同，不限定先后顺序。

示例:

```
示例 1：
输入：
["9001 discuss.leetcode.com"]
输出：
["9001 discuss.leetcode.com", "9001 leetcode.com", "9001 com"]
说明：
例子中仅包含一个网站域名："discuss.leetcode.com"。按照前文假设，子域名"leetcode.com"和"com"都会被访问，所以它们都被访问了9001次。

示例 2
输入：
["900 google.mail.com", "50 yahoo.com", "1 intel.mail.com", "5 wiki.org"]
输出：
["901 mail.com","50 yahoo.com","900 google.mail.com","5 wiki.org","5 org","1 intel.mail.com","951 com"]
说明：
按照假设，会访问"google.mail.com" 900次，"yahoo.com" 50次，"intel.mail.com" 1次，"wiki.org" 5次。
而对于父域名，会访问"mail.com" 900+1 = 901次，"com" 900 + 50 + 1 = 951次，和 "org" 5 次。
```

思路

- 处理字符串

Solution

```
class Solution {
public:
    vector<string> subdomainVisits(vector<string>& cpdomains) {
        map<string,int> domain;
        vector<string> output;
        for(int i=0; i<cpdomains.size(); i++){
            size_t pos1 = cpdomains[i].find(" ");//找出空格的位置
```

```
            string temp = cpdomains[i].substr(0, pos1);//截取出表示访问次数的
字符串
            int count = atoi(temp.c_str());//转为int
            if(pos1!=cpdomains[i].npos){
                string temp =
cpdomains[i].substr(pos1+1,cpdomains[i].size()-1);//截取出最低一级的域名
                //output.push_back(temp);
                domain[temp] += count;//向map中插入最低一级的域名
                //cout << temp << endl;
            }
            size_t pos2 = cpdomains[i].find(".");//找出"."的位置
            while(pos2!=cpdomains[i].npos){
                string temp =
cpdomains[i].substr(pos2+1,cpdomains[i].size()-1);//依次截取高级域名
                domain[temp] += count;//向map中插入域名
                pos2 = cpdomains[i].find(".", pos2+1);//查找下一个"."
                //cout << temp << endl;
            }
        }
        map<string,int>::iterator it;
        for(it = domain.begin(); it!=domain.end(); it++){
            string temp = std::to_string(it->second) + " " + it->first;//注
意将int转为string
            output.push_back(temp);
            //cout << it->first << ":" << it->second << endl;
        }
        //cout << domain.size() <<endl;
        return output;
    }
};
```

P682 棒球比赛

你现在是棒球比赛记录员。 给定一个字符串列表，每个字符串可以是以下四种类型之一： 1.整数（一轮的得分）：直接表示您在本轮中获得的积分数。

1. "+"（一轮的得分）：表示本轮获得的得分是前两轮有效 回合得分的总和。
2. "D"（一轮的得分）：表示本轮获得的得分是前一轮有效 回合得分的两倍。
3. "C"（一个操作，这不是一个回合的分数）：表示您获得的最后一个有效 回合的分数是无效的，应该被移除。

每一轮的操作都是永久性的，可能会对前一轮和后一轮产生影响。 你需要返回你在所有回合中得分的总和。

实例：

```
输入：["5","2","C","D","+"]
输出：30
解释：
```

```
第1轮：你可以得到5分。总和是：5。
第2轮：你可以得到2分。总和是：7。
操作1：第2轮的数据无效。总和是：5。
第3轮：你可以得到10分（第2轮的数据已被删除）。总数是：15。
第4轮：你可以得到5 + 10 = 15分。总数是：30。

输入：["5","-2","4","C","D","9","+","+"]
输出：27
解释：
第1轮：你可以得到5分。总和是：5。
第2轮：你可以得到-2分。总数是：3。
第3轮：你可以得到4分。总和是：7。
操作1：第3轮的数据无效。总数是：3。
第4轮：你可以得到-4分（第三轮的数据已被删除）。总和是：-1。
第5轮：你可以得到9分。总数是：8。
第6轮：你可以得到-4 + 9 = 5分。总数是13。
第7轮：你可以得到9 + 5 = 14分。总数是27。
```

思路

- 理解题意即可

Solution

```cpp
class Solution {
public:
    int calPoints(vector<string>& ops) {
        if(ops.size() == 0) return 0;
        int ans = 0;
        stack<int> s;
        for (int i=0; i<ops.size(); ++i) {
        if (ops[i] == "+") {
            int top = s.top();
            s.pop();
            int newtop = top + s.top();
            s.push(top);
            s.push(newtop);
        }
        else if (ops[i] == "D")
            s.push(s.top() * 2);
        else if (ops[i] == "C"){
            s.pop();
        }
        else s.push(stoi(ops[i]));
        }
        while(!s.empty()){
            ans += s.top();
            s.pop();
```

```
        }
        return ans;
    }
};
```

P118 杨辉三角

给定一个非负整数 *numRows*，生成杨辉三角的前 *numRows* 行

示例:

```
输入：5
输出：
[
     [1],
    [1,1],
   [1,2,1],
  [1,3,3,1],
 [1,4,6,4,1]
]
```

思路

- 简单

Solution

```
class Solution {
public:
    vector<vector<int> > generate(int numRows) {
        vector<vector<int> > vs;
        for(int row=0; row<numRows; ++row){
            vector<int> v(row+1,1);
            for(int i=0; ++i<row; v[i]=vs[row-1][i-1]+vs[row-1][i]);
            vs.push_back(v);
        }
        return vs;
    }
};
```

P1002 查找常用字符

给定仅有小写字母组成的字符串数组 A，返回列表中的每个字符串中都显示的全部字符（包括重复字符）组成的列表。例如，如果一个字符在每个字符串中出现 3 次，但不是 4 次，则需要在最终答案中包含该字符 3 次。

你可以按任意顺序返回答案。

实例：

```
输入：["bella","label","roller"]
输出：["e","l","l"]

输入：["cool","lock","cook"]
输出：["c","o"]
```

思路

- 用第一个单词的字母，遍历其他的单词即可

Solution

```
class Solution {
public:
    vector<string> commonChars(vector<string>& A) {
        vector<string> ans;
        int num = A.size();
        int count[26];

        string use[26] = {
"a","b","c","d","e","f","g","h","i","j","k","l","m","n","o","p","q","r","s"
,"t","u","v","w","x","y","z" };

        for (int i = 0; i < 26; i++)
            count[i] = 101;
        for (int i = 0; i < num; i++) {
            int word[26] = { 0 };
            int len = A[i].length();
            for (int j = 0; j < len; j++)
                word[(int)A[i][j] - 97]++;
            for (int j = 0; j < 26; j++)
                if (count[j]) {
                    count[j] = count[j] < word[j] ? count[j] : word[j];
                }
        }
        for (int i = 0; i < 26; i++) {
            if (count[i]) {
                for (int j = 0; j < count[i]; j++) {
                    string s(1, (char)(i + 97));
                    ans.push_back(s);
                }
            }
        }
        return ans;
    }
};
```

P762 二进制表示中质数个计算置位

给定两个整数 L 和 R ，找到闭区间 [L, R] 范围内，计算置位位数为质数的整数个数。

（注意，计算置位代表二进制表示中1的个数。例如 21 的二进制表示 10101 有 3 个计算置位。还有，1 不是质数。）

示例:

```
输入: L = 6, R = 10
输出: 4
解释:
6 -> 110 (2 个计算置位，2 是质数)
7 -> 111 (3 个计算置位，3 是质数)
9 -> 1001 (2 个计算置位，2 是质数)
10-> 1010 (2 个计算置位，2 是质数)

输入: L = 10, R = 15
输出: 5
解释:
10 -> 1010 (2 个计算置位，2 是质数)
11 -> 1011 (3 个计算置位，3 是质数)
12 -> 1100 (2 个计算置位，2 是质数)
13 -> 1101 (3 个计算置位，3 是质数)
14 -> 1110 (3 个计算置位，3 是质数)
15 -> 1111 (4 个计算置位，4 不是质数)
```

思路

- 思路简单

Solution

```
class Solution {
public:
    int countPrimeSetBits(int L, int R) {
        int ans=0;
        for(int i=L;i<=R;i++)
        {
            int tmpNum=i,tmpCnt=0;
            while(tmpNum!=0)
            {
                tmpNum&=tmpNum-1;
                tmpCnt++;
            }
            if(isPrime(tmpCnt)) ans++;
        }
        return ans;
```

```
    }
    bool isPrime(int num)
    {
        if(num<2) return false;
        for(int i=2;i*i<=num;i++)
        {
            if(num%i==0) return false;
        }
        return true;
    }
};
```

P521 最长特殊序列 I

给定两个字符串，你需要从这两个字符串中找出最长的特殊序列。最长特殊序列定义如下：该序列为某字符串独有的最长子序列（即不能是其他字符串的子序列）。

子序列可以通过删去字符串中的某些字符实现，但不能改变剩余字符的相对顺序。空序列为所有字符串的子序列，任何字符串为其自身的子序列。

输入为两个字符串，输出最长特殊序列的长度。如果不存在，则返回 -1。

实例：

```
输入: "aba", "cdc"
输出: 3
解析: 最长特殊序列可为 "aba" (或 "cdc")
```

思路

- 两个字符串不相同,就是特殊,返回长的；相同就没有，返回-1；

Solution

```
class Solution {
public:
    int findLUSlength(string a, string b) {
        return (a == b) ? -1 : (a.size() > b.size() ? a.size() : b.size());
    }
};
```

P136 只出现一次的数字

给定一个**非空**整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

示例:

```
输入：[2,2,1]
输出：1

输入：[4,1,2,1,2]
输出：4
```

思路

- 若第一次出现，插入哈希集
- 第二次出现，冲哈希集内删除
- 最后剩下的就是那个只出现一次的数字

Solution

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        unordered_set<int> bobo;
        int ans;
        for(auto i : nums){
            if(bobo.count(i))   bobo.erase(i);
            else    bobo.insert(i);
        }
        for(auto j : bobo)  ans = j;
        return ans;
    }
};
```

P575 分糖果

给定一个偶数长度的数组，其中不同的数字代表着不同种类的糖果，每一个数字代表一个糖果。你需要把这些糖果平均分给一个弟弟和一个妹妹。返回妹妹可以获得的最大糖果的种类数。

实例：

```
输入：candies = [1,1,2,2,3,3]
输出：3
解析：一共有三种种类的糖果，每一种都有两个。
      最优分配方案：妹妹获得[1,2,3],弟弟也获得[1,2,3]。这样使妹妹获得糖果的种类数最多。

输入：candies = [1,1,2,3]
输出：2
解析：妹妹获得糖果[2,3],弟弟获得糖果[1,1]，妹妹有两种不同的糖果，弟弟只有一种。这样使得妹妹可以获得的糖果种类数最多。
```

思路

- 当糖果种类大于数组大小的一半时（len/2），妹妹可以任选(len/2)种的糖果;
- 当糖果种类(c)小于数组大小的一半时（len/2），妹妹最多可以任选c种的糖果.
- 所以计算糖果的种类即可，借助set结构元素不同的特性，即可统计糖果的种类

Solution

```
class Solution {
public:
    int distributeCandies(vector<int>& candies) {
        if(candies.empty())
            return 0;
        int len=candies.size();
        set<int> s;
        for(int i=0;i<len;i++)
        {
            s.insert(candies[i]);
        }
        int c=s.size();
        if(c>=len/2)
            return len/2;
        else
            return c;
    }
};
```

P429 N叉树的层次遍历

给定一个 N 叉树，返回其节点值的*层序遍历*。 (即从左到右，逐层遍历)。

实例:

```
无
```

思路

- 迭代

Solution

```
class Solution {
public:
    vector<vector<int>> levelOrder(Node* root) {
        if (root == NULL) {
            return vector<vector<int>> ();
        }
        vector<vector<int>> res;
        LevelOrderCore(root, res, 0);
```

```
        return res;
    }
    void LevelOrderCore(Node* root, vector<vector<int>>& res, int index) {
        if (root == NULL) {
            return ;
        }
        //如果当前层数大于已经存储的vector 大小，则新分配一个内层vector
        if (index >= res.size()) {
            res.push_back(vector<int>());
        }
        res[index].push_back(root->val);
        for (auto i = 0; i != root->children.size(); ++i) {
            LevelOrderCore(root->children[i], res, index + 1);
        }
    }
};
```